Improving All-to-Many Personalized Communication in Two-Phase I/O

Qiao Kang*, Robert Ross§, Robert Latham§, Sunwoo Lee*, Ankit Agrawal*, Alok Choudhary*, and Wei-keng Liao*

* *Department of Electrical and Computer Engineering, Northwestern Univeristy*
2195 Sheridan Rd., Evanston, IL 60208, USA
{qiao.kang, slz839, wkliao}@ece.northwestern.edu

§ *Mathematics and Computer Science Division, Argonne National Laboratory*
9700 S. Cass Ave., Lemont, IL 60439, USA
{rross,robl@mcs.anl.gov}@mcs.anl.gov

***Abstract*—As modern parallel computers enter the exascale era, the communication cost for redistributing requests becomes a significant bottleneck in MPI-IO routines. The communication kernel for request redistribution, which has an all-to-many personalized communication pattern for application programs with a large number of noncontiguous requests, plays an essential role in the overall performance. This paper explores the available communication kernels for two-phase I/O communication. We generalize the spread-out algorithm to adapt to the all-to-many communication pattern of two-phase I/O by reducing the communication straggler effect. Communication throttling methods that reduce communication contention for asynchronous MPI implementation are adopted to improve communication performance further. Experimental results are presented using different communication kernels running on Cray XC40 Cori and IBM AC922 Summit supercomputers with different I/O patterns. Our study shows that adjusting communication kernel algorithms for different I/O patterns can improve the end-to-end performance up to 10 times compared with default MPI-IO implementations.**

***Index Terms*—MPI-IO, ROMIO, two-phase I/O, communication traffic throttling**

I. Introduction

The Message Passing Interface (MPI) standard defines a set of programming interfaces for interprocess communication [1]. MPI-IO, a submodule of the MPI standard, provides interfaces for parallel shared-file access. I/O operations in scientific applications such as as [2]–[6] are implemented with MPI-IO libraries. MPI processes can collectively open a file to perform I/O operations. Implementations of the collective I/O functions can coordinate processes' operations to achieve better end-to-end performance compared with independent I/O. A well-known collective I/O design 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 consists of a communication phase and an I/O phase. I/O aggregators, a subset of processes, are assigned with file access regions, denoted as file domains. The aggregators gather I/O requests and data, based on their file domain, from the rest of the processes in the communication phase and carry out I/O operations with file systems in the I/O phase. In general, the communication between nonaggregators and I/O aggregators is many-to-many. ROMIO, the implementation of the MPI-IO functions used most frequently in high-performance computing (HPC) and provided by vendors as part of their MPI implementation [8], implements two-phase I/O in multiple rounds of communication and I/O with asynchronous MPI functions. When the number of noncontiguous I/O requests is significant, the communication phase of two-phase I/O exhibits an all-to-many personalized communication pattern, where the "many" group refers to the I/O aggregators. The communication cost may exceed the I/O cost for large parallel jobs when the number of noncontiguous I/O requests is significant because of high communication contention and straggler effects [9], [10] caused by multiple rounds of all-to-many communication, especially as the number of processes increases.

In this paper, we explore all-to-many personalized communication algorithms: pairwise, spread-out, and the two-layer aggregator method (TAM) for improving two-phase I/O communication performance. We extend the spread-out algorithm adopted by the implementations of personalized all-to-all in major MPI production libraries to adapt to the all-to-many communication pattern by reducing the straggler effect resulting from an unbalanced communication workload. Throttling techniques are additionally applied to the asynchronous MPI point-to-point implementation to reduce communication contention. For evaluation, we replace the metadata and communication kernels of two-phase I/O in ROMIO with different all-to-many personalized communication algorithms. Experiments are conducted on two different supercomputing systems: Cori, a Cray XC40 supercomputer with Intel KNL processors and the Lustre file system, and Summit

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

IBM Power System AC922 nodes equipped with IBM POWER9 CPUs and IBM GPFS. We use three benchmark programs: E3SM-IO [5], FLASH/IO [11], and BTIO [12]. These three I/O benchmarks have different metadata and data communication patterns.

Applying communication throttling in a two-phase I/O communication phase with an all-to-many pattern can improve the communication performance by up to 10 times, compared with the current implementation of ROMIO two-phase I/O. Our balancing strategy for the spread-out algorithm improves communication performance by over 30%. The two-layer aggregation method [13], an implicit throttling approach that performs intranode and internode communications separately, works best for I/O patterns that require all-to-many communication because internode communication contentions are significantly reduced. However, we expect that applying the proposed spread-out reordering method, along with the throttling technique, to the communication kernels of existing communication designs such as TAM can further reduce internode communication contentions for the future exascale applications running on a considerable number of compute nodes. When applications do not exhibit significant many-to-many communication contentions, the current ROMIO implementation for all-to-many communications can outperform the rest of the communication kernels. Thus, the communication kernel of two-phase I/O should be dynamically chosen based on application I/O patterns.

II. Background

We define the terminology of personalized communication. Let P be the array of all processes ranked from 0 to $p-1$. A sorted array of sender ranks S transfer their data with arbitrary length to a sorted array of receiver ranks R. Let $m_{i,j}$ be the message sent from rank i to rank j. $m_{i,j}$ is empty if either $i \notin S$ or $j \notin R$. We refer to $m_{0,j}, ..., m_{p-1,j}$ as the local messages at rank j. We assume that point-to-point communication operations (send/receive) between any two arbitrary processes are available. An algorithm schedules the send and receive operations to accomplish the data transfer. All-to-many communication has $S = P$. All processes are sending personalized messages to a subset of processes.

Our point-to-point communication model has two components: the startup term t_s and the per-byte transfer (inverse bandwidth) time t_w. The startup and transfer terms are classical communication models, as mentioned in [14]. Sending k bytes from a sender to a receiver, including self-send, has a communication cost of $t_s + kt_w$. For simplicity of analysis, we assume a textbook one-port communication constraint, which means the maximum number of either concurrent send or receive operations is 1. We can, however, extend our results to an n-port communication constraint by discussing send and receive operations in blocks of processes. We denote the term communication contention as the additional time cost resulting from excess concurrent point-to-point communication requests at a process.

A. Motivation

Our research is motivated by the communication pattern of two-phase I/O [7]. Two-phase I/O consists of communication and I/O phases. A few MPI processes called I/O aggregators are selected as I/O proxies for all processes. Only I/O aggregators exchange data with the file systems. Other processes communicate with I/O aggregators to complete their I/O requests. For collective write, all processes send their I/O requests and write data to I/O aggregators. I/O aggregators then exchange data with file servers according to the gathered I/O requests. Parallel I/O for large files in the exascale computing era can cause memory overflow at I/O aggregators, especially when the number of I/O aggregators is much smaller than the number of processes in the computation. ROMIO [8], a widely used MPI-IO implementation adopted by major production libraries, avoids the memory overflow by implementing two-phase I/O in multiple rounds. The I/O driver processes file domains with limited sizes in each round. Thus, the communication of two-phase I/O consists of multiple rounds of many-to-many communication. With a large number of noncontiguous I/O requests, the communication pattern becomes all-to-many. Our recent study of the I/O performance of the E3SM [5] model has shown that such all-to-many communication dominates the overall performance. The performance degradation is more severe as the number of processes scales.

B. Related Work

Personalized communication is implemented by the `MPI_Alltoallv` and `MPI_Alltoallw` functions. Many works of literature have focused on all-to-all communication patterns ($S = R = P$). We summarize the mainstream algorithms that are implemented by production libraries such as MPICH [15] and OpenMPI [16].

Bruck's algorithm [17] is an efficient algorithm for transmitting small messages in personalized communication. It has a performance advantage over the traditional recursive-doubling approach in practice [18]. This algorithm finishes in $\log(p)$ steps for p that is a power of two. If p is not a power of two, some slight overhead occurs'. At step i, rank x sends nonlocal messages gathered from previous rounds to $x + 2^{i-1} \mod p$. When $S = R = P$ and all receivers have the same local data size k, the overall performance is $\log(p)\, t_s + \frac{k}{2}\left(\log(p) + 2^{\log(p)}\right) t_w$ This performance is desirable when k is small.

For parallel I/O problems, k is usually a large value, so the communication kernel should adopt algorithms that optimize the t_w term. This class of algorithms is designed based on the principle that processes are always busy receiving their personalized messages. The pairwise algorithm is designed for large data exchange when p is a power of two. The algorithm finishes in $p-1$ steps. At step i, rank j exchanges data with rank $i \oplus j$. Another algorithm, called the spread-out algorithm, can also handle the case when p is not a power of two; rank j exchanges data with rank $j-i+p \mod p$ at step i. When $S = R = P$ and all receivers have the same local data size k, both the pairwise and spread-out algorithms have communication cost of $(p-1)t_s + kt_w$. These two methods have advantages over Bruck's algorithm for a large k.

Two-phase transmission for all-to-all exchange is an efficient strategy for reducing the total number of internode communications. Träff and Rougier have proposed a local node gathering strategy for `MPI_Alltoall` data using communicator splitting [19]. SLOAV [20] is an improvement over the Bruck's algorithm with two-phase message transmission that handles data with variable size for `MPI_Alltoallv` and `MPI_Alltoallw`. The two-layer aggregation method (TAM) applies the intranode gathering principle to MPI two-phase communication [13]. A subset of processes, denoted as local aggregators, gathers I/O requests from all processes. then, local aggregators and I/O aggregators carry out two-phase I/O. With the help of intranode aggregation, internode communication contentions are significantly reduced because a smaller number of processes participate in data exchange among compute nodes.

C. MPI Asynchronous Communication

Modern supercomputers have multiple intranode and internode communication channels. Applications can overlap computation and communication for better performance. Therefore, implementing personalized communication patterns with asynchronous MPI functions may utilize hardware resources more efficiently. To implement algorithms for all-to-all with asynchronous MPI communication functions, the MPICH library posts all `MPI_Isend` and `MPI_Irecv` requests in the order defined by the algorithms. A subsequent `MPI_Waitall` for all requests is made in the end.

There are three major MPI asynchronous communication implementations. One method is a thread-based approach, which is adopted by the current branches of MPICH and MVAPICH [21]. `MPI_Init` creates a communication thread for every process that continuously handles point-to-point communication requests posted from the master thread. When a large number of asynchronous send requests are posted to a receiver, however, the mutex-based approach that most MPI implementations adopt today for thread safety can be a source of contention [22]. Another method is to assign dedicated "ghost" processes for processing asynchronously communication requests offloaded from user processes [23]. The "ghost" process approach offers a more flexible core usage compared with the thread-based method, but multiple processes can still race for shared resources. The third approach is to utilize hardware interrupts as in the Blue Gene/Q [24]–[26], and Cray systems.

Algorithm 1: The spread-out algorithm.

```
1  rank ← local process rank ID
2  if $rank \in R$ then
3      for $i \in [0, .., p-1]$ do
4          src ← $(\text{rank} + i) \mod p$
5          rank receives $m_{\text{src,rank}}$ from src
6      end
7  end
```

Algorithm 2: The balanced spread-out algorithm.

```
1  rank ← local process rank ID
2  if $rank \in R$ then
3      $j$ ← local receiver index in $R$
4      rank_list ← Any ordering of process ranks
5      $q \leftarrow p \mod |R|$
6      if $j < q$ then
7          $r_{index} \leftarrow \lceil \frac{p}{|R|} \rceil j$
8      else
9          $r_{index} \leftarrow \lceil \frac{p}{|R|} \rceil q + (j-q) \lfloor \frac{p}{|R|} \rfloor$
10     end
11     for $i \in [0, .., p-1]$ do
12         src ← rank_list[$(r_{index} + i) \mod p$]
13         rank receives $m_{\text{src,rank}}$ from src
14     end
15 end
```

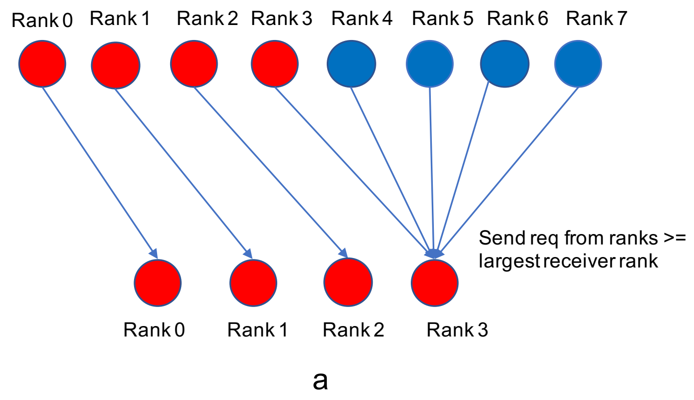
III. Design

In this section, we propose an algorithm for all-to-many personalized communication that evenly spreads out the communication workload. The new algorithm is a generalization of the spread-out algorithm used for `MPI_Alltoallv` and `MPI_Alltoallw` in MPICH. We refer to this improvement as the balanced spread-out algorithm. We compare the spread-out and the balanced spread-out algorithm from the point of view of communication straggler effects. A throttling technique for both algorithms is also discussed for reducing communication contention.

A. The Balanced Spread-Out Algorithm

Algorithm 1 is the traditional spread-out algorithm adopted by major production libraries. The algorithm depicts the receiving order of local messages at an arbitrary receiver. Since R is not necessarily equal to P, non-receivers do not enter the condition at Line 2. When the communication pattern has a few receivers with lower ranks that receive from all processes, non-receivers can cause a potential contention at the beginning, as illustrated in Figure 1 (a). All processes with ranks larger than the largest receiver rank $r_{\max} \in R : r \leq r_{\max} \forall r \in R$ among receivers directly send their messages to $r_{\max}$. For large $p - r_{\max}$, contention occurs due to the straggler effect, which is explained later.

Algorithm 2 resolves the contention scenario of Algorithm 1 by providing a load-balancing solution. To generalize the solution, we construct a rank_list owned by all processes. The array rank_list can be a simple ordering of



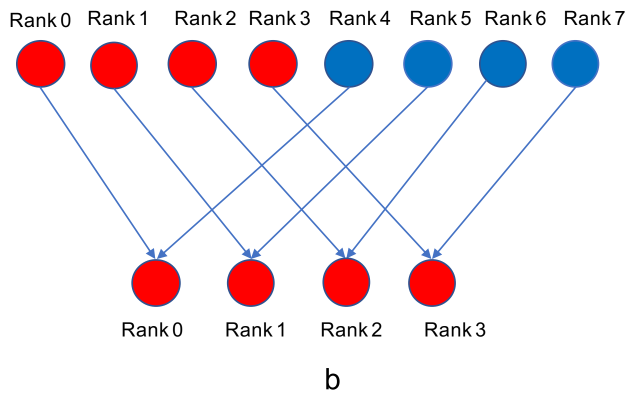


Figure 1: This figure illustrates the communication workload of all processes in the spread-out and balanced spread-out algorithms in the first round. The red circles represent processes that are receivers. The blue circles represent the rest of the processes. An arrow between two circles represent a point-to-point communication. (a) The communication workload is imbalanced in the spread-out algorithm. Rank 3 receives data from 3–7 in the first round. (b) The communication workload is balanced in the balanced spread-out algorithm with virtual rank list $\{0, 4, 1, 5, 2, 6, 3, 7\}$.

processes from 0 to $p-1$. It can also be any ordering of P, such as round-robin, based on compute nodes to adapt to different types of rank assignments. The algorithm divides p processes in the rank_list into $|R|$ groups. If $|R|$ does not divide p, $p \mod |R|$ groups have size of $\lceil \frac{p}{|R|} \rceil$, and the rest of the groups have size $\lfloor \frac{p}{|R|} \rfloor$. Receivers and the process groups form a bijection mapping. The first process in the group that corresponds to an arbitrary receiver is rank_list$[r_{index}]$. The receiver posts its requests from senders in order of rank_list$[r_{index}]$,.., rank_list$[r_{index} + p \mod p]$. Figure 1 (b) is an example of the initial communication step of the proposed algorithm. We use a rank_list setting $\{0, 4, 1, 5, 2, 6, 3, 7\}$ as an example. A receiver posts receive operations from two senders at a time.

B. Straggler Effect in Communication

A communication straggler refers to the communication time difference between a process and the average. Straggler effects are the idle time that a process waits for other processes to finish because of dependency on available data or operation orders. The original spread-out algorithm suffers from straggler effects when applied to all-to-many case, as shown in Figure 1 (a). With a one-port communication constraint, process rank 2 starts to receive data from process rank 4 after process rank 4 has finished the sending operation to rank 3. In general, any receive operation at ranks 0–2 has to wait for all the rest of the senders to finish their previous communications.

We assume that the latency of a point-to-point data transfer has a white noise $\epsilon \sim \mathcal{N}\left(0, \sigma^2\right)$., where σ^2 is the variance of the noise term. Let μ be the mean communication time. For small data size, the ratio μ to σ is large. The message size exchanged per round does not exceed the collective buffer size, so the noise term is not negligible even for very large files. Straggler effects can be formulated as the following. When a receiver attempts to receive from a sender, there are two cases. The first case is that the sender is ready to send, and the second case is that the sender has not finished its previous send operations. When a process finishes receiving its first message and becomes ready to receive its second message, both cases can occur with an equal chance. The first case does not cause delay, but the second case causes a delay in the receive operation at the receiver. The expectation of the straggler effect X_1 after the first step is shown Equation 2. We use ϕ and Φ to denote the probability and cumulative functions of standard normal distribution.

$$E\left(X_1\right) = \frac{1}{2} \times 0 + \int_0^\infty x\phi\left(x\right) dx \tag{1}$$

$$= \sigma\sqrt{\frac{1}{2\pi}} \tag{2}$$

For the spread-out example, rank 0 has straggler effects, denoted by random variable X_i for each of sender ranks $i \in \{2, ..., p-1\}$. Straggler effects from senders with higher ranks are dependent on those from lower ranks. Nevertheless, by treating them independently, we can approximate the expectation of total delay in Equation 4.

$$E\left(\sum_{i=1}^{p-1} X_i\right) \approx \sum_{i=1}^{p-1} \sigma\sqrt{\frac{1}{2\pi}} \tag{3}$$

$$= \left(p-1\right)\sigma\sqrt{\frac{1}{2\pi}} \tag{4}$$

The total delay scales as the number of processes increases. On the other hand, for Algorithm 2, the expectation of total straggler effects at any receivers is close to zero as long as $|R| \leq \frac{p}{2}$. For example, in Figure 1, receivers send to themselves at the beginning. Rank 0 starts to receive from rank 4 after it finishes its self-send. It is unlikely that rank 1 has not finished its self-send by the time when rank 0 finishes receiving data from rank 4. We use X_1' to denote the random variable for the straggler effect from the second communication in Algorithm 2. If we assume

$\sigma = \frac{1}{8}\mu$, a reasonably large variance, the probability of the first straggler effect happening $(X'_1 > 0)$ when $|R| = \frac{p}{2}$ is Equation 5 for Algorithm 2, which is much less than $\frac{1}{2}$ for Algorithm 1. In addition, regardless of the value of σ, this probability is always less than $\frac{1}{2}$.

$$1 - \Phi\left(\frac{2-1}{\sqrt{\frac{1+2}{8^2}}}\right) = 1 - \Phi\left(\frac{8}{\sqrt{3}}\right) \tag{5}$$

$$< 2 \times 10^{-6} \tag{6}$$

Moreover, this probability reduces as $|R|$ or σ decrease. Therefore, the total straggler effects can be considered as negligible. When $p > |R| > \frac{p}{2}$, $p \mod |R|$ out of $|R|$ receivers have the same straggler effects as the spread-out algorithm. When $|R| = p$, the proposed algorithm has identical straggler effects as the spread-out algorithm. To sum up, the balanced spread-out algorithm is expected to have less performance degradation caused by straggler effects compared with the spread-out algorithm.

IV. Implementation

The pairwise algorithm is a store-and-forward design, so adjacent steps have data dependency. Hence this algorithm is usually implemented with MPI blocking functions. Spread-out algorithms, on the other hand, can be implemented with asynchronous MPI functions because there is no data dependency for the personalized communication pattern. Since our proposed algorithm is a generalization of the spread-out algorithm, it can also be implemented by using MPI asynchronous functions. We present two implementation methods that can improve the performance of the spread-out and balanced spread-out algorithms.

A. *Replacing Isend with Issend*

Both `MPI_Isend` and `MPI_Issend` can be used for posting data transfer requests from senders to receivers. The difference between these two, according to the standard, is the behavior when `MPI_Wait` is called for their requests.

`MPI_Wait` may return the request from `MPI_Isend` when the send buffer is ready to be modified. The request from `MPI_Issend` cannot be returned from `MPI_Wait` until the `MPI_Irecv` at the remote side has started receiving data from the corresponding `MPI_Issend`. For a single point-to-point communication, the end-to-end latency difference between using MPI_Isend and MPI_Issend is not significant. The difference is critical for the performance of all-to-many communication, however, especially when there are multiple rounds.

For small messages, some MPI implementations cache the data of `MPI_Isend` requests. `MPI_Wait` immediately returns after the message copy from the send buffer. If the all-to-many operation is executed many times in a loop with small message sizes, senders can accumulate a large number of data buffers for `MPI_Isend` requests because the program is not necessarily blocked by `MPI_Waitall` after data memory copy. Receivers, on the other hand, can escape from the `MPI_Waitall` blocking only after receiving all data in the `MPI_Irecv` requests in the current iteration. When a large number of requests accumulate at the senders, the communication performance at senders degrades dramatically, resulting in poor performance. We refer to this scenario as request contention at senders.

To avoid request contention at senders, we could call `MPI_Barrier` at the end of each iteration. The barrier would force synchronization of senders and receivers, so senders could not post requests in the next iteration until all requests in the current iteration were finished. Performing a global barrier per round, however, can significantly degrade performance. Alternatively, replacing `MPI_Isend` with `MPI_Issend` can resolve this issue. `MPI_Issend` forces receivers to acknowledge requests by calling `MPI_Irecv` at the remote side. This approach allows us to force senders to block until receivers have at least entered the sender's current round. We adopt this `MPI_Issend` strategy, avoiding the need to use a barrier to synchronize all processes and the associated delays.

B. *Throttling for Asynchronous Communication*

Oversubscription caused by a large number of MPI requests can be a performance bottleneck [9]. Asynchronous implementation of all-to-many algorithms often requires receivers to handle multiple `MPI_Irecv` requests simultaneously. When `MPI_Waitall` is called, receivers test whether data for individual `MPI_Irecv` have been received. The checking operations can be considered as a lightweight version of `MPI_Test`. A long list of requests can result in longer checking time. Furthermore, the orders of acknowledgment for senders and messages received for receivers are not constrained. Hence the worst-case checking complexity is proportional to the number of MPI requests in the queue. In addition, when a large number of MPI requests are passed to lower-level communication libraries, repeated checking can also happen at those libraries. Therefore, posting an extensive array of MPI requests is not recommended.

Throttling at a collective communication algorithm level is a useful technique to avoid performance degradation caused by oversubscription. The current `MPI_Alltoallv` and `MPI_Alltoallw` implementation of Algorithm 1 by MPICH has adopted this strategy. The throttling strategy separates `MPI_Irecv` requests into small groups with a maximum size of comm_size. For example, the MPICH master branch has a default size of 32, and Cray-mpich has comm_size (MPICH_ALLTOALLV_THROTTLE) equal to 8 by default. `MPI_Waitall` is called for every group of `MPI_Irecv` requests sequentially at receivers. The total number of concurrent communication is bounded by comm_size. Similarly, TAM [13] implicitly throttles for internode communication since only local and I/O aggregators perform internode communication.

Table I: Datasets used in our evaluation. The second column shows the total number of noncontiguous requests. For the E3SM F benchmark, the noncontiguous requests are collected from production runs using 21,600 processes. We present the strong-scaling results, so the noncontiguous requests partitioned among all p processes are used in our experiments. BTIO has more significant numbers of noncontiguous requests, and the number increases as the number of processes p.

Dataset	# Noncontiguous Requests	Write Amount
E3SM F	1.36×10^9	14GiB
FLASH/IO	$24p$	$8 \times 80^3 p$GiB
BTIO	$1024^2 p$	40GiB

V. Experimental Results

We conduct evaluations for MPI collective write in ROMIO by replacing its metadata and data communication kernels with five all-to-many communication algorithms.

Our implementations are based on the Lustre and GPFS drivers in the MPICH-3.3 release. The default ROMIO implements metadata and data communications by posting `MPI_Isend` and `MPI_Irecv` in ascending rank order followed by `MPI_Waitall`. We explore four communication algorithms for metadata and data exchange: (1) the current ROMIO implementation that posts all requests in ascending rank order with Issend, (2) the spread-out algorithm, (3) the balanced spread-out algorithm, (4) the pairwise algorithm, and (5) TAM. Figure 3 illustrates the communication patterns of these algorithms. For the balanced spread-out algorithm, we use the default rank list ordering $\{0, ..., p-1\}$. For TAM, we use the default setting of 512 for the number of local aggregators. If the number of processes is less than 512, TAM is equivalent to the default ROMIO two-phase I/O. All algorithms except the pairwise algorithm are implemented with the MPI asynchronous functions `MPI_Issend` and `MPI_Irecv`. The pairwise algorithm is implemented by using `MPI_SendRecv`.

We perform all evaluations on two supercomputing systems. One is Cori, a Cray XC40 supercomputer with Intel KNL processors and Lustre file system at the National Energy Research Scientific Computing Center (NERSC). Each Cori KNL node contains one CPU with 68 CPU cores. Compute nodes on Cori are connected with a dragonfly topology. We set the Lustre stripe count to be 64 and stripe size to be 1 MiB. In our experiments, we allocate 64 processes per Cori KNL node. The other supercomputer is Summit, at the Oak Ridge Leadership Computing Facility (OLCF), with IBM Power System AC922 nodes equipped with IBM POWER9 CPUs and IBM GPFS. Each Summit node has two CPUs with 21 CPU cores per socket. The interconnect topology on Summit is a non-blocking fat-tree with a Mellanox EDR 100G InfiniBand connection. The default MPI compilers are Cray-mpich on Cori and IBM Spectrum on Summit. Both MPI implementations are not open-sourced. Nevertheless, their end-to-end timing results can serve as baselines.

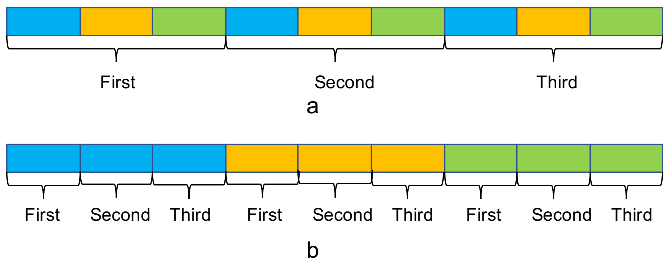


Figure 2: Two-phase I/O implementation for the Lustre and GPFS drivers. Different colors represent file domains owned by different I/O aggregators. We label the file domains handled by different rounds of two-phase I/O. (a) Lustre driver. (b) GPFS driver.

The Lustre driver assigns the number of I/O aggregators equal to the stripe count in a round-robin fashion across nodes by default. We present the performance using default aggregator placement that assigns ranks $64i \forall 0 \leq i < 64$ as I/O aggregators. This setting allows us to evaluate the advantage of the balanced spread-out algorithm over the original spread-out algorithm for all-to-many communication patterns. When the environmental variable AGGREGATOR_PLACEMENT_STRIDE is set to -1 on Cori, the placements of I/O aggregators are fully spread out across nodes, so ranks $\lfloor \frac{n}{64} \rfloor i \forall 0 \leq i < 64$ become I/O aggregators on n nodes for $p > n$. Therefore, the balanced spread-out algorithm becomes identical to the original spread-out algorithm when this hint is set. Moreover, the spread-out algorithm and the evenly spread-out algorithm are identical for experiments on Summit and 64-node experiments on Cori by default, so we do not list the performance of these two algorithms separately. On the other hand, GPFS drivers select the process with the lowest rank as an I/O aggregator for every node by default.

ROMIO implements Lustre and GPFS drivers in multiple two-phase I/O rounds with different access patterns. As illustrated in Figure 2 (a), the file views of an I/O aggregator in the Lustre driver are interleaved contiguous regions of stripe size. For each round of two-phase I/O, the Lustre driver processes one stripe, a contiguous block of data with a size equal to stripe size multiplied by stripe count. Our Lustre setting results at 1MiB (stripe size) collective buffer size.

The GPFS driver evenly divides the entire file domain into the number of I/O aggregators contiguous regions. For each round of two-phase I/O, each I/O aggregator handles a contiguous region of I/O requests within its file domain, as illustrated in Figure 2 (b). Furthermore, the default collective buffer size, the maximum number of bytes processed by an I/O aggregator per round, is 16 MiB in the GPFS driver. Therefore, the number of two-phase I/O rounds performed by the Lustre driver is 16 times that of the two-phase I/O rounds executed by the GPFS driver. Consequently, the numbers of two-phase I/O rounds in Lustre and GPFS drivers are different.

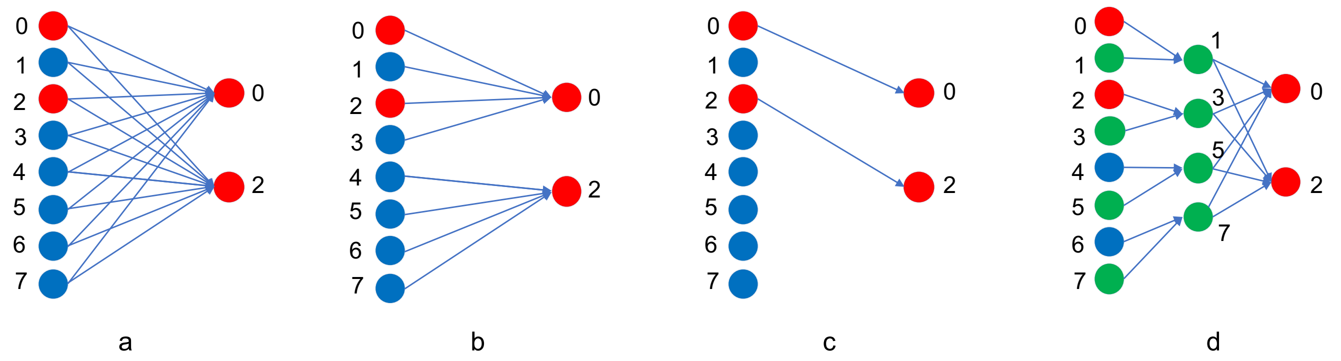


Figure 3: These figures illustrate the differences in the implementations we used for evaluation. We use the blue circles to denote nonaggregators. The red circles are I/O aggregators, and the green circles are local aggregators in TAM language. In this example, we have 2 I/O aggregators, 4 local aggregators, and 8 processes. We label their ranks next to them. The arrows indicate point-to-point communications. (a) Default ROMIO implementation. All communication requests are posted simultaneously. (b) The balanced spread-out algorithm with throttling limit 4. The difference between the original and balanced spread-out algorithm has been illustrated in Figure 1. With throttling, an I/O aggregator receives data from a batch of 4 processes at a time. After receiving from a batch is finished, it receives from the next batch. This figure shows the first round of communication. (c) Pairwise algorithm. An I/O aggregator receives data from 1 process at a time. After receiving data from rank x, it receives data from $x + 1 \mod p$. This figure shows the first round of communication. (d) Two-layer aggregation method (TAM). An arbitrary process sends data to 1 local aggregator. Then all local aggregators exchange data with all I/O aggregators.

Table I summarizes the datasets used in our experiments. The E3SM-IO F case has an all-to-many communication pattern for both metadata and data communication. FLASH/IO has many-to-many instead of all-to-many communication patterns, for both metadata and data. We show that algorithms designed for all-to-many patterns may no longer have advantages over the traditional approach. For BTIO, the metadata communication is all-to-many, and data communication is many-to-many. We can apply different communication strategies that adapt to these two patterns. The conclusion of this case study is that communication algorithms should be selected based on the I/O pattern.

A. E3SM-IO Benchmark

E3SM [5] is an exascale Earth system modeling program for simulating atmosphere, land, and ocean behavior in high resolution. Its I/O module is implemented with PIO library [27], which is built on top of PnetCDF [28]. Checkpointing is implemented by using nonblocking PnetCDF APIs. The PnetCDF library is a high-level parallel I/O library popular in the climate research community; the library is built on top of MPI-IO. Sending nonblocking requests are flushed by aggregating the request data and combining the MPI file views, followed by a single call to the MPI collective write function. The cost of posting the nonblocking APIs is negligible, so the end-to-end performance is almost equivalent to the timing of the collective I/O calls inside the PnetCDF flush API.

We evaluated a particular decomposition used in E3SM production runs, namely, F case [5]. The I/O kernel of E3SM has been extracted for I/O study [29]. F case has atmosphere, land, and runoff model components. The data access pattern in the F case consists of 1.36 billion noncontiguous write requests and a total write amount of 14 GiB. For all experiments in our evaluation, the E3SM-IO F case creates an all-to-many communication pattern for both metadata and data transfer from all processes to I/O aggregators.

To present strong-scaling results, we distribute all I/O requests from production runs evenly across all processes. In other words, the entire offset-length pairs from a process in the production run are assigned to a process. The noncontiguous I/O requests in E3SM-IO do not have a subarray pattern, and adjacent file offsets do not have a regular interval. Thus, the file access patterns of E3SM-IO differ from BTIO.

The end-to-end timing consists of computation, communication, and I/O components. Computation cost is dominated by request offset computation and the heap merge sort for I/O requests at I/O aggregators. I/O cost, by its name, is the cost for writing data gathered at I/O aggregators to file servers. Our focus is to improve the communication cost, so we put the computation and I/O cost into the “other” category.

Figure 4 presents the timing results of the E3SM-IO F case using the balanced spread-out algorithm with different throttling thresholds. The purpose of these experiments is to demonstrate that throttling is important for large-scale all-to-many communications. Figures 4 (a) and (b) show results on Cori 64 and 256 KNL nodes, respectively. As the throttling threshold increases, the communication cost increases sharply. Summit 128-node results in Figure 4 (c), on the other hand, do not have such observable performance degradation. Nevertheless, on 512 nodes, the contention is significant as shown in Figure 4 (d). Both computing systems suffer from oversubscriptions of communication resources caused by an excessive number of MPI requests as the number of processes scales. For the rest of the evaluations, we use

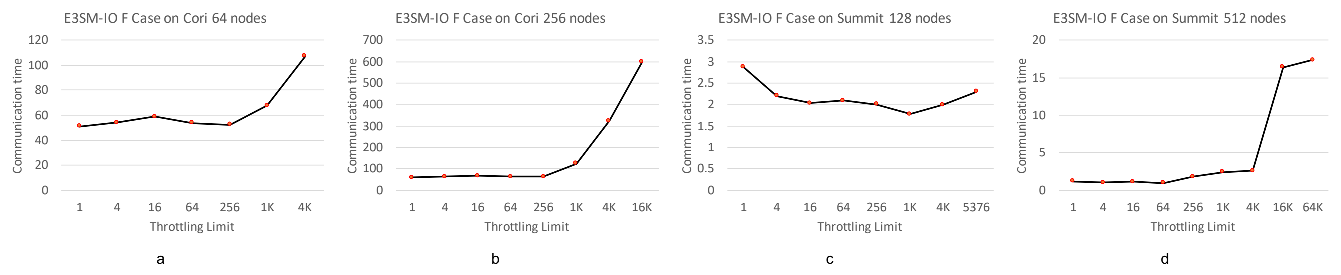


Figure 4: (a)–(d) illustrate the communication time of the E3SM-IO F case using the balanced spread-out algorithm with different throttling threshold. The figure titles identify the dataset, number of nodes, and computing system. The data communication pattern in E3SM-IO is all-to-many. The x-axis depicts the upper bound of the number of concurrent MPI_Irecv in any MPI_Waitall call.

a default throttling threshold 32, which is also adopted by MPICH `MPI_Alltoallw`, for both the original and balanced spread-out algorithms. Tuning this threshold can be important on certain systems, but it is not the focus of this paper.

Figures 5 (a) and (b) illustrate the E3SM-IO F case performance results running on Cori KNL. Both the spread-out algorithms have less metadata and data communication time compared with the other algorithms, which benefits from the communication throttling. Communication costs can be further divided into two parts. The first part is the metadata transfer from all processes to I/O aggregators. The second part is the data transmission from all processes to I/O aggregators based on metadata information. To avoid memory overflow for large files, ROMIO implements two-phase I/O by splitting the communication and I/O operations into multiple rounds. I/O aggregators perform two-phase I/O for requests that lie in a file domain with limited size per round. Thus, there are many consecutive all-to-many communications in the ROMIO implementation. For the Lustre file systems, the file domain has a maximum size of Lustre stripe. For GPFS, the maximum file domain size is the collective buffer size (16 MiB by default) multiplied by the number of I/O aggregators. Furthermore, the proposed evenly spread-out algorithm outperforms the original spread-out algorithm used by `MPI_Alltoallw`, as shown in Figure 5 (b) for 256-node experiments. Thus, avoiding the straggler effect discussed previously can improve the communication performance.

Figures 5 (c) and (d) show E3SM-IO F case performance results on Summit. As the number of processes scales up, ROMIO Issend has significantly higher metadata and data communication time. The spread-out algorithm with throttling can effectively reduce this communication cost, which in turn improves the performance. Different from the Cori results, the communication time of the pairwise algorithm on Summit becomes faster as the number of processes increases because the number of two-phase I/O round is $\lceil \frac{14.07\text{GiB}}{16\text{MiB}\times 512} \rceil = 2$ on Summit 512 nodes. The number of rounds on Cori, on the other hand, is a constant 225 regardless of the number of processes used given stripe count 64 and size 1 MB. The pairwise algorithm is an effective design for all-to-all personalized communication. However, the straggler effect for many `MPI_Recv`s slows performance significantly as the number of rounds increases for all-to-many communication. Asynchronous communication, on the other hand, suffers less from straggler effects because multiple communications are processed simultaneously.

The two-layer aggregation method (TAM) has a performance advantage with our settings compared with the rest of the algorithms. When TAM is used, the participants of the internode metadata and data communications are limited to local and I/O aggregators, so the communication scale is much smaller compared with explicit all-to-many communications. Instead of reducing contentions by throttling and communication reordering, TAM eliminates the contention implicitly by reducing the communication size. The reduction of communication contention by TAM is at the expense of casting extra intranode communication and memory footprints because the intranode aggregation phase requires local aggregators to receive extra data from other data within the same node. However, such extra costs are negligible compared with the reduction in communication cost, so TAM has a huge advantage over other explicit throttling approaches. The communication kernel of TAM is the same as the one adopted by default in ROMIO, which posts all send/receive requests in ascending rank order followed by a wait all operation. Unfortunately, we have not found any compelling evidence that replacing the communication kernels of two-layer aggregations with any all-to-many algorithms discussed in this paper improves performance with our benchmarks that run on a few hundred compute nodes. Nevertheless, when the number of nodes scales from hundreds to tens of thousands in exascale computing, we expect that the overwhelming number of MPI_Irecv operations queued at receivers will become a bottleneck for the internode aggregation of TAM because of the reasons discussed in the implementation section. Theoretically, the internode aggregation stage of TAM can suffer from performance degradation caused by communication contentions, even if only one process per node participates in the inter-node aggregation. As a result, replacing its internode communication kernel with the balanced spread-out algorithm will

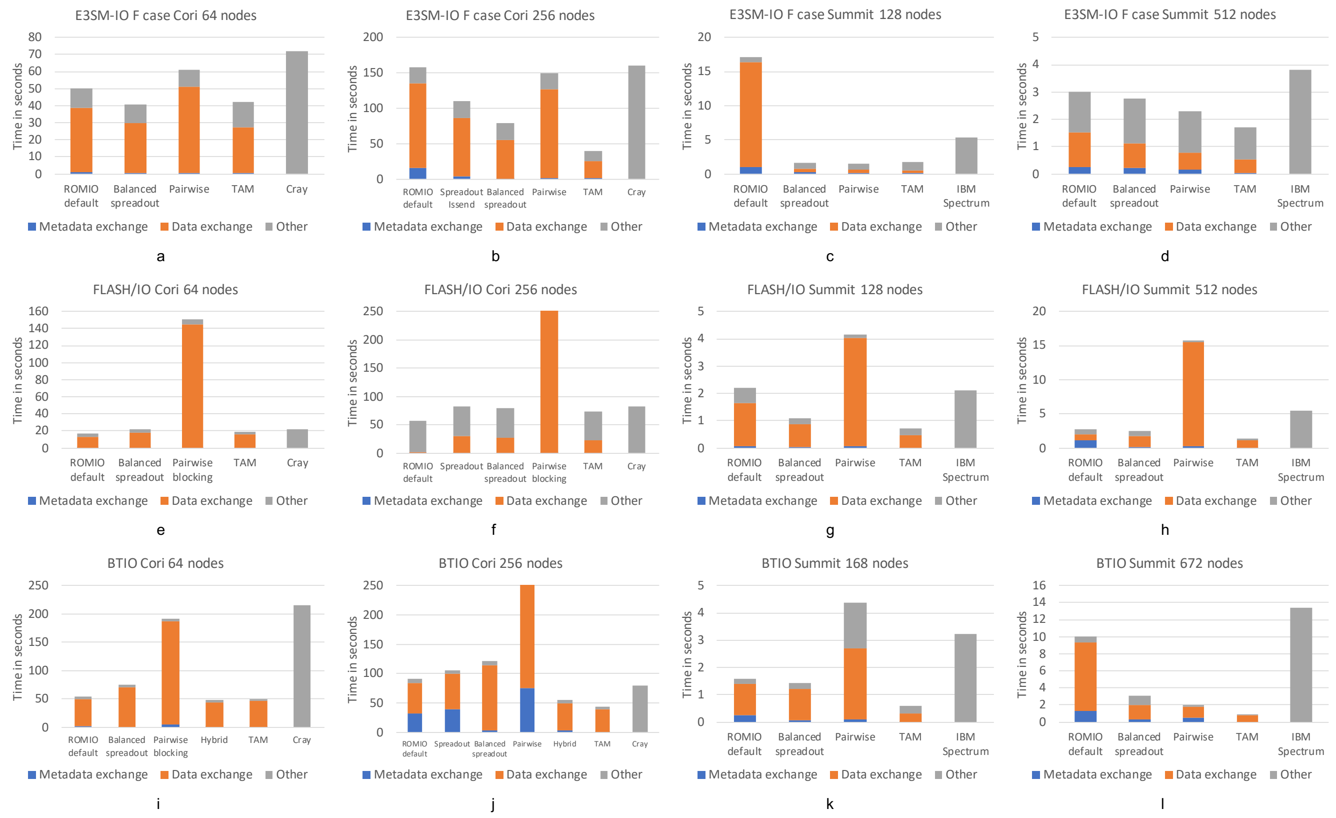


Figure 5: (a)–(l) are breakdown timings of different communication kernels on Cori and Summit. The figure titles indicate the dataset, number of nodes, and computing system. Cray-mpich and IBM Spectrum MPI libraries are not open-sourced, so we present their end-to-end time only. "ROMIO default" means that the current version of ROMIO is used. "Hybrid" means that we use the ROMIO default for metadata exchange and the balanced spread-out algorithm for data exchange. (a) E3SM-IO F case on 64 Cori KNL nodes. (b) E3SM-IO F case on 256 Cori KNL nodes. (c) E3SM-IO F case on 128 Summit nodes. (d) E3SM-IO F case on 512 Summit nodes. (e) FLASH/IO on 64 Cori KNL nodes. (f) FLASH/IO on 256 Cori KNL nodes. (g) FLASH/IO on 128 Summit nodes. (h) FLASH/IO on 512 Summit nodes. (i) BTIO on 64 Cori KNL nodes. (j) BTIO on 256 Cori KNL nodes. (k) BTIO on 128 Summit nodes. (l) BTIO on 512 Summit nodes.

help scale up TAM in the future for tens of thousands of compute nodes.

B. *FLASH/IO Benchmark*

The FLASH I/O benchmark suite is the I/O kernel of a block-structured adaptive mesh hydrodynamics code developed for the study of nuclear flashes on neutron stars and white dwarfs [3]. We use the checkpoint variables I/O, whose data pattern is an array of 24 variables that consist of 3D blocks of data, to evaluate communication kernels of two-phase I/O. We use a default block size of 8^3. Each 3D block has a fourth dimension of size one, so the flatten offsets of any 3D block are contiguous. A variable distributes its 3D blocks evenly to all processes. Each process has a fixed number of 80 to 82 3D blocks, so variable data sizes scale with the number of processes. As a result, FLASH/IO is a weak-scaling benchmark.

Figure 6 illustrates an example of I/O access orders of ROMIO two-phase I/O in multiple rounds on Lustre file systems with stripe size 3. Data gathered by I/O aggregators are illustrated in different colors. To simplify the illustration, we assume I/O data at all processes have

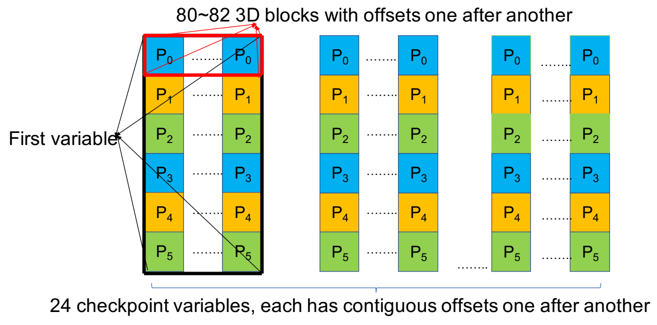

80~82 3D blocks with offsets one after another
First variable
P_0
P_1
P_2
P_3
P_4
P_5
24 checkpoint variables, each has contiguous offsets one after another

Figure 6: The ROMIO Lustre collective write driver partitions FLASH/IO data as the figure show. There are 24 checkpoint variables, each with contiguous offsets represented in 3D blocks. The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans three rows of file domains (blue, brown, and green) within a variable in this example. Both metadata and data communication patterns are not all-to-many.

the same size. In this example, each checkpoint variable spans two Lustre stripes, so there are two rounds of two-phase I/O per variable. Both metadata and data communication have many-to-many patterns instead of all-to-many. For instance, the aggregator with light-blue file domain receives I/O requests from process ranks 0 and 3 only during metadata exchange. It receives data from process rank 0 or 3 exclusively per two-phase I/O round.

In general, the communication contention in FLASH I/O is less significant than contentions in BTIO and E3SM F cases. With 80 3D blocks of 8^3 double-precision floating-point values, a process has $80 \times 8^3 \times 8\text{B} = 327680$ bytes of data. With our Lustre setting on Cori, an aggregator receives data from at most $\lceil \frac{1\text{MiB}}{327680\text{B}} \rceil = 4$ processes per two-phase I/O round. On the other hand, I/O aggregators on Summit receive data from $\lceil \frac{16\text{MiB}}{327680\text{B}} \rceil = 53$ processes per round. Furthermore, 1,260 processes send metadata to an aggregator on average for the GPFS driver running on 512 Summit nodes. The Lustre driver running on Cori KNL nodes, on the other hand, has only MPI 21 receive requests per I/O aggregator on average at the metadata exchange stage. Therefore, data exchange on Summit has a larger degree of contention compared with data exchange on Cori.

Figures 5 (e) to (h) summarize the performance results of writing FLASH I/O checkpoint in parallel. With the pairwise blocking kernel, the communication time for all experiments is slower compared with the rest of asynchronous communication implementations. For Cori results in Figures 5 (e) and (f), we present their timing values up to 250 seconds. However, the actual communication time is more than 20 minutes. The pairwise algorithm, implemented with `MPI_SendRecv`, constrains the order of data received by different aggregators with the blocking MPI functions. An aggregator may stay idle, waiting for communication of a process to finish, even though the aggregator does not receive any data from the process. Thus, applying the pairwise algorithm, designed for an all-to-all pattern, to a many-to-many communication pattern can cause a significant straggler effect. On Cori, algorithms with throttling techniques do not show an advantage over the original version that posts all requests in ascending rank order. With throttling comm_size 32, each `MPI_Waitall` handles at most one send request at a sender, so such an over-throttling becomes an overhead. On GPFS, I/O aggregators receive data from a larger number of senders, so data communication benefits from throttling, as shown in Figures 5 (g) and (h).

C. BTIO Benchmark

BTIO, developed by NASA's Advanced Supercomputing Division, is part of the benchmark suite NPB-MPI version 2.4 for evaluating the performance of parallel I/O [12]. I/O requests in BTIO are partitioned in a block-tridiagonal pattern over a three-dimensional square array. Therefore, the BTIO benchmark must run with a square number of MPI processes, which divide the cross-sections of a global 3D array evenly. Thus, we increment the number of nodes on Summit to 168 and 672. Figure 7 illustrates an example of the data partition pattern of BTIO with 9 processes and 3 aggregators. The three-dimensional array is divided into 27 subarrays of size $3 \times 3 \times 3$. All I/O requests within a subarray are located on a single process. We can view the BTIO data pattern as three planes of 3D subarrays of size $3 \times 3 \times 3$. There are $\sqrt{p}$ permutations of subarrays owned by processes on the planes. The first permutation is a row-major ordering of subarrays. In our example, Figure 7 (a) illustrates the first 2D subarray in the 3D array with this permutation method. The next permutation is by decreasing the subarrays row index and increasing the column index, taking the modulus of $\sqrt{p}$. For example, the next subarray plane is illustrated by Figure 7 (b) transformed from the initial row-major permutation. Figure 7 (c) shows the last subarray plane.

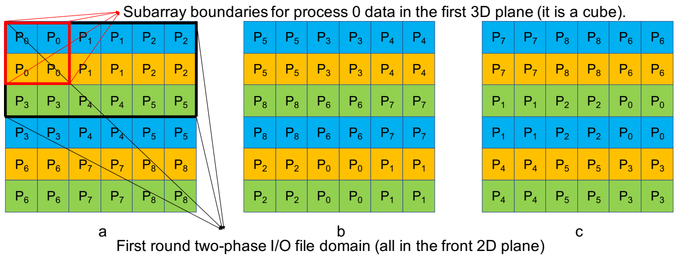


Figure 7: BTIO is a 3D array. The ROMIO Lustre collective write driver partitions BTIO data with three permutations (along the third dimension) represented by (a), (b), and (c). The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans a file domain (blue, brown, and green) of three rows of the 2D array. The metadata communication pattern is all-to-many, but the data communication pattern is not.

With multiple rounds of two-phase I/O implementation in ROMIO on Lustre, an I/O aggregator handles only a limited range of file domains per round. One round of two-phase I/O handles a contiguous region of file domains, evenly divided among I/O aggregators. In our example, we use three colors to represent the file domains at three I/O aggregators. I/O requests marked with the same color are aggregated to the same aggregator. Two-phase I/O in a 2D array of I/O requests completes in two rounds. For example, the top three rows in Figure 7 (a) are completed in the first two-phase I/O run. The bottom three rows are completed in the second two-phase I/O run. Consequently, for any two-phase I/O round, an I/O aggregator gathers data from at most 3 processes, so the data exchange pattern is many-to-many instead of all-to-many. Metadata exchange has a single round. I/O aggregators gather offset/length pairs for noncontiguous requests from all processes in our example, which exhibits an all-to-many communication pattern. Therefore, the metadata and data communication patterns can be different in the BTIO benchmark.

Figures 5 (i) and (j) illustrate the breakdown performance for running two-phase I/O on 64 and 256 Cori KNL nodes. The evenly spread-out algorithm has the best metadata exchange performance. However, the original ROMIO `MPI_Issend` implementation yields the best data exchange performance. As mentioned earlier, the data exchange pattern of BTIO is not all-to-many, so the original and balanced spread-out algorithms may underutilize the communication channels because of the throttling approach. The BTIO size has 1,024 40 MiB 2D data arrays stacked side by side. Each of the 2D data arrays is evenly divided among all processes in a different permutation of square blocks. Hence an aggregator gathers 1 MiB of data from approximately $\sqrt{p}$ processes per two-phase I/O round on the Lustre file systems. Thus, the size of communication is much less than the all-to-many communication for metadata exchange. Similar to FLASH/IO, throttling can have a negative effect since the number of requests handled by `MPI_Waitall` is too small. To optimize communication performance, we use the original ROMIO communication kernel for data exchange and the evenly spread-out algorithm for the metadata communication. The results presented in the hybrid column are better than the results of all other columns. Thus, the proposed all-to-many algorithms are not universal for any type of many-to-many communication in multiple rounds. When the many-to-many communication size is small, the I/O driver should fall back to the original two-phase I/O implementation without throttling to improve communication.

Figures 5 (k) and (l) illustrate the performance breakdown on Summit. Since the collective buffer size on GPFS is 16 MiB by default, almost half of the data in a 2D data array is collected by an aggregator. Thus, an I/O aggregator gathers data from approximately $\frac{1}{2}p$ of the processes. Although the communication pattern is not all-to-many, the number of senders is large enough to cause contention. Consequently, unlike the results on Cori, the evenly spread-out algorithm has an advantage over the traditional ROMIO Issend approach. The number of I/O aggregators scales up with the number of nodes on Summit, so there are fewer two-phase I/O rounds as the number of processes increases. Therefore, data communication with the pairwise algorithm is faster on 672 nodes than 168 nodes because straggler effects resulting from multiple rounds of data communications on Summit are smaller on a larger number of nodes. Our experiments on Cori, on the other hand, have a fixed number of I/O aggregators. Hence the pairwise algorithm slows down on a large number of nodes because the straggler effects also scale up with the increasing number of processes.

VI. Conclusion

Metadata and data communication become performance bottlenecks of two-phase I/O when the communication pattern is all-to-many in multiple rounds. In this paper, we replaced the two-phase I/O communication kernels In ROMIO that adapt to input I/O patterns, instead of using different all-to-many personalized communication algorithms. Experimental results demonstrated the necessity of communication balancing and throttling for all-to-many communication patterns. Moreover, we expect that adjusting personalized all-to-many communication kernels can further reduce communication contentions for existing communication designs such as the two-layer aggregation, especially when the number of compute nodes scales to meet the demand of exascale computing in the future. However, when an input I/O pattern does not exhibit an all-to-many communication pattern, two-phase I/O should fall back to the implementation without communication throttling.

Several opportunities remain for future research. The communication kernels are chosen by users with MPI hints to adapt to their application I/O patterns. It is useful to let the I/O drivers parse the I/O pattern and dynamically select appropriate communication kernels. Furthermore, the communication throttling approach can take advantage of compute node topology information. Topology-aware throttling can reduce communication contention at network switches, which in turn further improves communication performance.

A. *Acknowledgment*

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 is 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 Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

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

[2] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukic, S. Sehrish, and W. keng Liao, “Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures,” *New Astronomy*, vol. 42, pp. 49–65, July 2015.

[3] R. Latham, C. Daley, W. keng Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, “A case study for scientific i/o: Improving the flash astrophysics code,” *Computer and Scientific Discovery*, vol. 5, March 2012.

[4] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, *et al.*, “Terascale direct numerical simulations of turbulent combustion using s3d,” *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, 2009.

[5] P. M. Caldwell, A. Mametjanov, Q. Tang, L. P. Van Roekel, J.-C. Golaz, W. Lin, D. C. Bader, N. D. Keen, Y. Feng, R. Jacob, *et al.*, "The doe e3sm coupled model version 1: Description and results at high resolution," *Journal of Advances in Modeling Earth Systems*, 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," *Journal of Physics: Conference Series*, no. 78, 2007.

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

[8] R. Thakur, R. Ross, E. Lusk, and W. Gropp, "Users guide for romio: A high-performance, portable mpi-io implementation," Tech. Rep. 234, MCS dividion, Argonne National Lab., IL (United States), May 2002.

[9] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk, "Non-data- in mpi: analysis on blue gene/p," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 13–22, Springer, 2008.

[10] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, p. 52, ACM, 2007.

[11] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law, "Direct numerical simulations of turbulent lean premixed combustion," in *Journal of Physics: conference series*, vol. 46, p. 38, IOP Publishing, 2006.

[12] P. Wong and R. Der Wijngaart, "Nas parallel benchmarks i/o version 2.4," *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, 2003.

[13] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2682–2695, 2020.

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*, vol. 400. Benjamin/Cummings, 1994.

[15] "Mpich3." http://www.mpich.org/downloads/, 2017.

[16] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[17] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[18] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[19] J. L. Träff and A. Rougier, "Mpi collectives and datatypes for hierarchical all-to-all communication," in *Proceedings of the 21st European MPI Users' Group Meeting*, pp. 27–32, 2014.

[20] C. Xu, M. G. Venkata, R. L. Graham, Y. Wang, Z. Liu, and W. Yu, "Sloavx: Scalable logarithmic alltoallv algorithm for hierarchical multicore systems," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 369–376, IEEE, 2013.

[21] "Mvapich: Mpi over infiniband, 10gige/warp and roce." http://mvapich.cse.ohio-state.edu. Accessed: 2020.

[22] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "Mpi+threads: Runtime contention and remedies," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 239–248, 2015.

[23] M. Si and P. Balaji, "Process-based asynchronous progress model for mpi point-to-point communication," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 206–214, IEEE, 2017.

[24] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, *et al.*, "The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 94–103, 2008.

[25] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an asynchronous message driven programming paradigm on ibm blue gene/q," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 689–699, IEEE, 2013.

[26] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the cray linux environment core specialization feature to realize mpi asynchronous progress on cray xe systems," in *Proceedings of the Cray User Group Conference*, 2012.

[27] J. Edwards, J. Dennis, M. Vertenstein, and E. Hartnett, "Parallel io libraries (pio) - high-level parallel i/o libraries for structured grid applications." https://github.com/NCAR/ParallelIO. Accessed: 2019-05-23.

[28] "Pnetcdf." www.mcs.anl.gov/parallel-netcdf/, 2019.

[29] "Parallel i/o kernel case study – e3sm." https://github.com/Parallel-NetCDF/E3SM-IO. Accessed: 2019-03-06.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We run FLASH/IO, BTIO, and E3SM-IO on NERSC's Cori KNL nodes with default Cray MPICH and our customized MPICH implementation (for evaluating proposed algorithms). We also run the same set of benchmarks on OLCF Summit nodes. The modified ROMIO libraries have ad_lustre and ad_gpfs drivers that can be built on Cori and Summit. Benchmarks can be built by linking to these libraries. We have listed all Github URLs in the Artifacts Available (AA) section. For experiments, we use 64/256 KNL nodes on Cori and 128/512 IBM power9 nodes on Summit to run these benchmarks. On Cori, we used Lustre file system with stripe size 1MB and count 64. On Summit, we used GPFS file system (GPFS setting does not require us to set anything) We describe the details of benchmarks in the following.

FLASH/IO: part of PnetCDF version 1.12.1. We used a block size 8*8*8 for all experiments. Available on https://github.com/Parallel-NetCDF/PnetCDF/tree/master/benchmarks/FLASH-IO.

BTIO: part of NPB-MPI version 2.4. Execution mode is PnetCDF nonblocking I/O. XYZ grid is 1024*1024*1024 for all experiments. Available on https://github.com/Parallel-NetCDF/BTIO with (git master branch commit hash @d45c270)

E3SM-IO: version v.1.1.0 available on https://github.com/Parallel-NetCDF/E3SM-IO (git master commit hash @ed682a3). We use decomposition file f_case_72x777602_21632p.nc to carry out the experiments in our paper. This file picks up all NetCDF parameters.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: DOI: 10.5281/zenodo.3932426, GitHub
↪  URL: https://github.com/QiaoK/mpich/tree/ad\_lus⌋
↪  tre\_alltoall
Artifact name: Source code for all types of
↪  communication algorithms in this paper (modified
↪  ROMIO in MPICH)
Citation of artifact: Pavan Balaji, Dave Goodell, Ken
↪  Raffenetti, William Gropp, Wesley Bland, Hui
↪  Zhou, . . .  Shintaro Iwasaki. (2020, July 7).
↪  QiaoK/mpich alltoall1.0 (Version alltoall 1.0).
↪  Zenodo. http://doi.org/10.5281/zenodo.3932426
```

```
Persistent ID: DOI: 10.5281/zenodo.3932435, GitHub
↪  URL: https://github.com/QiaoK/mpich/tree/ad\_lus⌋
↪  tre\_tam2
Artifact name: Source code for TAM (modified ROMIO in
↪  MPICH)
Citation of artifact: Pavan Balaji, Dave Goodell, Ken
↪  Raffenetti, William Gropp, Wesley Bland, Hui
↪  Zhou, . . .  Shintaro Iwasaki. (2020, July 7).
↪  QiaoK/mpich: Two layer aggregation method
↪  (Version tam1.0). Zenodo.
↪  http://doi.org/10.5281/zenodo.3932435
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: NERSC Cori and OLCF Summit super computers

Operating systems and versions: Cray XC40 system and Red Hat Enterprise Linux

Compilers and versions: module craype-mic-knl on Cori and default IBM Spectrum on Summit

Applications and versions: FLASH/IO, BTIO, E3SM-IO, S3D-IO

Libraries and versions: MPICH 3.3

Key algorithms: two-phase I/O