# On Overlapping Communication and File I/O in Collective Write Operation

Raafat Feki [*], and Edgar Gabriel[*]

[*] Parallel Software Technologies Laboratory,
Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: {rfeki, egabriel}@uh.edu

***Abstract*—Many parallel scientific applications spend a significant amount of time reading and writing data files. Collective I/O operations allow to optimize the file access of a process group by redistributing data across processes to match the data layout on the file system. In most parallel I/O libraries, the implementation of collective I/O operations is based on the two-phase I/O algorithm, which consists of a communication phase and a file access phase. This papers evaluates various design options for overlapping two internal cycles of the two-phase I/O algorithm, and explores using different data transfer primitives for the shuffle phase, including non-blocking two-sided communication and multiple versions of one-sided communication. The results indicate that overlap algorithms incorporating asynchronous I/O outperform overlapping approaches that only rely on non-blocking communication. However, in the vast majority of the testcases one-sided communication did not lead to performance improvements over two-sided communication.**

## I. Introduction

Many scientific applications operate on data sets that span hundreds of Gigabytes or even Terabytes in size. Yet, even high end computing systems often lack the file I/O performance required to manage these enormous amounts of data. Consequently, many applications spend a significant amount of time reading and writing input and output files.

The Message Passing Interface (MPI) [15] is the most widely used parallel programming paradigm for large scale parallel applications. MPI incorporates since version 2 of the specification interfaces for parallel file I/O, often referred to as MPI I/O. By extending the MPI concepts to file I/O operations, multiple processes can simultaneously access the same file to perform read and write operations. This shared-file access pattern propagated by MPI I/O reduces the number of files created by an application, but may result in sub-optimal performance due to contention occurring on the file system level, e.g. because of the file locking.

Furthermore, the data decomposition used by parallel applications often leads to a non-contiguous data layout on the file system level from each individual processes perspective, which will typically result in a large number of small I/O requests and significant performance degradation of file I/O operations. To resolve this problem, MPI I/O introduced the concept of collective I/O operations. Collective I/O operations represent a higher level abstraction of file I/O operations executed by a group of processes, and allow for internal optimizations such as rearranging data across processes to match the data layout on the file system level, and merge individual I/O requests of different processes into fewer and larger global requests.

The most widely used collaborative technique used within collective I/O implementations is based on the two-phase I/O algorithm [8]. This algorithm consists of a communication phase (also referred to as the shuffle phase) and a file access phase. A subset of the processes, the so-called aggregators, are designated to perform the I/O operations in this algorithm. In the shuffle phase, data is redistributed to the aggregators by using inter-process communication, each aggregator being in charge of a contiguous partition of data in the file. In the file access phase, each aggregator issues the file I/O request to perform the actual read and write operations. If the overall data volume of the collective I/O operation exceeds certain thresholds, the two-phase algorithm is executed internally in multiple cycles in order to keep the additional memory requirements on the aggregator processes within reasonable limits. The shuffle phase is ultimately the reason for the performance improvement of collective I/O operations compared to individual I/O, as well as the main source for overhead in this algorithm.

Researchers have developed numerous techniques to optimize the performance of the two-phase I/O algorithm. A number of projects are focusing on overlapping multiple internal cycles of the shuffle phase and the file access phase. Sehrish et al.[18] and Tsujita et al. [24] applied this approach by using asynchronous communication and multi-threading, respectively. However, very little work has been done in exploring **what** operations are being overlapped, and **how** the shuffle phase is implemented.

The contribution of this paper is two-fold. First, we present and evaluate various design options for overlapping two internal cycles of the two-phase I/O algorithm, using both, the communication as well as the file access phase. Second, the paper explores using different data transfer primitives for the shuffle phase, such as non-blocking two-sided communication (i.e. `Isend/Irecv`) and multiple versions of one-sided communication (`Put`). In both instances, the focus of this work is on collective write operations. Our results indicate that overlap algorithms incorporating asynchronous I/O outperform overlapping approaches that only rely on non-blocking communication in most scenarios, and offer significant performance benefits of up to 22% compared to two-phase I/O algorithm not overlapping any internal operations. However, in the vast majority of the testcases one-sided communication did not lead to performance improvements over two-sided communication.

The remainder of the paper is organized as follows. Section II presents the most relevant related work in this domain. In section III, we detail the design and implementation of the different overlap methodologies and data transfer primitives. Performance evaluation is presented and analyzed in section IV. Finally, we present our conclusion in section V.

## II. Related Work

Parallel I/O represents a wide area of research, with collective I/O operations being one of the focal points. Collective I/O operations represent a higher level abstraction of file I/O operations executed by a group of processes, and allow for internal optimizations such as rearranging data across processes to match the data layout on the file system level, and merge individual I/O requests of different processes into fewer and larger global requests.

The ability of processes to coordinate file I/O requests is generally considered to be the most relevant aspect for achieving good performance for I/O operations in parallel applications. The most widely used approach used for collective I/O as of today is the two-phase I/O [8] algorithm. This algorithm shuffles the data among the MPI processes to match the data layout in the file and performs the actual read/write operation on a subset of processes (so called aggregators). Two phase I/O has been extended in a number of research projects to optimize for various scenarios, including datatype I/O [21], view-based collective I/O [3], resonance I/O [26], and dynamic segmentation and static segmentation algorithms [4]. Liao et.al. [11] analyzed various file domain partitioning methods for improving the performance of collective I/O operations. Most of these algorithms tweak on how the shuffle and read/write operations are performed, but use the same fundamental approach as two-phase I/O uses. Numerous researchers [16], [14], [13] have furthermore focused on collective buffering techniques to improve the performance of collective I/O operations. Wang at el. introduced internal pipeling to manage collective I/O operations and limit the number of processes posting I/O requests to a storage server [25].

Some researches focused on optimizing the implementation of two-phase I/O by proposing a topology-aware solution for data aggregation. Weifeng Liu et al. [12] utilized the Linear Assignment Problem (LAP) to efficiently assign file domains to aggregators. The new data distribution over the aggregators reduces the total hop-bytes of collective I/O operation and thus improving the performance.

Yuichi Tsujita et al. targeted different optimization techniques in their paper[23] by aligning the aggregator data to the striping access pattern and enhance it with a round robin algorithm for data distribution over multiple aggregators within a same node in order to alleviate the data transfer contention.

As part of their work on View-based I/O, Blas et al. [3] also reported on improved collective I/O performance by using read-ahead for collective read operations, and use multiple threads to perform these operations in the background, effectively overlapping the read operations with other ongoing operations.

The work most closely related to this paper has been performed by Sehrish et al. [18]. The authors proposed an overlapping scheme similar to one of our proposed models, implementing the operations using non-blocking point-to-point operations. This paper did however not consider what part of the collective I/O operations can be overlapped, nor have they explored using different data transfer primitives. Similarly, Tsujita et al. [24] applied a similar approach by using asynchronous multi-threading.

One-sided communication has been used in numerous studies to optimize communication in parallel applications [2], [7]. In collective I/O, the current ROMIO [22] implementation of the two-phase algorithm supports a one-sided communication version of the shuffle phase in which they used passive-target synchronization. Tessier et al.[20], proposed an algorithm in which they used double-buffering and one-sided communication. However, their method included only active-target synchronization.

## III. Design and Implementation

In the following, we introduce first four different design options for overlapping two internal cycles of the two-phase I/O algorithm, focusing on collective write operations. Second, we explore using different data transfer primitives for the shuffle phase, including non-blocking two-sided communication and multiple versions of one-sided communication.

### A. Overlap Algorithms

In the original implementation of the two-phase I/O algorithm, all processes have to send their local data to the aggregators via inter-process communication. Each aggregator holds data from multiple processes in a temporary buffer. During the file access phase, the aggregators issue I/O requests to the file system to write the content of the temporary buffer onto disk. The next shuffle phase cannot be initiated until the temporary buffer is completely flushed and ready to receive the next chunks of data. These two phases are carried out in multiple cycles until the entire data is written.

In the overlapped two-phase I/O, the temporary buffer is divided into two separate sub-buffers such that the size of each one would be equal to the half of the buffer size in the original implementation. By doing so, we can run different operations in each sub-buffer, hence overlap the shuffle phase and the I/O phase.

Two different operations are performed in each sub-buffer: 1) Inter-process communication (send/receive operations), and 2) I/O operations. Both operations might be called using either blocking or non-blocking implementations. The use of non-blocking functions over two collective sub-buffers is the main key to perform overlapping. However, there are multiple ways to implement the overlapping technique depending on the choice of the asynchronous functions and the phases that will be overlapped. Hence, we propose four different approaches.

In the following, we describe the various overlap algorithms. A brief note on the terminology used: whenever an algorithm utilizes a non-blocking or asynchronous code section, we use the the postfixes `_init` and `_wait` to indicate the start and the end of the non-blocking section, while code sections not having these postfixes indicate blocking implementations. Hence, `shuffle` and `write` represent blocking versions of the shuffle and I/O phase respectively, while e.g. `shuffle_init` and `shuffle_wait` indicate the start and the completion of the non-blocking version of the shuffle operation.

*1) Communication Overlap:* In this first approach, non-blocking send/receive operations are being used while maintaining a blocking version of the write operation. Algorithm 1 provides a high level overview of this approach. A shuffle phase is initiated on the first sub-buffer $p_1$ (`shuffle_init`) before the start of the internal cycles. In each cycle, a shuffle phase is started, followed by a wait operation of the previous shuffle operation (`shuffle_wait`). The completion of the first shuffle operation means that all required data has been received in the sub-buffer and the buffer is ready to be written. While a write operation is ongoing, the other shuffle operation continues running in the background. Since the I/O operation is synchronous, the next cycle cannot be started until the file I/O phase is completed on the aggregator. Then, pointers representing the two sub-buffers are being swapped (`swap_buffer_pointers`), ensuring that buffer used in the last cycle in the shuffle operation is being used in the write step next, and vice versa.

**Algorithm 1** Communication Overlap

```
Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$
 1: $p_1 \leftarrow tempbuf_1$
 2: $p_2 \leftarrow tempbuf_2$
 3: shuffle_init ($p_1$)
 4: for i=1 to NumberOfCycles do
 5:     shuffle_init ($p_2$)
 6:     shuffle_wait ($p_1$)
 7:     write ($p_1$)
 8:     swap_buffer_pointers ($p_1$,$p_2$)
 9: end for
10: shuffle_wait ($p_1$)
11: write ($p_1$)
```

Some notes on the nomenclature used in this and all subsequent algorithms: function names ending with `_init` represent a code section that initiates communication and/or file I/O operations and will not block during execution, while functions ending with `_wait` represent code sections enforcing the completion of previously initiated operations. A function without either of those endings represent a blocking code section, which could however be implemented using a sequence of initiation and completion, but doesn't necessarily have to.

Non-blocking shuffle operations can be implemented using non-blocking point-to-point data transfer operations in MPI, such as `MPI_Isend` and `MPI_Irecv`, and `MPI_Wait` for the completion. An important performance consideration for this code version will however be the ability of the MPI library to make progress after initiating the communication operations. MPI libraries provide progress for pending non-blocking data transfer operations either when invoking an MPI function, or more recently also through a specific progress thread [10].

*2) Writes Overlap:* This algorithm represents the counterpart to the Communication-Overlap version discussed above, using however blocking shuffle steps and asynchronous write operations. Asynchronous write operations can be implemented for example using the `aio_write()` functionality ( and although slightly beyond the scope of this paper, it should be noted however that quality of the support for this function is dependent on the file system used for the file I/O operations). Within the scope of this paper, the assumption is that using `MPI_File_iwrite` will provide the method best suited on the given file system.

**Algorithm 2** Write Overlap

```
Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$
 1: $p_1 \leftarrow tempbuf_1$
 2: $p_2 \leftarrow tempbuf_2$
 3: shuffle ($p_1$)
 4: write_init ($p_1$)
 5: for i=1 to NumberOfCycles do
 6:     shuffle ($p_2$)
 7:     write_init ($p_2$)
 8:     write_wait ($p_1$)
 9:     swap_buffer_pointers ($p_1$,$p_2$)
10: end for
11: write_wait ($p_2$)
```

Whether the Communication or the Write Overlap algorithm is expected to lead to better performance depends on multiple factors, most notably: i) costs of the shuffle phase vs. the file I/O phase and ii) whether the MPI library or the Operating System is better at ensuring progress of communication operations or I/O requests in the background. In our experience, the file I/O phase is on most clusters more expensive than the shuffle phase. Furthermore, since `aio_write` operations are often execute by an OS thread, progress of non-blocking write operations will often be better than progress of communication operations when not using a separate progress thread.

*3) Write-Communication Overlap:* The third approach uses a non-blocking implementation of the write operation as well as for the shuffle phase. In each cycle, the algorithm initiation an asynchronous write operation on the first sub-buffer followed by initiating a shuffle stage on the second sub-buffer. Algorithm 2 shows this approach.

**Algorithm 3** Write-Communication Overlap

```
Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$
1: $p_1 \leftarrow tempbuf_1$
2: $p_2 \leftarrow tempbuf_2$
3: shuffle ($p_1$)
4: for i=1 to NumberOfCycles do
5:     write_init ($p_1$)
6:     shuffle_init ($p_2$)
7:     wait_all ($p_1$, $p_2$)
8:     swap_buffer_pointers ($p_1$,$p_2$)
9: end for
```

*4) Write-Communication-2 Overlap:* A slightly revised version of the approach shown in algorithm 2 is given by avoiding that the non-blocking shuffle and non-blocking write operation finish approximately at the same time. Instead, this version follows more closely a data-flow model, in that the completion of any non-blocking operation is immediately followed by posting the follow-up operation first. Unlike the other versions, this algorithm handles two shuffles and two writes operations in each iteration which correspond to 2 cycles.

**Algorithm 4** Write-Communication-2 Overlap

```
Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$
1: $p_1 \leftarrow tempbuf_1$
2: $p_2 \leftarrow tempbuf_2$
3: shuffle ($p_1$)
4: write_init ($p_1$)
5: for i=1 to NumberOfCycles do
6:     write_wait ($p_2$)
7:     shuffle_init ($p_2$)
8:     shuffle_wait ($p_1$)
9:     write_init ($p_1$)
10:    shuffle_wait ($p_2$)
11:    write_init ($p_1$)
12:    write_wait ($p_1$)
13:    shuffle_init ($p_1$)
14: end for
```

## B. Data Transfer Primitives

Whereas we focused in the previous section on identifying the different possible algorithms to implement the overlapping technique, we will discuss in this section two possible communication models that can be used for the shuffle phase: Two sided communication (send/receive) and one sided communication (Put/Get).

*1) Two-sided Communication:* The current implementations of the two-phase algorithm uses a two-sided communication model. During the shuffle phase, MPI processes send the data from their local buffer to the corresponding aggregator using non-blocking communication (`MPI_Isend`, `MPI_Irecv`). Internally, MPI libraries typically use different protocols for short and long messages, namely an *eager* and a *rendezvous* protocol. For short message, the eager protocol sends the data to the receiving process, independent of whether the receiver is ready to receive the data item or not. If the receiver has not yet posted the matching receive operation, the data will be buffered in an *unexpected message* queue on the receiver process. Consequently, a receive operation has to check the unexpected message upon posting the operation, which can be costly if the queue contains many messages.

MPI libraries utilize a rendezvous protocol for long messages, which requires a hand-shake between sender and receiver process. This ensure that large messages will not end up in the unexpected message queue of the receiver processes, limiting the additional memory requirement on that processes. However, the rendezvous protocol prevents a sender from continuing execution until the receiver process is ready to receive the data. For the MPI library and network interconnect used in the evaluation section IV (Open MPI master using UCX 1.6.1 on an InfiniBand network), the rendezvous protocol is used for messages starting from 512 KBytes in size.

In collective I/O operations, a large number of processes are communicating with a few aggregator processes. Since aggregator processes have significantly higher workload than non-aggregators – due to the file I/O access operations that they have to perform –, two-sided communication will typically result in many entries in the unexpected message queue of an aggregator, or, if the messages are too large, the rendezvous protocol will enforce that non-aggregators will have to 'slow down' to the speed of the aggregator processes.

*2) One-sided Communication:* One sided communication, also known as Remote Memory Access (RMA), is a communication model added to the MPI specification starting from version 2. This model does not impose a synchronization of sender and receiver during communication: only one process is required for the data transfer, by either `Put`-ting or `Get`-ting data from the remote process. Even though the target process does not contribute to the communication operation in itself, it has to define and expose a region of its main memory for the operation, a so-called *window*.

One-sided communication is often considered faster and more light-weight compared to two-sided communication, due to the fact that there is no message matching required on the receiver side, and there is no unexpected message queue that needs to be parsed for every receive operation.

Within the scope of this work, which focuses on overlapping internal cycles of the two-phase collective I/O operation, we allocate two separate windows analogous to the two collective sub-buffers, using `MPI_Win_allocate`, the size of the windows being the size of the sub-buffers for aggregators and zero for non-aggregators.

One-sided operations also include methods that allow a process to control when to grant access to a memory window, or more generally speaking, a synchronization method between the processes. MPI provides two synchronization models: active-target and passive-target synchronization. Implementations have been developed for both methods.

*a) Active-target RMA:* Active-target RMA is a collective synchronization model. The simplest version of this synchronization method requires a call to `MPI_Win_fence` to start and end an exposure epoch. When closing an exposure epoch, the standard requires that all outstanding RMA operations on that window have completed.

In our implementation, an `MPI_Win_fence` function is used at the start of the `shuffle_init` operation, and a second one whenever we need to ensure the completion of the data transfer. This provides both, the origin and target processes (the aggregators), the information required to continue the next step and/or cycle in the algorithm. However, `MPI_Win_fence` is known to be an expensive operation.

*b) Passive-target RMA:* Passive-target RMA provides more flexibility than active-target RMA, since the target process is not directly involved in the synchronization operation itself. In this model, the origin process has to acquire a lock on the remote window of the target process using `MPI_Win_lock`, execute its RMA operations, and release the lock using `MPI_Win_unlock`. The completion of the RMA operation is guaranteed for that particular window in the origin side. However, the target process does not know when the data transfer operation to his local buffer has finished.

There are two challenges using this model in the two-phase I/O algorithm. First, the MPI specification offers a choice between two lock types: `MPI_LOCK_SHARED` or `MPI_LOCK_EXCLUSIVE`. The second option only allows one process to write into a window at a time, which will serialize the shuffle phase and thus harm the performance. `MPI_LOCK_SHARED` allows concurrent access to a window, and is usually used for read operations. However, since we can guarantee that different process will not overwrite each others data during the shuffle phase, we decided to use this version in our code.

The second issue concerns the target processes. On one hand, aggregator processes need to know when all data-transfer operations on a sub-buffer have finished in order to initiate the I/O operations. On the other hand, the origin processes must not execute any `MPI_Put` operation on any sub-buffer before the aggregator has not finished writing its content to file. To meet these two requirements, `MPI_Barrier` synchronize had to be introduced to ensure correct semantics of the operation.

## IV. Performance Evaluation

In the following section we evaluate the different overlap methodologies introduced in section III as well as the different data transfer primitives described in section III-B.

For our tests we used two platforms: the *crill* cluster at University of Houston, and the *Ibex* cluster at the KAUST Supercomputing Laboratory.

On the *crill* cluster we used a partition consisting of 16 nodes with four 2.2 GHz 12-core AMD Opteron processor (48 cores per node, 768 cores total) and 64 GB memory per node. The cluster uses a QDR InfiniBand network interconnect, using UCX v 1.6.1 as the underlying communication library for the MPI library. A BeeGFS parallel file system v7.0 distributed over all 16 nodes has been used for our tests, with a stripe size of 1MB.

The *ibex* cluster is a heterogeneous cluster with different families of AMD and Intel CPUs. In our tests we used the Skylake CPU family partition consisting of 108 nodes with 2.6 GHz 40-core Intel Xeon Gold 6148 Processor and 376 GB memory per node. Similarly to crill, the cluster also uses a QDR InfiniBand network interconnect using UCX v 1.6.1 as the underlying communication library. The experiments were performed under a BeeGFS filesystem consisting of 3.6 PB of storage on which we set 16 storage targets and a stripe size of 1MB.

All of the algorithms and versions described in the previous sections have been implemented using the ompio [6] parallel I/O framework in Open MPI [9], by modifying the existing *vulcan* fcoll component. For all subsequent tests, the default ompio parameters and settings have been used, e.g. a collective buffer size of 32MB and the automatic runtime aggregator selection algorithm [5].

In order to evaluate the different approaches, three different benchmarks have been used.

1) **IOR**: The Interleaved or Random (IOR)[19] benchmark is a synthetic parallel I/O benchmark used for testing performance of parallel file systems using different access patterns through different interfaces, e.g. POSIX I/O and MPI-I/O. It has several high-level parameters that can be used to define the I/O pattern (e.g. transfer size, block size, and segment count). For our tests, we mimicked a 1-D data distribution among processes by setting the transfer size and the block size both to 1GB and the segment count to 1. Tests have been executed for 10 different process counts between up to 704 processes, creating files that range between 16 and 704 GB in size.
2) **Tile I/O**: MPI-TILE-IO[17] is a synthetic benchmark to test parallel I/O operations on a two dimensional dense dataset. In our measurements, the number of tiles is set according to the number of processes ensuring that each dimension is equal to the square root of number of processes. Our tests used two different configuration: a tile size of 256 bytes with $2048 \times 1024$ elements per process, and tile size of 1 MByte and $32 \times 16$ elements per process. The Tile I/O benchmark was executed for process counts ranging from 16 to up to 729 processes.
3) **Flash I/O**: The FLASH I/O benchmark [1] suite is an extracted I/O kernel from the FLASH application. The benchmark is based on a lock-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes in astrophysics. The benchmark produces a checkpoint file, a plotfile with centered data, and a plotfile with corner data. We focus in our study on the checkpoint file, since it is the largest of three output files generated. Tests have been executed for various configuration using between 16 and 704 processes.

For each benchmark test case, we run between 3 and 9 measurements using each of the previously described algorithms and for each process count. For the summary statistics as shown e.g. in table I, all measurements series were used. When comparing individual data points (e.g. in fig. 1, we used the minimum execution time across all measurements within a series for that benchmark, process count and algorithm.

## *A. Overlap Methodology*

First, we evaluate the performance of the different overlap algorithmss described in section III. For this, we ran the benchmarks described above with all four overlap algorithms, as well as a version of the two-phase collective I/O algorithm that does not overlap shuffle and file I/O access phases at all. Table I presents for each overlap algorithm the total number of test runs in which the corresponding algorithm provided the best performance, across all benchmarks, platforms, process counts, and problem sizes.

The foremost conclusion of table I is that there is no clear winner or best approach across all test cases and platforms. In fact, even the version of the code not performing any overlap between the shuffle and the file I/O phase – which was originally only included to provide a base-line number – lead in 59 out of the 352 test series executed as part of this study with the lowest execution time, i.e. in approx. 16% of the test cases. The results however also indicate, that in 251 out of the 352 test series (71%) an overlap algorithm that used asynchronous write operations outperformed other approaches, showing a clear overall benefit of using asynchronous file I/O in the file access phase on both clusters.

Although the technical specifications of the clusters seem very similar, the overall performance numbers obtained are very different. First, despite of the fact that both clusters use a QDR InfiniBand network interconnect, the maximum bandwidth between two nodes on the Ibex cluster was higher than on the crill cluster, due to the slightly older AMD processors (Magny Cours) used on crill (approx. $3,400$ MB/s vs. $2,600$ MB/s). Furthermore, the BeeGFS storage on Ibex provided significantly higher write bandwidth compared to the BeeGFS file system on crill. The parallel file system on crill is based on using two additional hard drives in each of the 16 compute nodes, while Ibex uses a large scale parallel storage system. However, the crill cluster has been used in a dedicated mode for these measurements, resulting in significantly less variance for the data obtained, while the Ibex cluster was shared with other users during the tests, resulting in larger performance variations.

Figure 1 shows the differences obtained with the two clusters for the Tile I/O benchmark using 1M tile size for 256 and 576 processes.

The crill cluster shows no significant performance benefit for using any overlap algorithm for 256 processes, and approx. 6% performance improvement by overlapping shuffle and I/O phase for 576 processes. The Ibex cluster other hand showed performance improvements in both cases, approx. 34% for 256 processes, and 17% for 576 processes. A detailed analysis

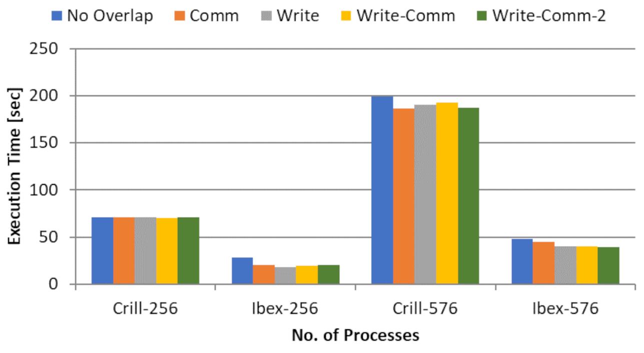

Fig. 1. Execution time of the Tile I/O benchmarks for 1M tile size using 256 and 576 processes.

of communication time and I/O time was performed using the no-overlap version of the code, in order to get a break down of how much time is spend in the shuffle vs. the file I/O access phase. For the 576 process test case, the collective I/O operation spends 93% of its time in file I/O access phase, and only approx. 7% in I/O operation on the crill cluster, while on Ibex it spends approx. 23% of the overall time in communication, offering therefore a much larger window for improvements. Thus, despite of the fact that nearly the entire communication time could be hidden behind I/O operations for this test case on crill, it resulted only in a limited 6% performance improvement overall on this platform, and a significantly larger improvement on Ibex.

This behavior is also confirmed by analyzing the average improvement obtained with each overlap algorithm and benchmark case for the crill 2 and the Ibex cluster 3. The average values shown in these graphs are determined by calculating the relative improvement in the execution using an overlap algorithm over the *no overlap* version, excluding however data points in which *no overlap* was faster than the overlap version (i.e. negative improvements). The values therefore represent the average improvement per overlap algorithm and benchmark **if** a performance improvement over the *no overlap* version was observed.

The average improvement on the crill cluster was between 3.7% and 9.2%, with overlap algorithms using an asynchronous write operation outperforming the communication overlap version in all instances. The same holds for the Ibex cluster, the average improvement was however higher, ranging between 8.6% and up to 22.3%.

## *B. Data Transfer Primitives*

In the second part of the evaluation, we focus on the data transfer primitives used by the two-phase I/O implementation. As discussed in section III-B, different implementations of the shuffle step have been developed based on the *Write-Communication-2* overlap algorithm, using non-blocking two-sided communication, one-sided communication using `MPI_Put` and `MPI_Win_fence` for synchro-

TABLE I
NUMBER OF RUNS AN OVERLAP ALGORITHM RESULTED IN BEST PERFORMANCE

| Benchmark | No Overlap | Comm Overlap | Write Overlap | Write-Comm Overlap | Write-Comm 2 Overlap |
|---|---|---|---|---|---|
| IOR | 21 | 11 | 32 | 28 | 15 |
| Tile I/O 256 | 17 | 13 | 18 | 31 | 26 |
| Tile I/O 1M | 10 | 6 | 18 | 20 | 17 |
| Flash I/O | 11 | 12 | 11 | 16 | 19 |
| Total: | 59 | 42 | 79 | 95 | 77 |

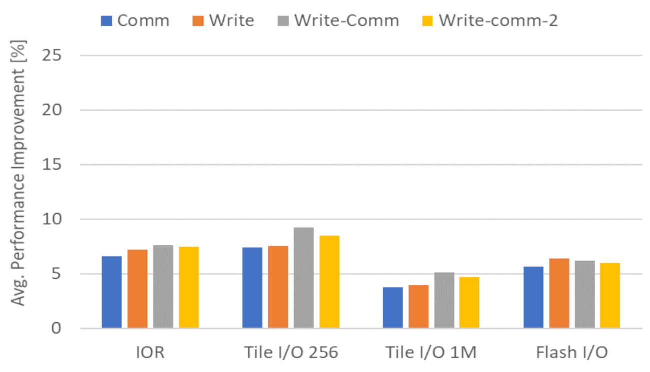

Fig. 2. Average relative performance improvement on the crill cluster for each overlap algorithm and benchmark.

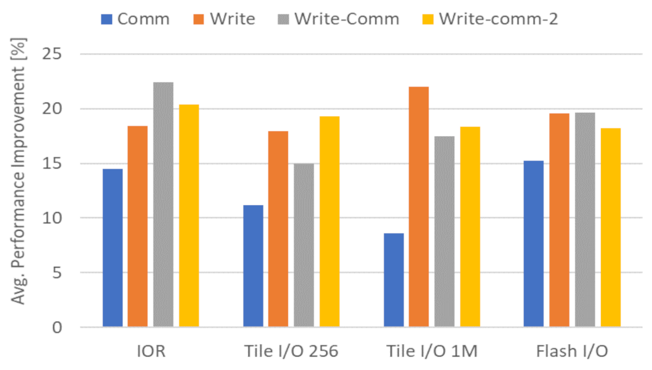

Fig. 3. Average relative performance improvement on the ibex cluster for each overlap algorithm and benchmark.

nization, and one-sided communication using `MPI_Put` and `MPI_Win_lock/unlock`. Tests have been executed on the crill as well as the Ibex cluster using the IOR, and the Tile I/O benchmarks for both 256 and 1M tile sizes.

Fig. 4 summarizes the results of the analysis. The graph shows for each benchmark test case the number of times a particular implementation resulted in the best performance overall. The results indicate that in the overwhelming number of test-cases (75%) two-sided data communication resulted in the best performance, and outperformed either of the two versions using one-sided communication. The results where consistent across both clusters. Ultimately, the synchronization costs introduced by the `MPI_Win_fence` and/or `MPI_Barrier` required for the one-sided operations to ensure correct output data out-weight the performance benefits of an `MPI_Put` operation over `Isend/Irecv` communication.

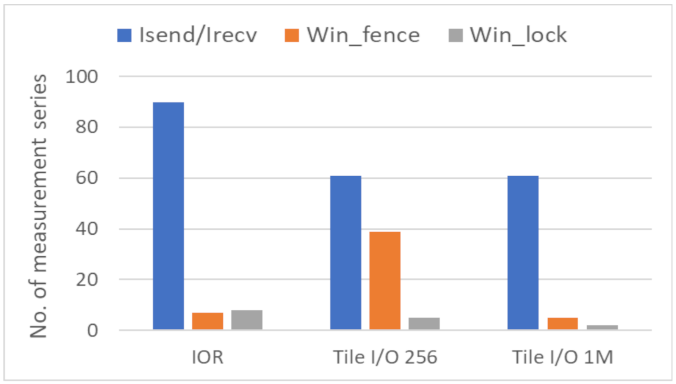

Fig. 4. Summary statistics comparing the number of times each of the three different data transfer primitives resulted in the best performance for each benchmark.

The only notable deviation of the general trend is for the Tile I/O benchmark when using the small 256 Byte tiles. In this scenario, the one-sided communication using `MPI_Win_fence` for synchronization achieved in approx. 37% of the test cases the best performance. On average, the performance gain over two-sided communication was approx. 27% on crill, and 30% on the Ibex cluster in these scenarios. It will require some further analysis to fully understand reasons for the difference in the performance behavior between this benchmark case and the IOR respectivel the Tile I/O benchmark for 1MB tile sizes. It should be noted however that both of the other two benchmarks operate on significantly larger, contiguous memory regions than the Tile I/O 256 benchmark does, which contains many, smaller, dis-contiguous data elements.

There was furthermore another interesting trend in this analysis. The benefits of using one-sided communication increased for larger process counts on the crill cluster. Only one measurement series out of 80 test cases showed benefits when using one-sided communication for tests using less than 256 processes on crill. For tests using 256 processes or more, 35 out of 84 test cases lead better performance when using one-sided communication over the two-sided counterpart.

## V. CONCLUSIONS

In this paper, we presented and evaluated various design options for overlapping two internal cycles of the two-phase

I/O algorithm. The paper further explored using different data transfer primitives for the shuffle phase, namely non-blocking two-sided communication, one-sided communication using active-target synchronization, and one-sided communication using passive-target synchronization. Our results indicate that overlap algorithms incorporating asynchronous I/O operations outperform overlapping approaches that only rely on non-blocking communication, and offer significant performance benefits of up to 22% compared to two-phase I/O algorithm not overlapping any internal operations. In the vast majority of the testcases however, using one-sided communication for the shuffle step did not lead to performance improvements compared to two-sided communication.

This work can be extended in multiple directions. Similar tests to the ones presented in this paper can be performed with more benchmarks, larger process counts, using different network interconnects, and parallel file systems. Some preliminary tests performed by the authors for example on a Lustre parallel file system showed very different results compared to the ones obtained on the BeeGFS file systems used in this study, due to significant performance problems of the `aio_write` operations on Lustre.

**Acknowledgments.** Partial support for this work was provided by the National Science Foundation under Award No. SI2-SSI 1663887. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors would also like to thank KAUST Supercomputing laboratory for providing compute time on the Ibex cluster for this project.

## References

[1] FLASH user guide. http://flash.uchicago.edu/website/.

[2] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[3] Javier Garcia Blas, Florin Isaila, David E Singh, and Jesus Carretero. View-based collective i/o for mpi-io. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 409–416. IEEE, 2008.

[4] Mohamad Chaarawi, Suneet Chandok, and Edgar Gabriel. Performance Evaluation of Collective Write Algorithms in MPI I/O. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 5544, pages 185 – 194, Baton Rouge, USA, 2009.

[5] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster 2011 conference*, page t.b.d, Austin, Texas, USA, 2011.

[6] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. OMPIO: A Modular Software Architecture for MPI I/O. In *in Y. Cotronis, A. Danalis, D. S. Nikolopoulus, J. Dongarra (Eds.) 'Recent Advances in the Message Passing Interface', Lecture Notes in Computer Science, vol. 6960*, pages 81–90. Springer, 2011.

[7] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 58–58. IEEE, 2005.

[8] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 04 2008.

[11] Wei-keng Liao and Alok Choudhary. Dynamically Adapting File Domain Partioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In *Proceedings of the Supercomputing Conference*, 2008.

[12] Weifeng Liu, Jie Zhou, and Meng Guo. Topology-aware strategy for mpi-io operations in clusters. *Journal of Optimization*, 2018, 2018.

[13] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. In *Proceedings 16th International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2001.

[14] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2003.

[15] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard Version 3.1*, June 2015. http://www.mpi-forum.org.

[16] Bill Nitzberg and Virginia Lo. Collective buffering: Improving parallel I/O performance. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*, pages 148–157. IEEE, 1997.

[17] R. Ross. *Parallel I/O Benchmarking Consortium*. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/.

[18] Saba Sehrish, Seung Woo Son, Wei-keng Liao, Alok Choudhary, and Karen Schuchardt. Improving collective I/O performance by pipelining request aggregation and file access. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 37–42. ACM, 2013.

[19] Hongzhang Shan and John Shalf. Using IOR to analyze the I/O performance for HPC platforms. 2007.

[20] François Tessier, Venkatram Vishwanath, and Emmanuel Jeannot. Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 70–80. IEEE, 2017.

[21] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[23] Yuichi Tsujita, Atsushi Hori, Toyohisa Kameyama, Atsuya Uno, Fumiyoshi Shoji, and Yutaka Ishikawa. Improving collective mpi-io using topology-aware stepwise data aggregation with i/o throttling. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 12–23, 2018.

[24] Yuichi Tsujita, Kazumi Yoshinaga, Atsushi Hori, Mikiko Sato, Mitaro Namiki, and Yutaka Ishikawa. Multithreaded two-phase i/o: Improving collective mpi-io performance on a lustre file system. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 232–235. IEEE, 2014.

[25] Zhixiang Wang, Xuanhua Shi, Hai Jin, Song Wu, and Yong Chen. Iteration based collective i/o strategy for parallel i/o systems. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 287–294. IEEE, 2014.

[26] Xuechen Zhang, Song Jiang, and Kei Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.