Improving MPI Collective I/O for High Volume Non-Contiguous Requests With Intra-Node Aggregation

Qiao Kang, Sunwoo Lee, Kaiyuan Hou, Robert Ross, Ankit Agrawal,
Alok Choudhary, *Fellow, IEEE*, and Wei-keng Liao

Abstract—Two-phase I/O is a well-known strategy for implementing collective MPI-IO functions. It redistributes I/O requests among the calling processes into a form that minimizes the file access costs. As modern parallel computers continue to grow into the exascale era, the communication cost of such request redistribution can quickly overwhelm collective I/O performance. This effect has been observed from parallel jobs that run on multiple compute nodes with a high count of MPI processes on each node. To reduce the communication cost, we present a new design for collective I/O by adding an extra communication layer that performs request aggregation among processes within the same compute nodes. This approach can significantly reduce inter-node communication contention when redistributing the I/O requests. We evaluate the performance and compare it with the original two-phase I/O on Cray XC40 parallel computers (Theta and Cori) with Intel KNL and Haswell processors. Using I/O patterns from two large-scale production applications and an I/O benchmark, we show our proposed method effectively reduces the communication cost and hence maintains the scalability for a large number of processes.

Index Terms—Parallel I/O, MPI collective I/O, two-phase I/O, non-contiguous I/O

1 Introduction

The message passing interface (MPI) standard defines a set of programming interfaces for parallel shared-file access, commonly denoted as MPI-IO [1]. Many large-scale scientific applications adopt MPI-IO directly or indirectly through parallel I/O libraries to obtain high I/O performance [2], [3], [4], [5], [6]. There are two types of MPI-IO functions: collective and independent. The collective functions require all processes that collectively open the same shared file to participate in the calls. Such a requirement provides an opportunity for an MPI-IO implementation to coordinate activities between processes to achieve better performance. A well-known example is the two-phase I/O strategy [7], which has become the implementation backbone for collective I/O in almost all MPI libraries.

Two-phase I/O conceptually consists of a communication phase and an I/O phase. A subset of the MPI processes, defined as I/O aggregators, act as I/O proxies for the rest of the processes. The aggregate access file region of a collective I/O call is divided among the aggregators into nonoverlapping regions, called file domains. In the communication phase, all processes send their I/O requests to the aggregators based on their file domain assignments. In the I/O phase, aggregators make system calls to read from or write the received requests to the file. The two-phase I/O strategy has delivered high performance on parallel machines in the past two decades. The success of a two-phase I/O strategy relies on a fast communication network by paying a relatively small cost on exchanging request data among processes to obtain a higher gain in file system access time. This tradeoff works effectively as the speed of I/O systems is much slower than the communication systems. However, as the scale of parallel computers grows, soon into an exascale computing era, the number of processes running applications also increases. The communication of the two-phase I/O exhibits an all-to-many message exchange pattern, whose cost may exceed the I/O phase for large parallel jobs when the number of non-contiguous I/O requests is significant, due to the high possible communication congestion on the I/O aggregators [8], [9].

In this paper, we present an improvement for MPI collective I/O, denoted as the two-layer aggregation method (TAM), which adds an intra-node request aggregation layer so that the communication in the two-phase strategy consists of two layers of request aggregations. In the intra-node aggregation layer, MPI processes running on the same compute nodes perform a request aggregation to a subset of processes, denoted as local aggregators. In contrast to local aggregators, we denote the I/O aggregators in the original two-phase I/O as the global aggregators. As communication takes place among processes on the same node, the cost is expected to be relatively small. This intra-node aggregation performs on all

- *Q. Kang, S. Lee, K. Hou, A. Agrawal, A. Choudhary, and W.-k. Liao are with the Department of Electrical and Computer Engineering, Northwestern Univeristy, Evanston, IL 60208. E-mail: {qiao.kang, slz839, khl7265, ankitag, choudhar, wkliao}@ece.northwestern.edu.*
- *R. Ross is with the Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439.
E-mail: rross@mcs.anl.gov.*

*Manuscript received 31 Dec. 2019; revised 17 May 2020; accepted 2 June 2020. Date of publication 5 June 2020; date of current version 19 June 2020.
(Corresponding author: Qiao Kang.)
Recommended for acceptance by R. Tolosana.
Digital Object Identifier no. 10.1109/TPDS.2020.3000458*

compute nodes independently and concurrently. Once receiving the requests from local processes, the local aggregators coalesce them into fewer contiguous requests. After coalescing non-contiguous I/O requests, the local and global aggregators across compute nodes enter the traditional two-phase I/O to complete collective read or write operations. In this paper, we refer to the communication between local and global aggregators as the inter-node aggregation. An advantage of TAM is the reduction of the number of inter-node communications at global aggregators since the number of local aggregators is much less than the number of processes. In the traditional two-phase I/O, each global aggregator may receive requests from all other MPI processes, potentially causing communication contention at the global aggregators for large-scale applications with a significant number of non-contiguous I/O requests. The intra-node aggregation alleviates such a problem by breaking the all-to-many communication into two layers so that the global aggregators receive requests only from the local aggregators.

We implement TAM in ROMIO, the implementation of the MPI-IO functions used most frequently in HPC and provided by vendors as part of their MPI implementation [10]. Our performance evaluations are conducted on Theta, Cray XC40 parallel computer with Intel KNL processors at the Argonne National Laboratory, and Cori, a Cray XC40 supercomputer with Intel Haswell processors at the National Energy Research Scientific Computing Center (NERSC). Comparisons of TAM against the traditional two-phase I/O from the latest implementation of ROMIO are presented using three I/O benchmarks: E3SM-IO [5], S3D-IO [11], and BTIO [12]. E3SM-IO and S3D-IO are I/O kernels of two large-scale production applications E3SM and S3D, respectively, while BTIO is a benchmark from NASA's NAS Parallel Benchmarks. These benchmarks contain a vast number of non-contiguous I/O requests that result in inter-node communication congestion for two-phase I/O as the number of nodes scale. From the experimental results, we observe a significant time reduction in communication costs. The intra-node communication cost, which is the extra cost introduced by TAM, vanishes as the number of processes increase, which matches our expectation. We analyze how the choice of the number of TAM local aggregators affects the performance from the inter-node communication network utilization and congestion point of view. The overall performance improvements are up to 29 and 6.7 times faster for collective write and read.

In Section 2, we discuss the two-phase I/O bottlenecks and existing solutions. In Section 3, we propose our TAM improvement for two-phase I/O in detail. Finally, we evaluate the TAM in Section 4.

2 Background

The implementation of two-phase I/O [13] in ROMIO selects a subset of MPI processes, denoted as I/O aggregators, as proxies to carry out the file access operations for the remaining processes. The aggregate access file region of a collective I/O call is divided among the aggregators into nonoverlapping regions, called file domains. Each aggregator is responsible for reading and writing for its file domain assigned to the file. When rearranging the requests from non-aggregators to aggregators, also known as the communication phase, aggregators gather I/O requests that intersect its file domain. In the I/O phase, aggregators coalesce the received requests and read/write data from/to the file system.

Many MPI libraries such as MPICH [14] and OpenMPI [15] adopt ROMIO as the implementation for MPI-IO functions. For parallel jobs running multiple MPI processes per compute nodes, ROMIO selects one aggregator per node in its default settings. The selection of I/O aggregators happens at file open time.

Due to the fact of I/O devices being the slowest hardware component of a parallel computer system, the I/O phase is often the bottleneck of collective I/O. Many strategies have been proposed in the past decades to reduce its cost. Ma improved MPI-IO output performance with active buffering and threads [16]. A strategy that aligns the file domains with the file locking protocol is presented in [17], [18]. Chaarawi and Gabriel proposed an algorithm that selects the number of aggregators automatically based on the file view and process topology [19]. LACIO is developed as a strategy to exploit the logical I/O access pattern among processes and physical layouts of file access to optimize I/O performance [20]. Two phases I/O pipelining that overlaps the communication with file access is proposed in [21], [22]. With the emerging of the solid-state driver (SSD), the cost of file access can be further reduced [23]. Burst buffering is proposed to take advantage of faster SSD devices to improve parallel I/O performance [24], [25]. Although SSD cannot always replace traditional hard disks, hybrid usage of both disks by placing requests with high I/O cost to a small number of SSDs can achieve reasonably good I/O performance [26], [27], [28].

Research has been conducted recently to reduce the timing cost of the communication phase. Tsujita *et al.* proposed a method for overlapping the I/O phase with the communication phase with the help of multi-threading [29]. Cha and Maeng applied a node reordering approach for reducing communication costs with non-exclusive scheduling [30]. MPICH-G2 presents a multi-level topology-aware strategy for MPI collective communication that can improve communication by considering reducing both intra-node and inter-node traffic [31]. TAPIOCA proposes a topology-aware two-phase I/O algorithm that takes advantage of double-buffering and one-sided communication to reduce the process idle time during data aggregation [32], [33]. HierKNEM, a kernel-assisted topology-aware collective framework, improves communication performance on multi-core systems by using multiple layers of collective algorithms [34]. Chakraborty *et al.* further improves the kernel-assisted collective techniques by reducing communication contention [35]. Optimizations that consider the communication topology have demonstrated their potentials for enhancing the two-phase I/O performance.

The request rearrangement in the two-phase I/O exhibits an all-to-many communication pattern, where for collective write operations, all the MPI processes send their requests to the I/O aggregators. Our recent study for the I/O performance of E3SM [5] model has shown that the communication phase dominates the overall performance for writing cubed sphere variables whose I/O pattern consists of a long list of small and non-contiguous requests on every MPI

TABLE 1
This Table Summarizes the Terms Used in This Article

Term	Description
p	number of processes in collective I/O
q	number of processes per node in collective I/O
c	number of local aggregators per node
k	average number of I/O requests per process
L	set of local aggregators
G	set of global aggregators
p_l	number of local aggregators
p_g	number of global aggregators
v	I/O amount in bytes

process. This communication pattern can cause network contention as the number of processes scales.

3 Design of Two-Layer Aggregation Method

Our proposed new method consists of three steps for a collective I/O operation: intra-node aggregation, inter-node aggregation, and I/O phase. Focusing on communication among processes running on the same node, the intra-node aggregation gathers all requests into a subset of processes, denoted as local aggregators. During this step, there is no communication taking place across compute nodes. Once the requests are received, the local aggregators coalesce the requests into a potentially smaller number but larger contiguous requests. During the inter-node aggregation step, local aggregators send the coalesced requests to the global aggregators based on the assigned file domains. The I/O phase remains the same as the original two-phase I/O: The global aggregators fulfill the gathered I/O requests with the file system. Thus, the proposed techniques are independent of underlying file systems.

Both collective read and write distribute metadata of I/O requests from all processes to local aggregators, followed by local aggregators to global aggregators, and global aggregators to file systems. Data in collective write has the same flow as its metadata of I/O requests. Data in collective read, on the other, flows from file systems to global aggregators, followed by global aggregators to local aggregators, and local aggregators to all processes. Consequently, the data communication pattern of collective read is the reverse of the collective write data exchange pattern.

Let p be the number of processes that participate in collective I/O operations with process rank IDs ranging from 0 to $p-1$. For processes that are placed on the same node, we also use local rank IDs, the ascending order of process rank IDs on this node, to refer to them. For instance, a compute node with three process rank IDs $\{2, 5, 7\}$ placed on it has local rank IDs $\{0, 1, 2\}$. Let k be the average number of I/O requests per process. Let L be a set of local I/O aggregators, and p_l be the number of local aggregators in total. Let G be the global aggregators selected in the original two-phase I/O, and p_g denote the number of global aggregators. We keep the number of global aggregators the same as the original two-phase I/O because the selection of global aggregators has been optimized for the I/O stage. However, our approach does not limit the choice of global aggregators. For the rest of this paper, we refer to the term communication contention as the additional communication cost when there are a large number of concurrent receive operations at global aggregators. The defined terms are summarized in Table 1.

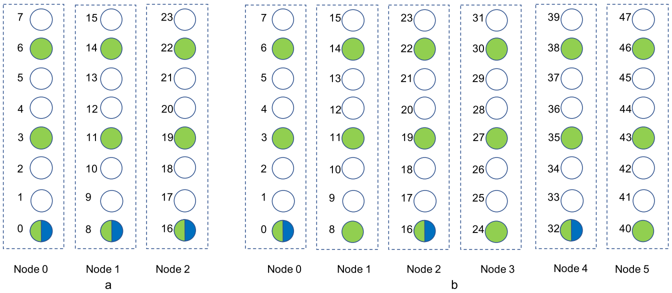


Fig. 1. (a) The placement of 12 local and three global aggregators on three compute nodes while 8 MPI processes are running on each node. This configuration illustrates the case when the number of nodes equals the number of global aggregators. (b) The placement of 16 local and two global aggregators on six nodes in the case the number of nodes is more than the global aggregators. Blue circles represent global aggregators. Green circles represent local aggregators. A global aggregator can also serve as a local aggregator. The placement policy for local and global aggregators is to spread them out evenly among the available resource to prevent possible communication contention.

3.1 Intra-Node Aggregation

The intra-node aggregation selects one or more local MPI processes as local aggregators. Each local process sends its requests to one of the local aggregators on the same node. We partition local processes evenly into p_l groups, so each local aggregator receives from approximately the same number of non-aggregators. A local aggregator gathers non-contiguous I/O requests for processes with rank IDs higher than its rank and smaller than the next local aggregator's rank. Since each non-aggregator sends its request to only one local aggregator, the communication exhibits a many-to-one pattern.

Our design selects local aggregators based on a spread-out principle. The selection of local aggregators depends only on local rank IDs within a node, so the real rank IDs do not have to be contiguous within a node. Formally, let q be the number of MPI processes running on a compute node, c be the number of local aggregators on the node, and $e = (q \bmod c)$. We select processes with local rank IDs $\lceil \frac{q}{c} \rceil \cdot i$ for $i = 0, 1, \ldots, e-1$ and $\lceil \frac{q}{c} \rceil e + \lfloor \frac{q}{c} \rfloor (i - e)$ for $i = e, \ldots, c-1$ as local aggregators.

Sometimes a process rank can serve as both local and global aggregators. Fig. 1 uses two examples to illustrate the selection policy for local and global aggregators: one (Fig. 1a) is when the number of compute nodes is equal to the global aggregators, and the other one (Fig. 1b) is when there are more compute nodes than the global aggregators. For both cases, $c = 3$ out of $q = 8$ MPI processes on each node are local aggregators, so $e = (8 \bmod 3) = 2$. Hence local rank IDs $\{0, 3, 6\}$ for every node are local aggregators. In the former case, there are three global aggregators. The local aggregator with the lowest rank ID per node also serves as a global aggregator. In the latter case, only three out of six nodes contain global aggregators, i.e., on nodes 0, 2, and 4, so rank IDs 0, 16, and 24 are both local and global aggregators.

We implement TAM in ROMIO. The current implementation of ROMIO stores the hostnames of all processes at open time, so local rank IDs that allow us to compute the rank IDs of local aggregators, are available without extra communication cost. The input of MPI collective I/O has an MPI Datatype that defines the file access regions from the file view of the process. At the start of collective I/O, the MPI Datatype is flattened into offset-length pairs that define the non-contiguous regions of file access. For collective write, every process first sends the number of its flattened non-contiguous I/O requests and the size of its write data to its corresponding local aggregator, followed by sending the write data. For collective read, processes send their non-contiguous I/O requests to local aggregators in the same way as the collective write. Data exchange from local aggregators to all processes, on the other hand, happens when the local aggregators have received the read data from global aggregators.

The proposed local aggregator selection policy takes advantage of the possibility that non-contiguous I/O requests from processes of adjacent ranks are likely contiguous, which allows coalescing by the same local aggregator. Furthermore, the input non-contiguous I/O requests to collective I/O are in monotonously non-decreasing order according to the MPI standard, so ROMIO implements two-phase I/O using this fact. If TAM is implemented on the top of existing ROMIO two-phase I/O, the local aggregators must sort all its requests before entering the inter-node and I/O phases of two-phase I/O because local aggregators are the providers of non-contiguous I/O requests from two-phase I/O perspective. Once local aggregators have gathered all non-contiguous I/O requests, they sort non-contiguous I/O requests into a monotonously non-decreasing order based on the file offsets. The offset-length pairs received from a non-aggregator are already sorted in monotonically non-decreasing order themselves, due to the requirement by the MPI-IO standard [1]. We apply the heap merge sort algorithm to merge and sort all non-contiguous I/O requests gathered at each local aggregator. Its time complexity is $O(\frac{p \cdot k}{p_l} \log(\frac{p}{p_l}))$. After sorting all non-contiguous I/O requests, the aggregators coalesce any two consecutive requests that are contiguous.

ROMIO implements the two-phase I/O communication kernel with point-to-point MPI asynchronous functions `MPI_Isend` and `MPI_Irecv` followed by `MPI_Waitall`, so we adopt the same communication functions for the intra-node and inter-node communications for TAM to make a direct comparison with the current ROMIO implementation in the experimental result section. Point-to-point communication functions can avoid the necessity for constructing new sub-communicators by reusing the input communicator. However, collective communication functions, such as `MPI_Alltoallw`, may improve communication performance.

The aggregated data size at local aggregators can exceed the memory limit when the input file size is large. To resolve this problem, we combine the intra-node aggregation phase with the original two-phase I/O that finishes in multiple rounds. However, this implementation strategy sacrifices the advantage of intra-node I/O requests coalescing, resulting in a larger metadata size exchanged in the inter-node aggregation phase later.

3.2 Inter-Node Aggregation

Inter-node aggregation is essentially the communication phase of the original two-phase I/O, but with participation from only the local aggregators as the I/O requesters. In the traditional two-phase I/O, the communication is an all-to-many, between all p processes and p_g global aggregators. Figs. 2a and 2b illustrate the communication complexity of two-phase I/O and TAM, respectively. It is clear to observe the reduction of communication contention on the global aggregators when TAM is used. The two-phase I/O can be considered a special case of TAM when p_l is equal to p. In this case, the intra-node aggregation is skipped.

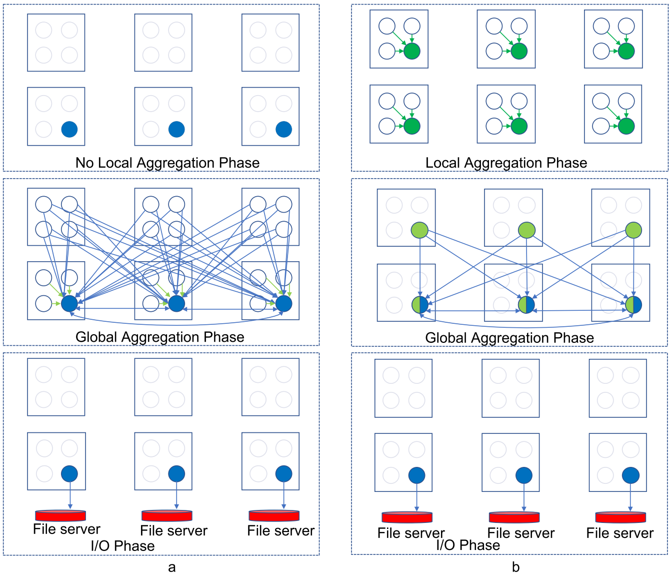


Fig. 2. Illustrations of communication pattern in a collective write operation for two-phase I/O in (a) and TAM in (b). Each square box represents a compute node, and circles are MPI processes. The six green circles are local aggregators, and the three blue ones are global aggregators. Inter-node and intra-node communications are blue and green arrows, respectively. A one-to-one mapping connects the global aggregators and the file servers. In (a), there is no local aggregation phase. The all-to-many (24-to-3) communication pattern shows a potential contention at the global aggregators. In (b), the communication clearly shows a reduced contention (6-to-3) at the global aggregators with the help of local aggregation phase.

If the number of global aggregators is higher than the number of compute nodes, there will be more than one global aggregator placed on the same compute node. If the number of global aggregators is less than the number of compute nodes, then a subset of compute nodes is selected, and one process on each of the selected compute nodes becomes a global aggregator. In this case, some compute nodes do not contain global aggregators. Our scheme spreads out global aggregators across all ranks on a node because the method may utilize hardware resources better when multiple CPUs are on the same node. For example, each Haswell node of the supercomputer Cori has two CPUs with a total of 32 cores. Processes with ranks from 0 to 15 are on the cores from the first CPU, and processes with ranks from 16 to 31 are placed on the cores from the second CPU by default. Spreading out aggregators on the same node maximizes the utilization of both CPUs on the same node. However, TAM can adapt to any existing approach for global aggregator placements. For example, Cray's MPI-IO evenly distributes global aggregators across all nodes. Within a node, it assigns the lowest ranks as global aggregators by default.

At the end of intra-node aggregation, every local aggregator has $\frac{p \cdot k}{p_l}$ number of sorted offset-length pairs on average.

Every local aggregator computes lists of its non-contiguous I/O requests to different global aggregators based on the file domains assigned to them. For collective write, local aggregators send their non-contiguous I/O requests to global aggregators, which presents a many-to-many inter-node communication. For applications with a high volume of non-contiguous access requests, it is frequent that a global aggregator has non-contiguous I/O requests from all processes evenly. Thus, a global aggregator receives $\frac{p \cdot k}{p_l \cdot p_g}$ number of requests from every local aggregator. For collective read, the inter-node data communication pattern is reversed, but the metadata and data contents per communication remain the same. Local aggregators receive data from global aggregators in this stage. Therefore, the inter-node data communication pattern is p_g to p_l for collective read, instead of p_l to p_g in the case of collective write. Since p_g is not necessarily equal to p_l, this asymmetry of communication patterns can cause a difference in collective read and write performance.

For two-phase I/O, each global aggregator receives $\frac{p \cdot k}{p_g}$ number of requests on average from every process during the inter-node aggregation. Sorting requests received from local aggregators with heap merge sort algorithm can improve performance since file requests can coalesce. This approach can significantly reduce the I/O cost by avoiding exchanging a large number of non-contiguous I/O requests with file servers. However, sorting is not necessary if the implementation chooses to apply data sieving, which is enabled by default for collective read in ROMIO. If data sieving is disabled, sorting a p_l number of sorted arrays of requests for the traditional two-phase I/O is $O(\frac{p \cdot k}{p_g} \log p)$. TAM has offset sorting operations at both intra-node aggregation and inter-node aggregation. The total offset sorting complexity of intra-node and inter-node aggregation phases for TAM is $O\left(\frac{p \cdot k}{p_l} \log\left(\frac{p}{p_l}\right) + \frac{p \cdot k}{p_g} \log\left(p_l\right)\right)$. When $p_l \geq p_g$, TAM has a smaller time complexity of sorting than two-phase I/O for collective write. This assumption is valid for file systems that choose p_g values based on available file servers that are much less than the number of compute nodes.

3.3 I/O Phase

In our design, the I/O phase remains the same as the original two-phase I/O implemented in ROMIO. Only global aggregators enter this phase. In addition, two-phase I/O and TAM have the same input and output at the I/O phase.

After inter-node aggregation, a global aggregator has gathered non-contiguous I/O requests within its file domain. When data sieving is enabled, a large chunk of data, with start and end offsets defined by the minimum and maximum byte of the I/O requests gathered at a global aggregator, is read from the file system. For collective read, data buffers for I/O requests are filled using memory copy from this large chunk of data. For collective write, a global aggregator updates the data sieving chunk according to the I/O requests. Later, the data sieving chunk is written back to file servers as a contiguous chunk. If data sieving is not enabled, the read and write for non-contiguous requests from file servers are independent I/O.

Although different file systems store files differently, the improvement of TAM over two-phase I/O is not limited to a specific file system. As long as all global aggregators have to receive high volumes of non-contiguous I/O requests from a large number of processes, TAM can improve the performance since the inter-node communication cost can be reduced with the help of intra-node aggregation.

4 Experimental Results

We conduct our experiments on Theta, a Cray XC40 parallel computer system with Intel KNL processors at Argonne National Laboratory and Cori, a Cray XC40 supercomputer with Intel Haswell processors at the National Energy Research Scientific Computing Center (NERSC). Each KNL node contains one CPU with 64 CPU cores. Each Haswell node contains two CPUs with a lower count of 16 CPU cores on each CPU. Moreover, the bisection bandwidths of the interconnect network of these two supercomputers are different. Running benchmarks on both systems enable us to study whether TAM adapts to different hardware systems and whether the communication contention of two-phase I/O can occur on different hardware architectures. We set the Lustre stripe size to 1 MiB and stripe count to 56, the total number of available OSTs on Theta. All performance results are presented in a strong-scaling evaluation. For every parameter setting, we average timings from five independent executions. Using four I/O benchmarks, E3SM-IO (G and F cases), BTIO, and S3D-IO, we present a performance comparison between TAM and the two-phase I/O implementation in ROMIO.

The ROMIO library from the MPICH release version of 3.3 implements collective read and write differently for Lustre file systems. By default, the implementation of Lustre collective write driver differs from the implementation of collective read driver in ROMIO in terms of the file domain partitioning and collective buffer size. Collective read adopts a static-cyclic file domain partitioning strategy. Each global aggregator has a contiguous file domain that aligns with lock boundaries. File domains of global aggregators for collective write on Lustre file systems, on the other hand, are group-cyclic. With our Lustre setting, each global aggregator owns $\frac{x}{56\text{MiB}}$ number of file domain chunks of size 1MiB for writing a file of size x. Two-phase I/O finishes read and write in multiple rounds. p_g is equal to Lustre stripe size (56) by default for collective write to avoid lock contention. The maximum size of any file domain handled by ROMIO Lustre collective write driver per round is a contiguous Lustre stripe, which is 56MiB in our Lustre setting. For collective read, p_g is equal to the number of compute nodes, and the maximum data size handled per round by each global aggregator is a contiguous chunk of 16MiB by default for all file systems. Thus, the collective read driver reads at most p_g contiguous 16MiB file domains per cycle. The difference in the implementations for collective read and write creates a chance for us to evaluate the impact of p_g and collective buffer size on performance for TAM.

We customize the default ROMIO library into a version that performs at least as well as the Cray MPICH library. This customized version is used as the baseline of our evaluations. TAM is implemented on top of this customized version ROMIO. First, we modify the default MPICH ROMIO to have the same global aggregator placement as Cray MPI for collective write. Using the environmental variable

"MPICH_MPIIO_HINTS_DISPLAY," we observe that Cray MPI selects global aggregators from different compute nodes in a round-robin fashion on Lustre file systems. For example, to select four aggregators from 2 nodes, each running 64 MPI processes with contiguous ranks, Cray MPI picks processes with rank IDs in the order of $\{0, 64, 1, 65\}$. Cray MPI library is not open-source, so we use experiments to analyze its algorithms for collective read and write. According to our analysis on Cray MPI library (compiler craype/2.6.1 and toolkit cray-mpich/7.7.10) with Darshan DXT utility to dump file offsets and lengths information for all MPI processes, Cray MPI library selects the Lustre stripe size of global aggregators by default for collective read and write with the file domain partitioning and collective buffer size identical to the ROMIO collective write implementation. Such implementation of collective read has sub-optimal performance compared with the ROMIO collective read implementation given a large number of processes and nodes because collective read does not suffer from performance degradation of Lustre lock contention [18]. Therefore, we adopt the current ROMIO collective read implementation strategy to carry out the evaluations of two-phase I/O and TAM for collective read evaluation.

We replace all `MPI_Isend` with `MPI_Issend` during the aggregation. This change is critical for a large number of collective I/O requests, where the two-phase I/O must be carried out in multiple rounds. Posting asynchronous send requests using `MPI_Isend` may be cached by the operating system if the message size is small. In this case, at the end of each round of two-phase, even though a call to `MPI_Waitall` is made, processes may continue into the next rounds and post more asynchronous send requests. Therefore, the send requests accumulate in the message queue. A high number of pending asynchronous send requests could seriously hurt the communication performance, due to the possible overwhelming in the message queue processing. Replacing `MPI_Isend` with `MPI_Issend` prevents non-aggregators from continuing into the next round of two-phase I/O, as `MPI_Issend` requires all pending send requests to be received before `MPI_Waitall` returns.

4.1 E3SM-IO Case Study

E3SM [5] is an exascale earth system modeling program for simulating atmosphere, land, and ocean behavior in high resolution. It is an I/O module developed to use PIO library [36], which is built on top of PnetCDF [37]. Writing data at each checkpoint is through posting nonblocking PnetCDF APIs that allow small requests to be aggregated and flushed together. PnetCDF library is a high-level parallel I/O library popular in the climate research community, which is built on top of MPI-IO. Flushing pending nonblocking requests are implemented by aggregating the request data and combining the MPI file views before making a single call to the MPI collective write function. In our experiments, the cost of posting the nonblocking APIs is negligible, and thus, we measure the timing of collective write calls inside the PnetCDF flush API.

We evaluate two data decompositions used in E3SM production runs, namely F and G cases [5]. The I/O kernel of E3SM has been extracted for I/O study [38]. F case has atmosphere, land, and runoff models components. The data access pattern in the F case consists of 1.36 billion non-contiguous write requests and a total write amount of 14 GiB. The G case has active ocean and sea-ice components. G case decomposition file has an MPAS grid data structure that consists of pentagons and hexagons on top of a spherical surface. The I/O pattern of G case contains a shorter list of non-contiguous requests, 180 million in total, and a data size of 85 GiB. The data request file offsets and lengths of individual processes from the production runs are recorded. In both F and G cases, the numbers of non-contiguous requests are different among processes, but the difference is small. To present the performance in a strong-scaling, we assign requests of all processes from the production run evenly among the processes used in our experiments. The assignment is based on the units of processes. In other words, the entire offset-length pairs from a process in the production run is assigned to a process. The non-contiguous I/O requests in E3SM-IO do not have a subarray pattern. Adjacent file offsets do not have a regular interval. Thus, the file access patterns of E3SM-IO differ from BTIO and S3D-IO.

Fig. 3 shows I/O bandwidth comparisons of TAM and two-phase I/O in the strong-scaling evaluation from 256 to 16 K processes on KNL and Haswell nodes. We do not present results beyond 16 K processes since the two-phase I/O performance results show performance slow down towards 16 K processes already. Similar performance degradation is expected as the number of processes scales further. We set 512 local aggregators for experiments on both KNL and Haswell nodes. This setting of p_l for TAM is not necessarily the optimal value for all datasets. Nevertheless, we prove the point that TAM improves the performance of collective I/O without tuning parameters harshly. In general, the results on Cori Haswell have higher bandwidth than the results on Theta KNL. A study of communication and I/O bandwidth on Cori also presents similar results [39]. Haswell node has more sockets and fewer cores (processes) per node, so processes on the same Haswell node share fewer resources compared with KNL nodes. Thus, the contention is less significant on Haswell nodes. In addition, performance degradation of write results is more significant than the performance degradation of collective read results for two-phase I/O on both Haswell and KNL nodes, due to the differences in the implementations of collective read and write mentioned earlier. Nevertheless, two-phase I/O running on both systems drops its bandwidth as the number of processes increases from 4 to 16 K. TAM, on the other hand, does not encounter such performance degradation, though two-phase I/O can outperform TAM for smaller jobs, due to the intra-node aggregation overheads of TAM. From Fig. 3, we observe that TAM maintains a good write/read bandwidth when the number of processes increases.

We use the average number concurrent receive operations at global aggregators for metadata exchanges to illustrate communication contention for all datasets in Fig. 4. This metric is equivalent to the average number of concurrent outgoing and incoming communication requests at global aggregators for collective read and write data exchange. From Figs. 4a and 4b, we can observe that the degree of contention of E3SM F and G cases using two-phase I/O scales

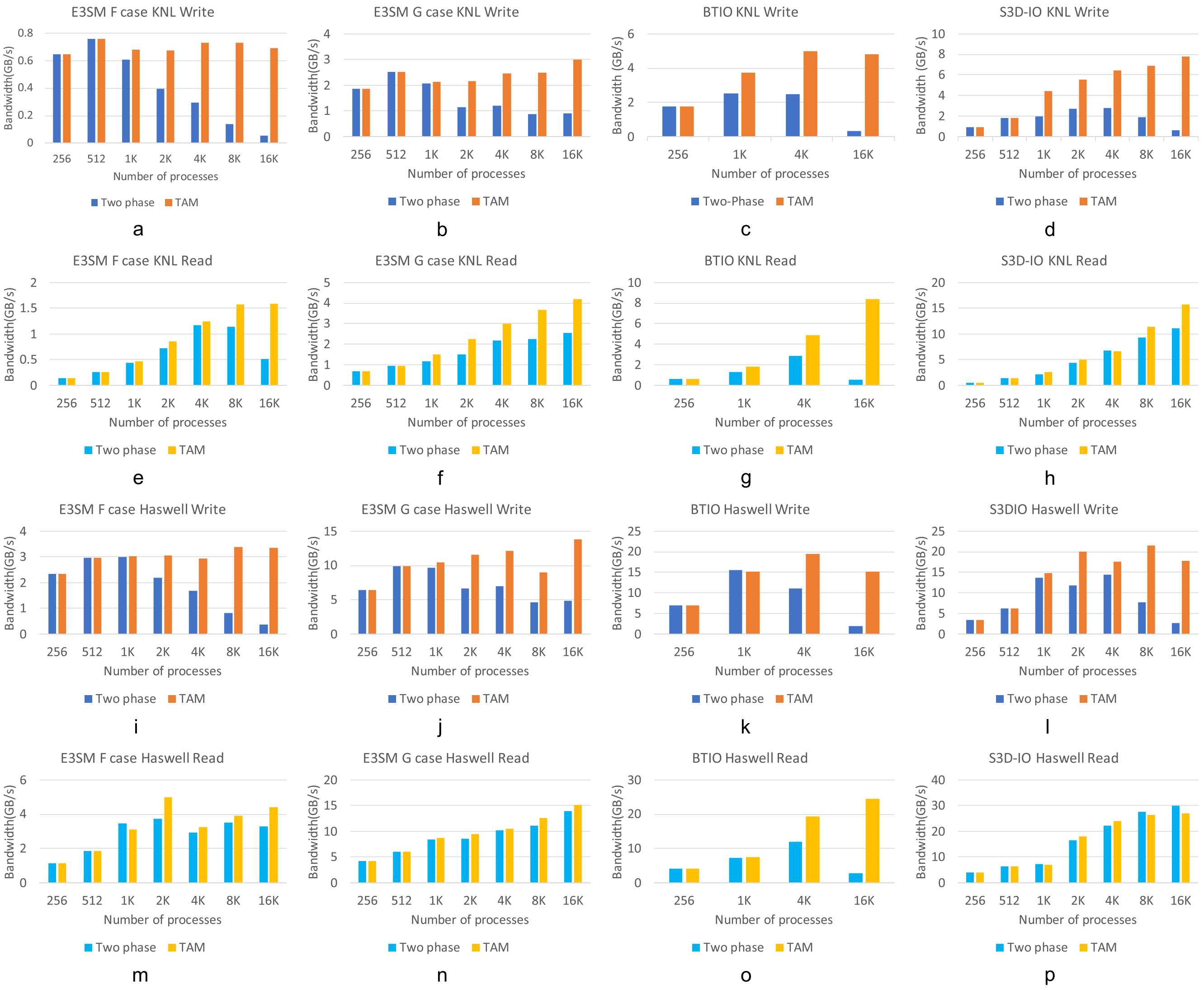


Fig. 3. The I/O bandwidth comparisons of TAM and two-phase I/O in the strong-scaling evaluation from 256 to 16K processes on KNL and Haswell nodes. Haswell nodes have 32 physical cores, and KNL nodes have 64 physical cores. We set the number of processes per node equal to the number of available physical cores. Setting $p_l = p$ automatically degrades TAM to two-phase I/O. For our TAM in these comparisons, we set $p_l = 512$ (or $p_l = p$ if $P < 512$) based on empirical results. For I/O patterns with a large number of non-contiguous I/O requests, this strategy works well. We explain why such p_l yields better performance later.

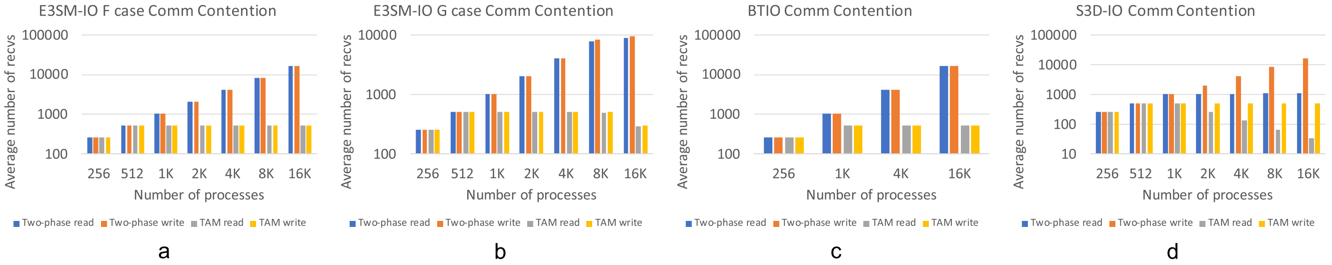


Fig. 4. The degree of contention for all datasets is denoted as the average number of receive operations for metadata exchange at gobal aggregators. TAM has 512 local aggregators in the comparisons.

with the number of processes. TAM, on the other hand, has a smaller degree of contention.

For intra-node aggregation, three components are contributing to the timing. The first component is the communication for gathering metadata to local aggregators. The communication pattern is many-to-one, so the total number of MPI send requests is p to be received by p_l local aggregators. Each of the $\frac{p}{p_l}$ gathering operations within a node can thus run simultaneously. The second component is to merge-sort the request file offsets at every local aggregator. The third component is memory operation for moving the request data into a contiguous space based on the sorted offsets. All three components have timings proportional to request data amount and the number of offsets. Therefore, as we increase the number of

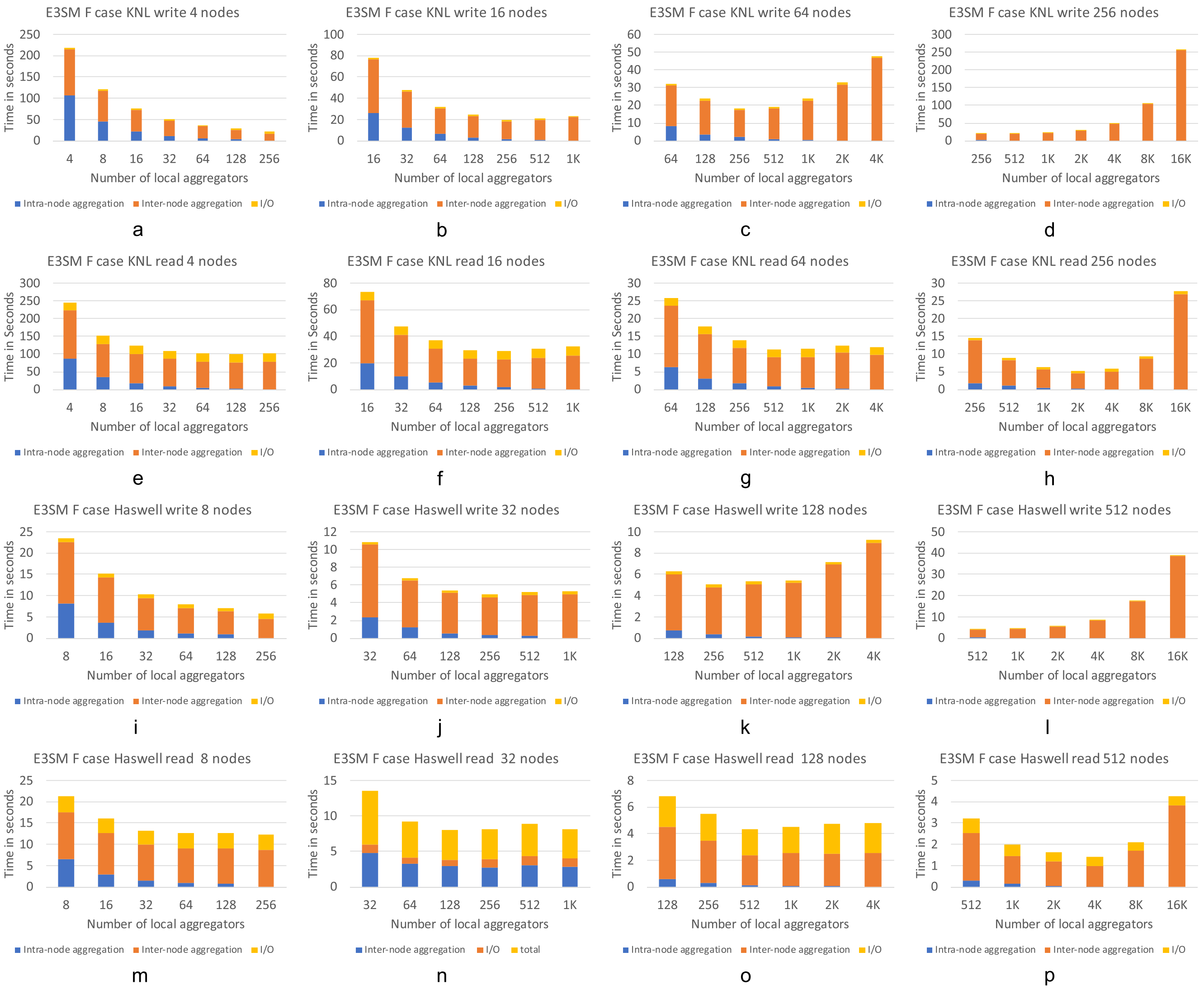


Fig. 5. Timing breakdown for E3SM F case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

local aggregators p_l, the data amount and the number of offsets per local aggregator decrease, so the time of intra-node aggregation decreases proportionally. From Figs. 5 and 6, we observe that the intra-node aggregation time decreases proportionally with increasing number of local aggregators. Consequently, the cost of the intra-node aggregation reduces and becomes negligible when running TAM on a large number of nodes.

In the strong-scaling study, the total read/write amount stays the same regardless of the number of processes. On the Lustre system, the number of global aggregators is fixed for collective write, so the cost of the I/O phase is constant. For collective read, p_g is equal to the number of compute nodes, so I/O phase cost vanishes as the number of compute nodes increases if global aggregators have similar file domain sizes. Thus, the bottleneck of the performance for TAM is the inter-node aggregation phase.

For inter-node aggregation, five components are contributing to its timing. The first two components are flattening the MPI file view into a list of offset-length pairs (calculating my requests) and calculating others' requests to identify the global aggregators who are responsible for writing the request (calculating other requests). In TAM, only local aggregators make calls for calculating my requests. The time complexity is proportional to the number of locally aggregated non-contiguous I/O requests. Calculating other requests involves a many-to-many inter-node communication from local aggregators to global aggregators. Its timing depends on the number of non-contiguous requests and the number of MPI requests, $p_l \cdot p_g$. For collective write, the third component is to merge-sort the offsets of the locally aggregated non-contiguous I/O requests. The fourth component is the construction of MPI derived datatypes at every global aggregator for sending/receiving data from local aggregators. The time complexities of the third and fourth components are both proportional to the number of I/O requests at local aggregators. The last component is the inter-node communication between local aggregators and global aggregators for data. In short, the execution time of inter-node aggregation is a sum of computation cost that depends on the number of I/O requests at local aggregators and the inter-node communication for metadata and data. Based on our experimental observations, inter-node communication time is the dominant factor.

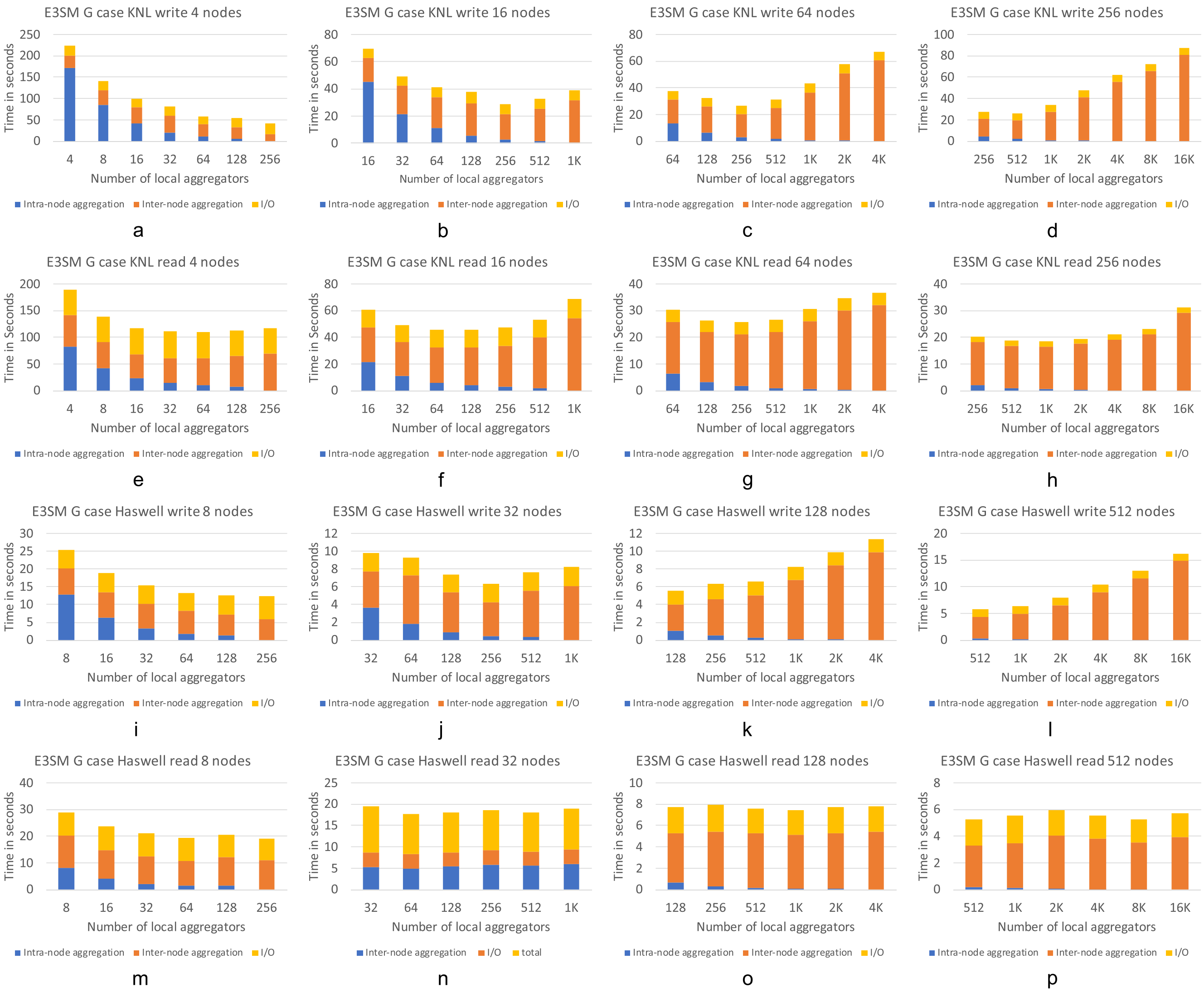


Fig. 6. Timing breakdown for E3SM G case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

Inter-node communication performance depends on whether the communication channels are under or over-utilized. A typical trend is shown in Figs. 5a, 5b, 5c, and 5d, the timing cost of E3SM F case KNL collective write. Initially, on four nodes, the inter-node aggregation cost decreases as the number of local aggregators increases. However, as the total number of nodes increases from 4 to 256, this trend changes. For example, Figs. 5b and 5c illustrate a convex curve of inter-node communication time concerning the number of local aggregators per node. Fig. 5d shows a monotonously increasing trend of inter-node communication as the number of local aggregators per node increases. The inter-node communication has a many-to-many communication pattern between local aggregators and global aggregators. If global aggregators communicate data with a small number of local aggregators, the communication channels are under-utilized at receiving nodes. Thus, the network channels of the nodes where the global aggregators are placed are not saturated up to the case when they communicate with all processes, which explains the observation from Fig. 5a. On the other hand, communication contentions occur if global aggregators exchange data with an enormous number of local aggregators. In Fig. 5d, communication contention starts at assigning more than four local aggregators per node. In the original two-phase I/O, every global aggregator must prepare to communicate metadata and data with all other processes when all-to-many communication occurs. Consequently, as the number of processes increases, the communication contention is expected to become worse at the global aggregators. We should choose a p_l that does not cause over-utilization and under-utilization of communication channels. For example, in Figs. 5b and 5c, $p_l = 256$ is a turning point that yields the lowest inter-node communication cost.

Collective read and write have different turning points and degrees of inter-node contention. For example, from Figs. 5d and 5h, KNL read results for F case has turning point at $p_l = 1\text{K}$. The turning point of the KNL write result, on the other hand, occurs at $p_l = 256$. This difference is expected since collective read and write have reversed inter-node communication patterns. In general, we observe the performance degradation for receiving data from a considerable number of processes is more severe than the contention for sending data to the same number of processes at global aggregators. We set p_l to be a constant 512 by default.

TABLE 2
Datasets Used in Our Evaluation

Dataset	# non-contiguous requests	I/O amount
E3SM G	1.72×10^8 to 1.76×10^8	85GiB
E3SM F	1.35×10^9 to 1.37×10^9	14GiB
BTIO	$40 \times 512^2 \sqrt{p}$	5GiB
S3D-IO	$16 \times 800^2 x$	61GiB

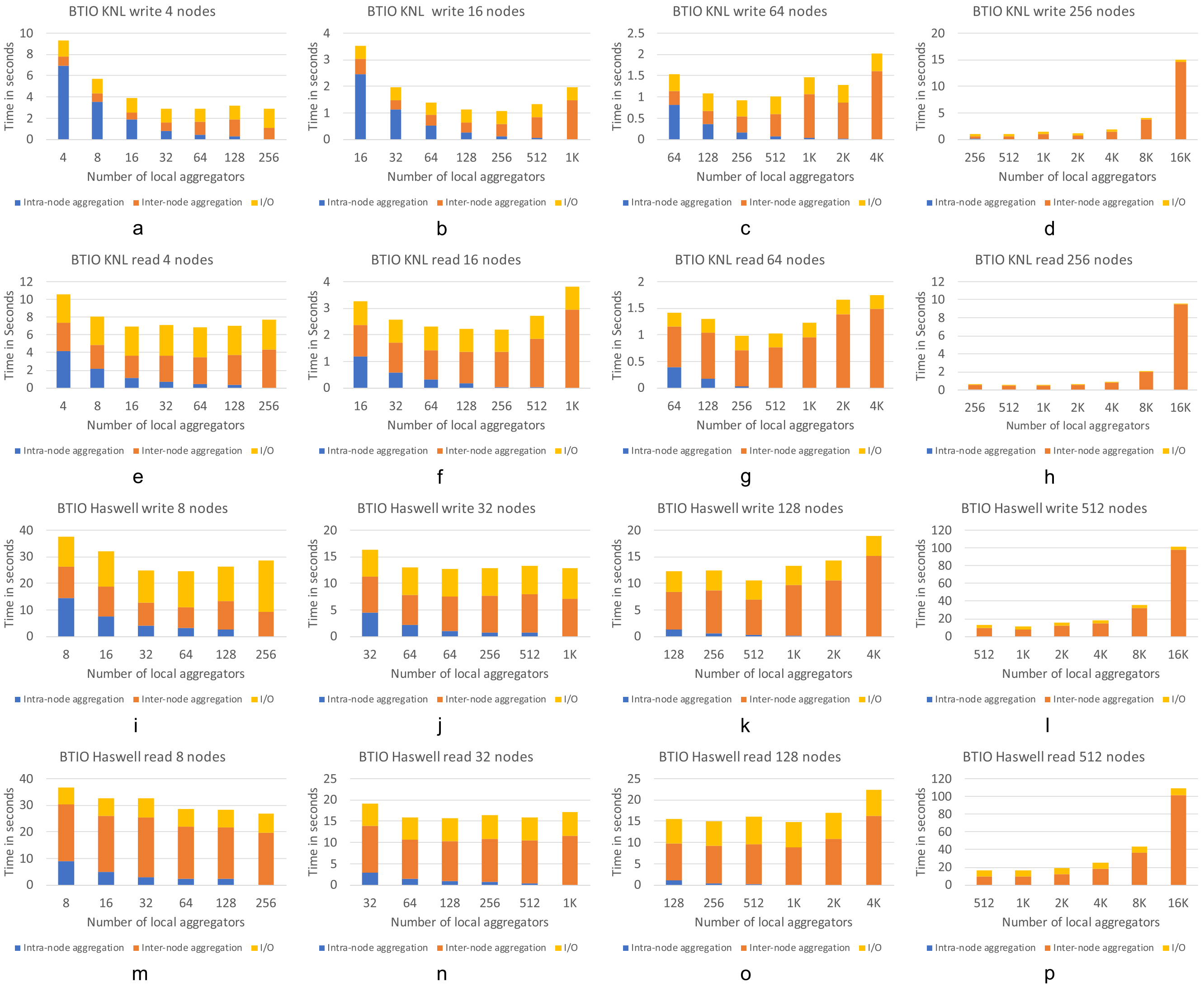
For E3SM F and G benchmarks, the non-contiguous requests are collected from production runs using 21600 and 9600 MPI processes, respectively. As we present the strong-scaling results, the non-contiguous requests are partitioned among all p processes used in our experiments. BTIO and S3D-IO have more significant numbers of non-contiguous requests, and the numbers increase as the number of processes p. For S3D-IO, x, y, and z are the number of processes used to partition X, Y, and Z dimensions. $x \cdot y \cdot z$ is the total number of processes.

However, the optimal choice of p_l depends both on the underlying hardware architecture and the degree of contention illustrated in Fig. 4. Determining the optimal p_l based on communication patterns and hardware systems can be an interesting future study.

4.2 BTIO Benchmark

Developed by NASA Advanced Supercomputing Division, BTIO is part of the benchmark suite NPB-MPI version 2.4 for evaluating the performance of parallel I/O [12]. BTIO uses a block-tridiagonal partitioning I/O pattern over a three-dimensional array. BTIO requires a square number of MPI processes to run, which divide the cross-sections of a global 3D array evenly. We set the global array size to be $512 \times 512 \times 512$. The total write amount is of size $8 \times 512^3 \times 5\text{B} = 5\text{GiB}$ with five 8-byte double values per variable. Unlike E3SM-IO, which has a near-constant size of non-contiguous requests, the total number of non-contiguous requests for BTIO increases along with the number of processes, as shown in Table 2. This property allows us to test TAM for data with a small I/O data size but a large number of I/O requests.

Fig. 7 presents the breakdown timings of BTIO with TAM. With 16K processes, the size of non-contiguous requests is $512 \times 40 \times \sqrt{16384} = 1.34 \times 10^9$. Two-phase I/O completes data exchange in multiple rounds. Each has a file domain size bounded by the Lustre stripe. Although the overall data exchange pattern for BTIO is high, as shown in Fig. 4, not all



a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

Fig. 7. Timing breakdown for BTIO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

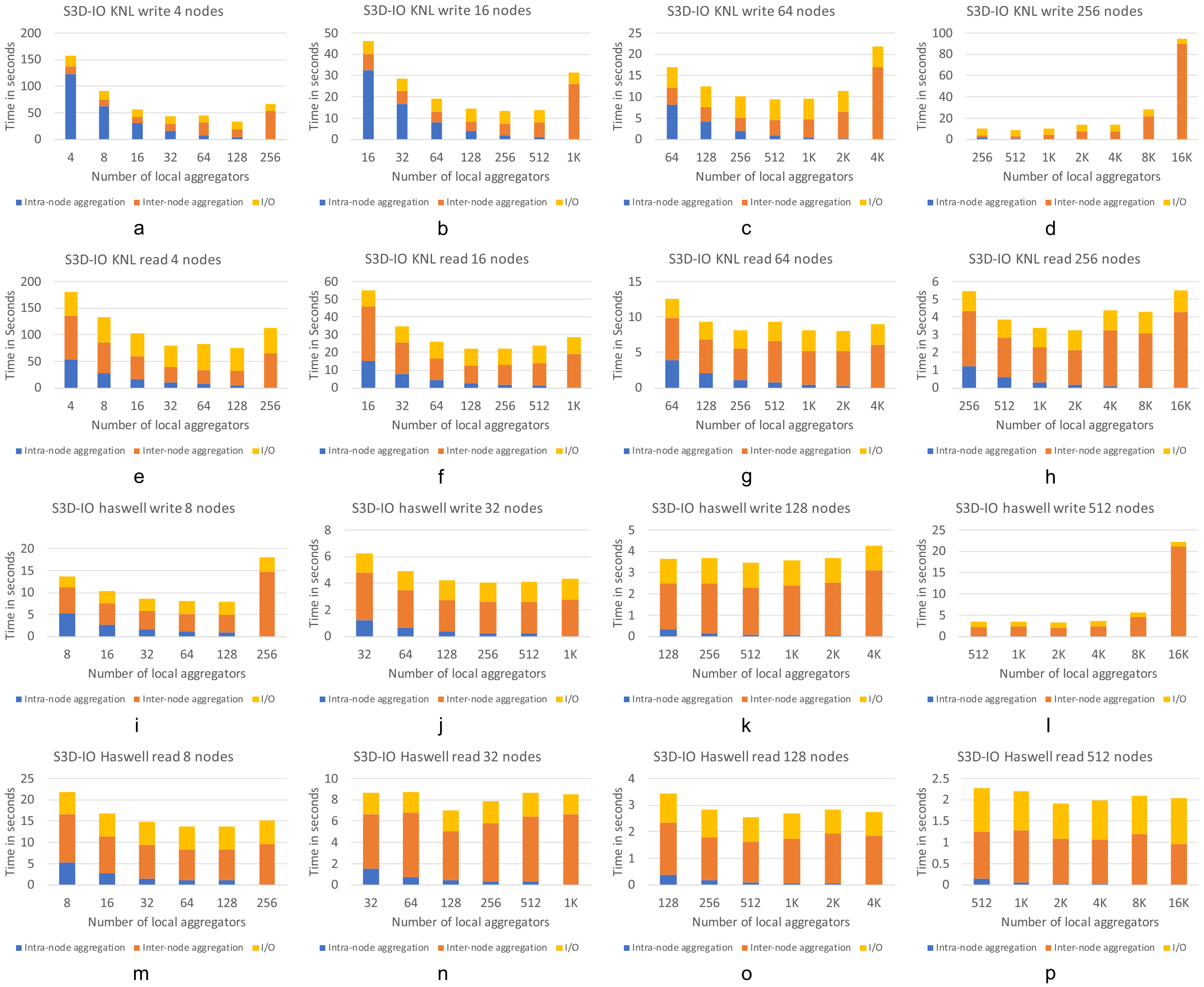


Fig. 8. Timing breakdown for S3D-IO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

processes exchange data with global aggregators per two-phase I/O round. Therefore, data exchange of BTIO in two-phase I/O with a large number of processes does not show the same degree of communication contention compared with the metadata exchange of BTIO. Consequently, the exchange of metadata is the bottleneck of two-phase I/O for BTIO due to a high degree of communication contention, though the number of bytes transferred in the metadata exchange is less than the data exchange. Taking advantage of the spatial locality property of BTIO, TAM reduces the cost of metadata exchange at the inter-node aggregation stage by reducing the number of offset/length pairs transmitted with I/O request coalescing. The number of the coalesced requests is $\left(\frac{1}{2}\right)^{\frac{p}{p_l}}$ of the original request size. Furthermore, TAM can reduce the degree of contention for metadata exchange, since only local aggregators and global aggregators perform inter-node communication.

4.3 S3D-IO Case Study

S3D-IO case study is the I/O kernel of a parallel turbulent combustion application, named S3D, developed at Sandia National Laboratories [11]. The I/O kernel is a checkpoint of three-dimensional arrays, corresponding to a 3D Cartesian mesh. Four variables are written at every checkpoint: mass, velocity, pressure, and temperature. Pressure and temperature are 3D arrays, while mass and velocity are 4D arrays with the fourth dimension sizes 11 and 3. Processes partition the first three dimensions of every variable in a block-block-block fashion. Thus, a process owns 16 non-contiguous 3D subarrays. We set the global 3D array size to $800 \times 800 \times 800$, the same as [18]. This setting results in a total write amount of size $8 \times (11 + 3 + 1 + 1) \times 800^3\text{B} = 61\text{GiB}$ with 8-byte double type per value.

The block-block-block partitioning of the global 3D array is expected to produce non-contiguous requests that mostly can be coalesced at the local aggregators, a similar effect observed in the BTIO benchmark. The number of non-contiguous I/O requests after coalescing at intra-node aggregation phase is $\left(\frac{1}{2}\right)^{\frac{p}{p_l}}$ of the original one. The total number of non-contiguous requests for S3D-IO is 800^2x. x is the number of processes in the first dimension. Nevertheless, inter-node aggregation using two-phase I/O is still a bottleneck of S3D-IO as the number of processes increase.

As shown in Fig. 8, TAM does not show a significant improvement for reading S3D-IO collectively. With our

Lustre setting for collective write in ROMIO, a process has I/O requests that intersect with the file domains of all global aggregators. Therefore, the communication pattern is all processes to all global aggregators when two-phase I/O is used. Collective read, on the other hand, does not have such a communication pattern. File domains of global aggregators in ROMIO for collective read are contiguous blocks that align with lock boundaries. Thus, a process only needs to communicate with a subset of global aggregators with file domains that overlap with the 3D subarrays owned by the process. Hence the communication pattern of two-phase I/O collective read does not cause a severe degree of contention compared with collective write. In Fig. 4d, we can confirm that the degree of contention of two-phase I/O collective read is approximately 1,000, which is ten times less than that of two-phase I/O collective write with 16 K processes. As a result, TAM does not improve the performance of the two-phase I/O collective read in the same way as collective write for S3D-IO.

4.4 Limitation of TAM

The benchmarks and application kernels in our experiments, though differ in detailed I/O pattern, have high numbers of non-contiguous I/O requests. Thus, global aggregators receive data that originated from all processes at the inter-node aggregation phase. However, TAM is designed for I/O patterns with a large volume of non-contiguous requests that cause a high communication contention for global aggregators and all processes. If inter-node communication contention is not severe, the intra-node aggregation phase becomes redundant. In this case, MPI collective I/O implementation can adopt the traditional two-phase I/O strategy because the intra-node aggregation stage is redundant given such a scenario.

5 Conclusion

In this paper, we have demonstrated the communication cost of the two-phase I/O strategy can become the performance bottleneck of the MPI collective I/O. Our proposed TAM is designed to tackle the problem from the angle of reducing communication contention at global aggregators. Adding an intra-node aggregation effectively reduces the communication contention at the global I/O aggregators and thus allows collective I/O to scale up with more MPI processes. Experiments show that TAM works well for applications that make a large number of non-contiguous I/O requests from every process. As the HPC community is entering the exascale era, keeping MPI-IO implementation scalable to higher numbers of MPI processes has become increasingly important.

The design concept of TAM does not limit the first phase aggregation to be within a compute node. We have illustrated that the cost of intra-node aggregation stage in strong-scaling vanishes as the number of processes scales up. When the total number of nodes scales much more massive than our experimental settings, the first phase aggregation across multiple nodes is expected to have a small cost as well. Inter-node aggregation to global aggregators, on the other hand, suffers from communication contention even if there is one local aggregator per node, so the end-to-end time can be lower if we reduce the number of local aggregators by performing the first phase aggregation across a small number of compute nodes.

There are other opportunities to improve the collective I/O performance further. One possibility is to overlap the communication with the I/O as the pipelining approaches proposed in [21], [22]. Moreover, the intra-node aggregation in TAM is motivated by the scenario when the number of MPI processes is significant, resulting in a high number of messages sending to a small amount of I/O aggregators. However, when the number of MPI processes is not significant enough, such as assigning one communication process per node in the MPIOpenMP programming model, TAM is less effective compared with the applications that assign a large number of MPI processes per node. Nevertheless, as the number of nodes scales up in exascale computing, the first phase aggregation across multiple nodes can still improve the communication performance. Future work includes the extension of a TAM to consider MPI processes allocated at compute nodes that are physically near with each other sharing the same communication hardware, such as routers in the identical cabins.

Acknowledgments

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and in part by the DOE Office of Advanced Scientific Computing Research. This work was also supported in part by the DOE award numbers DE-SC0014330 and DE-SC0019358. This research used resources of the National Energy Research Scientific Computing Center, a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, and the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] MPI Forum, MPI: *A Message-Passing Interface Standard. Version 3.1*, Jun. 4th, 2015. [Online]. Available: www.mpi-forum.org

[2] S. Habib *et al.*, “HACC: Simulating sky surveys on state-of-the-art supercomputing architectures,” *New Astron.*, vol. 42, pp. 49–65, Jul. 2015.

[3] R. Latham *et al.*, “A case study for scientific I/O: Improving the FLASH astrophysics code,” *Comput. Sci. Discov.*, vol. 5, Mar. 2012, Art. no. 015001.

[4] J. H. Chen *et al.*, “Terascale direct numerical simulations of turbulent combustion using S3D,” *Comput. Sci. Discov.*, vol. 2, no. 1, 2009, Art. no. 015001.

[5] P. M. Caldwell *et al.*, “The DOE E3SM coupled model version 1: Description and results at high resolution,” *J. Advances Model. Earth Syst.*, vol. 11, pp. 4095–4146, 2019.

[6] K. Schuchardt, B. Palmer, J. Daily, T. Elsethagen, and A. Koontz, “IO strategies and data services for petascale data sets from a global cloud resolving model,” *J. Phys.: Conf. Ser.*, vol. 78, 2007, Art. no. 012089.

[7] J. del Rosario, R. Bordawekar, and A. Choudhary, “Improved parallel I/O via a two-phase run-time access strategy,” *ACM SIGARCH Comput. Archit. News*, vol. 21, pp. 31–38, Dec. 1993.

[8] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk, “Non-data-communication overheads in MPI: Analysis on blue gene/P,” in *Proc. Eur. Parallel Virt. Mach./Message Passing Interface Users’ Group Meet.*, 2008, pp. 13–22.

[9] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, Art. no. 52.

[10] R. Thakur, R. Ross, E. Lusk, and W. Gropp, "Users guide for ROMIO: A high-performance, portable MPI-IO implementation," MCS dividion, Argonne National Lab., Lemont, IL, USA, Tech. Rep. 234, May 2002.

[11] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law, "Direct numerical simulations of turbulent lean premixed combustion," in *J. Phys.: Conf. Ser.*, vol. 46, 2006, Art. no. 38.

[12] P. Wong and R. Der Wijngaart, "NAS parallel benchmarks I/O version 2.4," NASA Ames Research Center, Moffet Field, CA, USA, Tech. Rep. NAS-03–002, 2003.

[13] J. M. Del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *ACM SIGARCH Comput. Archit. News*, vol. 21, no. 5, pp. 31–38, 1993.

[14] Mpich3, 2017. [Online]. Available: http://www.mpich.org/downloads/

[15] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. 11th Eur. PVM/MPI Users' Group Meet.*, 2004, pp. 97–104.

[16] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving MPI-IO output performance with active buffering plus threads," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, p. 10.

[17] W.-K. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proc. ACM/IEEE Conf. Supercomput.*, 2008, Art. no. 3.

[18] W.-K. Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 260–272, Feb. 2011.

[19] M. Chaarawi and E. Gabriel, "Automatically selecting the number of aggregators for collective I/O operations," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2011, pp. 428–437.

[20] Y. Chen, X.-H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp, "LACIO: A new collective I/O strategy for parallel I/O systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 794–804.

[21] S. Sehrish, S. W. Son, W. K. Liao, A. Choudhary, and K. Schuchardt, "Improving collective I/O performance by pipelining request aggregation and file access," in *Proc. 20th Eur. MPI Users' Group Meeting*, 2013, pp. 37–42.

[22] 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 *Proc. Symp. High Perform. Comput.*, 2012, pp. 7:1–7:8.

[23] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 381–392.

[24] K.-Y. Hou *et al.*, "Integration of burst buffer in high-level parallel I/O library for exascale computing era," in *Proc. Workshop Parallel Data Storage Data Intensive Scalable Comput. Syst. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 1–12.

[25] D. Bin, S. Byna, K. Wu, H. Johansen, J. Johnson, and N. Keen, "Data elevator: Low-contention data movement in hierarchical storage system," in *Proc. IEEE 23rd Int. Conf. High Perform. Comput.*, 2016, pp. 152–161.

[26] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A cost-aware region-level data placement scheme for hybrid parallel I/O systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.

[27] S. He, X.-H. Sun, and B. Feng, "S4D-Cache: Smart selective SSD cache for parallel I/O systems," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.

[28] S. He, Y. Wang, X.-H. Sun, C. Huang, and C. Xu, "Heterogeneity-aware collective I/O for parallel I/O systems with hybrid HDD/SSD servers," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 1091–1098, Jun. 2017.

[29] 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 *Proc. 22nd Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2014, pp. 232–235.

[30] K. Cha and S. Maeng, "Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling," *J. Supercomput.*, vol. 61, no. 3, pp. 966–996, 2012.

[31] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proc. 14th Int. Symp. Parallel Distrib. Process.*, 2000, pp. 377–384.

[32] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O library for optimized topology-aware data aggregation on large-scale supercomputers," in *Proc. Int. Conf. Cluster Comput.*, 2017, pp. 70–80.

[33] 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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 19:1–19:11.

[34] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Kernel-assisted and topology-aware MPI collective communications on multicore/many-core platforms," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 1000–1010, 2013.

[35] S. Chakraborty, H. Subramoni, and D. K. Panda, "Contention-aware kernel-assisted MPI collectives for multi-/many-core systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 13–24.

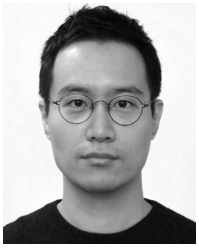
[36] J. Edwards, J. Dennis, M. Vertenstein, and E. Hartnett, "Parallel IO libraries (PIO) - High-level parallel I/O libraries for structured grid applications." Accessed: May 23, 2019. [Online]. Available: https://github.com/NCAR/ParallelIO.

[37] PnetCDF, 2019. [Online]. Available: www.mcs.anl.gov/parallel-netcdf/

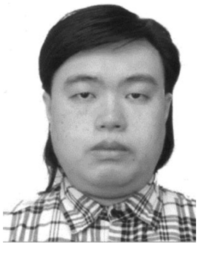
[38] Parallel I/O kernel case study – E3SM. Accessed: Mar. 06, 2019. [Online]. Available: https://github.com/Parallel-NetCDF/E3SM-IO

[39] J. Liu *et al.*, "Understanding the I/O performance gap between Cori KNL and Haswell," Lawrence Berkeley National Lab. (LBNL), Berkeley, CA, USA, 2017.

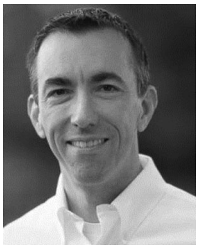
Qiao Kang received the BS (first-class) degree in computer science and statistics from the University of St. Andrews, St Andrews, United Kingdom. He is currently working toward the PhD degree with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois. His research interests lie in high-performance computing and spatiotemporal anomaly detection.



Sunwoo Lee received the bachelor's and master's degrees in computer engineering from Hanyang University, Seoul, South Korea. He is working toward the PhD degree with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois. His research interests include high-performance computing and machine learning.



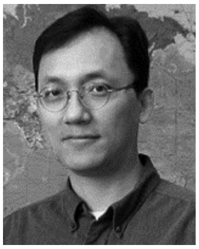
Kaiyuan Hou received the BS and MS degrees in computer science from the National Taiwan University, Taipei, Taiwan, in 2014 and 2016, respectively. He is working toward the PhD degree with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, since September 2016. His research interest include parallel I/O.



Robert Ross received the PhD degree in computer engineering from Clemson University, Clemson, South Carolina. He is a senior computer scientist with Argonne National Laboratory, a senior fellow with the Northwestern-Argonne Institute for Science and Engineering, and the director of the DOE SciDAC RAPIDS Institute for Computer Science and Data. His research interests include system software for high-performance computing systems, distributed storage systems, and libraries for I/O and message passing.

Ankit Agrawal received the PhD degree in computer science from Iowa State University, Ames, Iowa. He is a research associate professor with the Department of Electrical and Computer Engineering, Northwestern University. He specializes in interdisciplinary big data analytics via high-performance data mining, based on a coherent integration of high-performance computing and data mining to develop customized solutions for big data problems.

Alok Choudhary (Fellow, IEEE) received the PhD degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, Champaign, Illinois, in 1989. He is the founder of 4C Insights, a data science and AI software company. His research over the last decades focused on big data science, supercomputing, scalable data mining, machine learning, AI, and their applications in sciences, medicine, and business applications. He is a fellow of the ACM and AAAS.



Wei-keng Liao received the PhD degree in computer and information science from Syracuse University, Syracuse, New York, in 1999. He is a research professor with the Department of Electrical and Computer Engineering, Northwestern University. His research interests include the area of high-performance computing, parallel I/O, data mining, and data management for large-scale scientific applications.