MPI-IO Local Aggregation As Collective Buffering for NVMe Lustre Storage Targets

Michael Moore
HPC, AI & Labs
HPE
Austin, USA
michael.moore@hpe.com

Ashwin Reghunandanan
HPC, AI & Labs
HPE
Bangalore, India
ashwin.reghunandanan@hpe.com

Lisa Gerhardt
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
lgerhardt@lbl.gov

***Abstract*— HPC I/O workloads using shared file access on distributed file systems such as Lustre have historically achieved lower performance relative to an optimal file-per-process workload. Optimizations at different levels of the application and file system stacks have alleviated many of the performance limitations for disk-based Lustre storage targets (OSTs). While many of the shared file optimizations in Lustre and MPI-IO provide performance benefits on NVMe based OSTs the existing optimizations don't allow full utilization of the high throughput and random-access performance characteristics of the NVMe OSTs on existing systems. A new optimization in HPE Cray MPI, part of the HPE Cray Programming Environment, builds on existing shared file optimizations and the performance characteristics of NVMe-backed OSTs to improve shared file write performance for those targets. This paper discusses the motivation and implementation of that new shared file write optimization, MPI-IO Local Aggregation as Collective Buffering, for NVMe based Lustre OSTs like those in the HPE Cray ClusterStor E1000 storage system. This paper describes the new feature and how to evaluate application MPI-IO collective operation performance through HPE Cray MPI MPI-IO statistics. Finally, results of benchmarks using the new collective MPI-IO write optimization are presented.**

Keywords—Shared File, Performance, Collective MPI-IO, Lustre

I. Introduction and Background

The importance of shared file write workloads, also known as N:1, in HPC applications is evident in the long and continued history of enhancements and APIs developed to support efficient access to a single file across large MPI applications. Characterization of computational science applications indicate a shared or partially shared workload is an important workload in large HPC centers [1]. While shared file performance, especially writes, has long been a challenging workload the more recent adoption of NVMe storage has exposed new performance limitations in N:1 write workloads. We explore the N:1 shared file write performance limitations in the context of the HPE Cray ClusterStor E1000 NVMe-based Lustre storage targets (OSTs) and introduce a new optimization in HPE Cray MPICH for collective MPI-IO writes. Before describing the specifics of the optimization, we summarize the existing Lustre N:1 write optimizations accessible through the standard POSIX interfaces and Lustre library and then describe the higher level MPI-IO optimizations available in HPE Cray MPICH for collective MPI-IO writes.

A. Lustre optimizations

The lowest layer of the I/O software stack most applications interact with is the POSIX API and this is the layer where file system specific optimizations are accessible. As with many distributed, parallel file system supporting POSIX semantics Lustre uses a distributed locking scheme to ensure concurrent write accesses to a single file maintain consistency. The main obstacle to achieving shared file write performance similar to an optimal write workload is lock contention from distributed locking. The Lustre-specific shared file write optimizations focus on addressing this bottleneck.

1) Lustre Group Locks

Lustre group locks provide a mechanism for an application to instruct the file system to no longer use distributed write locking through the LDLM (Lustre Distributed Lock Manager) [2]. Instead of the default locking behavior a special Lustre API call is made to take a lock known as a group lock. The group lock communicates to the file system that it does not need to provide distributed locking for the specified file. Instead, the responsibility for ensuring consistency is handled by the application. This serves as an optimization for shared file writes by removing distributed lock contention allowing for a scalable solution even when large numbers of clients are writing to the same file and OST. Higher level I/O libraries, such as MPI-IO, make used of this optimization internally and expose the feature through the use of the MPI-IO hints mechanism. This allows an application to take advantage of the feature without any application modifications specific to the Lustre file system. For an application not using collective MPI-IO application code modifications using the Lustre library are required.

2) *Lustre Lockahead*

Lustre Lockahead is another optimization which alters the default locking behavior of Lustre [3]. The default LDLM behavior is optimized for a file-per-process (N:N) workload and when acquiring an LDLM lock, the byte range of the lock defaults to extending to the end of the file. While optimal for file-per-process write workloads this behavior creates unnecessary lock conflicts for a shared write workload. For an application using collective MPI-IO and collective buffering the MPI library determines which MPI ranks will function as aggregators and which offsets in the file each of thoe aggregators will be writing. Knowing the offsets, the MPI library can request locks for the specific byte ranges each aggregator will need to write. Then, when it's time to write the data the aggregator has already acquired the necessary locks to perform the writes. This optimization avoids creating lock contention from false sharing and allows the lock acquisition to occur asynchronously from the writes. As with Lustre group locks, Lustre Lockahead is supported in HPE Cray MPICH and does not require any application modifications. Lustre Lockahead and Lustre group locks are different solutions to the same problem and are not used together at the MPI application layer. The need for the alternate locking approach relates to higher-level I/O library behavior and is briefly described in the discussion of HDF5 with collective MPI-IO.

3) *Lustre Overstriping*

Lustre Overstriping is the last and most general shared file write optimization discussed. Lustre Overstriping allows a single file to allocate multiple stripe objects on the same Lustre OST. Without the use of Lustre Overstriping a single OST can only hold a single stripe per file. As discussed in the Lustre Lockahead feature, a Lustre client takes a lock on a given OST, more specifically, on the specific object identifier located on that OST associated with the file the client is accessing. By allowing multiple, unique stripe objects to exist on a single OST the number of locks that can be held by different clients at any one time is increased. More locks allow for more parallelism writing to the same underlying OST which improves performance. Using Lustre Overstriping requires no higher level libraries or code modifications – the striping of the shared file is all that needs to be specified which can be done using the Lustre `lfs` utility. This is the one Lustre shared file write optimization that can be used for any shared file write workload using POSIX or MPI-IO (independent or collective operations).

B. *Collective MPI-IO Optimizations*

Collective MPI-IO is one of the most popular interfaces for shared file access due to the API and available optimizations such as collective buffering. While describing the use of collective MPI-IO is outside the scope of the paper we focus on collective MPI-IO optimizations and analysis of the performance of those optimizations for shared file write workloads. We also discuss how optimizations in additional layers above MPI-IO, such as HDF5, and below MPI-IO, specifically Lustre, interact with the discussed collective MPI-IO optimizations.

Users of MPI-IO are likely familiar with the MPI-IO hint mechanism. MPI-IO hints allow the application to request MPI-IO to use specific features, file layouts, aggregator layouts, and other tunable parameters based on file name matching. The variable used in HPE Cray MPICH to specify MPI-IO hints is `MPICH_MPIIO_HINTS`. Specific examples will be provided in the paper and additional information is available in the HPE Cray MPICH `mpi` man page.

1) *Collective Buffering and Aggregators*

Collective buffering is an optimization available with collective MPI-IO that allows a subset of MPI ranks to perform the I/O to the underlying filesystem on behalf of all ranks in the collective MPI-IO call – they aggregate the data from many ranks into one or more write calls they perform. Historically, this two-phase optimization was largely used to allow smaller requests to be "aggregated" into larger, contiguous requests which tend to be higher performing. This two-phase I/O strategy of collective buffering is advantageous for many reasons, three relevant ones for this discussion are:

- Generating less write lock contention by having fewer file system clients accessing the file. By default in HPE Cray MPICH, one client per OST access the file instead of *N* (where *N* is less than or equal to the number of nodes in the job).
- The MPI implementation controls how data is allocated to aggregators. By default, for Lustre file systems, HPE Cray MPICH aligns an aggregator to write on Lustre stripe boundaries and to a single OST although more than one aggregator can be configured to write to a single Lustre stripe.
- Aggregator MPI ranks make larger, contiguous requests which minimizes seeking on the underlying OST devices for disk-based OSTs.

Although the default configuration in HPE Cray MPICH assigns a single aggregator to each OST, which was sufficient for previous storage with peak OST rates of 1 – 2 GB/s, most optimal collective buffering settings now should be assigning multiple aggregators per OST via the `cray_cb_nodes_multiplier` MPI-IO hint. Using multiple aggregators to write to a single OST is typically only advantageous in conjunction with non-default lock optimizations via the `cray_cb_write_lock_mode` MPI-IO hint. Although, as discussed, Lustre Overstriping can also help alleviate lock contention. In the context of collective buffering, aggregator placement counts each Lustre Overstripe as an OST and each is allocated aggregators. For a concrete example, using `cray_cb_nodes_multiplier=8` with a singly striped file uses an equivalent number of aggregators as a file with 8 Overstripes which uses the default of one aggregator per Lustre stripe. While the default of one aggregator per OST eliminates Lustre LDLM lock contention some potential performance is not realized. Due to increased drives per RAID device and higher performing NVMe devices the performance of a single process is only able to achieve a fraction of the peak performance of an HPE Cray ClusterStor E1000 disk or NVMe OST. The default of a single aggregator per OST will provide consistent performance across a range of OST types but it's recommended to use a non-default locking mechanism and more than 1 aggregator per OST for current HPE Cray ClusterStor E1000 NVMe and disk based OSTs.

Finally, as aggregator counts, Lustre stripe sizes, and performance are considered, the size of a collective write should also be kept in mind. The size, in terms of bytes, of a collective write is the number of ranks in the MPI communicator multiplied by the amount of data each rank is writing (count of elements multiplied by size of elements). With a file striped across *O* OSTs, assigning *N* aggregators per OST, and a Lustre stripe size of *M* MiBs, *N**O*M* MiB of data must be written by the collective write call to utilize all aggregators and perform full stripe writes. Take the specific example of 1,024 MPI ranks doing a collective write call with each rank writing 1MiB of data – 64 elements of 16,384 bytes. The collective write call would write a total 1 GiB of data. If the shared file was striped across 16 OSTs (*O*) with a stripe size of 16 MiB (*M*) and 8 aggregator ranks were assigned per OST (*N*) the 1 GiB of data of the collective write call would only use half of the aggregators. The collective write call is 1 GiB of data but the specified collective buffering parameters and Lustre striping cover 2 GiB of data. In this case only 4 aggregators per OST would be used. Ways to confirm all aggregators are being used are discussed in Section IV.

2) Collective Buffering Optimizations using Lustre Locking in HPE Cray MPICH

Collective Buffering, as previously described, optimizes the data write path for collective MPI-IO writes by using a subset of MPI ranks to perform file system writes on behalf of other nodes. Current ClusterStor E1000 OSTs require more than one aggregator to achieve peak performance and with multiple aggregators, a non-default lock optimization is required to realize potential write performance. HPE Cray MPICH supports four different locking configurations, three previously supported and the new mode described in this paper. Those are:

- Default Lustre locking for independent MPI-IO, collective MPI-IO, and collective MPI-IO with collective buffering
- Lustre group locks for collective MPI-IO with collective buffering and no independent MPI-IO
- Lustre Lockahead for collective MPI-IO with collective buffering and independent MPI-IO calls
- Lustre group locks for collective MPI-IO with local aggregation as collective buffering and no independent MPI-IO

3) HDF5

While the capabilities and features of HDF5 are not limited to collective MPI-IO, we limit the discussion in this paper to HDF5 using collective MPI-IO. In HDF5 versions prior to 1.10.0 HDF5 file metadata updates were performed through independent MPI-IO write operations [5]. This prevented using optimizations that require only collective operations be performed on the open file – specifically Lustre group locks could not be used since no independent MPI-IO operations are allowed on an open file. Since HDF5 version 1.10.0, the ability to use collective writes for metadata operations has been available. This allows using Lustre group locks with collective buffering for HDF5 writes in addition to Lustre Lockahead which was developed for that type of mixed independent and collective MPI-IO use case.

With several optimizations that are relevant to shared file workloads, Table 1 provides a synopsis of which APIs and workloads can make use of which optimizations. Overstriping can be used in conjunction with or without other lock optimizations. Lustre Lockahead and Lustre group locks are not used at the same time from the MPI application perspective.

API	Optimization
POSIX	Overstriping, Lockahead*, group locks*
Independent MPI-IO	Overstriping
Collective MPI-IO	Overstriping, coll. buff. with Lockahead, coll. buff with group locks, Local Aggregation as coll. buff.
HDF5, no coll. metadata	Overstriping, coll. buffering with Lockahead
HDF5, coll. meta	Overstriping, coll. buff. with Lockahead, coll. buff with group locks, Local Aggregation as coll. buff.

* requires application modification

Table 1. API and Optimization Mapping

II. Motivation

As the performance of individual OSTs has continued increasing the optimal collective buffering parameters and the effectiveness of collective buffering to achieve peak shared file writes continues to change. Currently, HPE Cray ClusterStor E1000 performance approaches 30 GB/s write throughput for an NVMe-based storage target and 8 GB/s write throughput for a disk-based storage target under optimal I/O workload and conditions. Specific results are detailed in Figure 1 but the typical buffered write rate of a single process is around 1.5 GB/s. For a single NVMe based OST a minimum of 20 aggregator ranks would be required; in practice more nodes are required. For typical HPC systems this creates a requirement for nearly full system size jobs, or more nodes than are available, to achieve optimal performance for shared file workloads using this method. Enabling higher performing shared file workloads at lower node counts was a key factor driving this investigation.

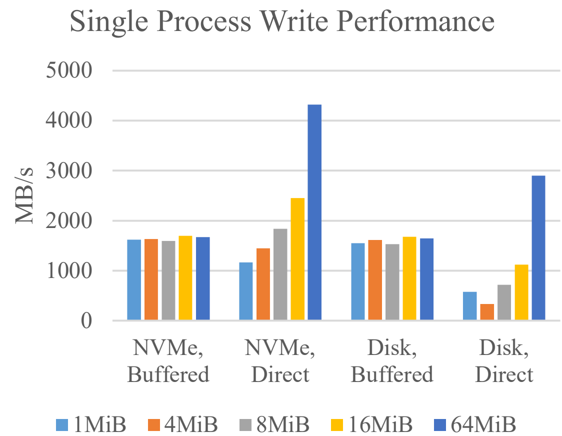

Figure 1. Single Process Write Performance

The single process limitation for buffered I/O to a single, shared file is not only a single process limitation but the same limit is present at the node level as Figure 2. Discussion of this issue is outside the scope of the paper but this limitation prevents additional aggregators on a node, when using buffered I/O, from providing improved performance. While using direct I/O from aggregators is possible it generally requires more processes submitting I/O to achieve optimal performance since the direct I/O requests only return when data is on the OST instead of when it's been copied to the node's page cache. A factor motivating this solution is to be able to use as many ranks, to submit I/Os, as possible – which is the total number of ranks participating in the collective call. Previously this approach wasn't feasible due to the seeking caused by such a large number of write requests with offsets across many gigabytes of data but with NVMe-based OSTs that is no longer a concern.

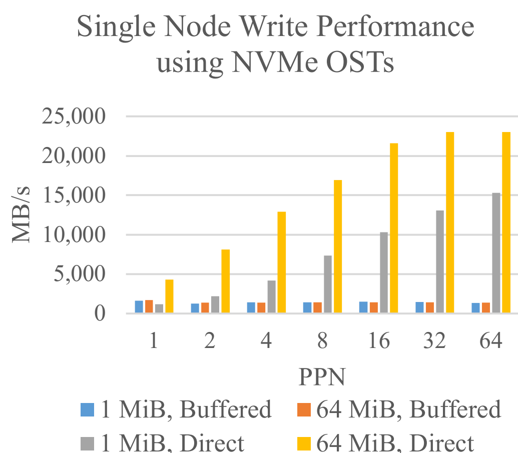

Figure 2. Single Node Write Performance

Regardless of using a single process per node or many processes the use of lock optimizations, Lustre group lock or Lustre Lockahead, is necessary to achieve optimal performance. Given this set of observations, a new feature was added to HPE Cray MPICH for collective MPI-IO, Local Aggregation as Collective Buffering, which allows each node and each rank to perform its own I/O and make use of Lustre group locks for optimized locking. In the next section we'll discuss the implementation, how to use the feature, and evaluate synthetic benchmarks covering baseline shared file performance benchmarks and evaluation of the new Local Aggregation as Collective Buffering feature

III. Implementation and Synthetic Benchmarks

A. *Implementation and Usage*

The implementation of Local Aggregation as Collective Buffering uses a new MPI-IO hint variable to distinguish it from the existing locking modes provided when collective buffering is enabled. The tunable, named `cray_nocb_write_lock_mode`, currently only supports the value 1, denoting the use of Lustre group locks. The tunable is validated for permissible values consistent with the other tunables that it depends on. In order to enable Local Aggregation as Collective Buffering, collective buffering needs to be disabled, the flag set to indicate that no independent MPI-IO requests will occur, and data sieving on writes disabled.

The total set of hints needed to use the new feature are:

```
MPICH_MPIIO_HINTS="*:romio_cb_write=disable:cray_nocb_write_lock_mode=1:romio_cb_read=disable:romio_ds_write=disable:romio_no_indep_rw=true"
```

When the above hint is provided, all ranks will do an open() as part of the MPI_File_open() collective call and also do the necessary steps to acquire a Lustre group lock on the shared file for writing. For each MPI-IO collective write call, a rank will directly write its own local data to the file. On file close the group lock is released.

This feature is still considered experimental. There were instances in synthetic benchmark testing of data validation issues. While the feature exists in shipped versions it is not enabled by default and is not documented; once the feature is documented in the HPE Cray MPICH man page it will have been tested and vetted for correctness.

B. *Synthetic Benchmarks*

All tests presented were performed on a Cray EX system with several hundred compute nodes running COS 2.4 and Slingshot 2.0. An HPE Cray ClusterStor E1000 comprised of two E1000 MDUs, 3 E1000Fs, and 3 E1000D-2s with ClusterStor NEO software version 6.2 was used. The compute nodes were CPU-only nodes with dual socket Milan 2.45 GHz CPUs and single injection Cassini NICs. The Lustre file system is mounted using KFI LND (Lustre Network Driver). HPE Cray MPICH 8.1.25 was used to compile and run benchmarks along with HDF5 version 1.12.0, which is packaged in the Cray Programming Environment.

```
IOR -a MPIIO -C -c -E -e -g -k -b 1m -t
1m -s 512 -O verbose=3 -o
TESTDIR/IOR_POSIX -w
```

Listing 1. Example IOR invocation

Although other benchmarks were planned, issues requiring debugging only allowed for the use of IOR 3.3.0, the canonical MPI I/O benchmark, for synthetic benchmark measurements. Unless otherwise noted, these tests set the IOR transfer size equal to the block size and the segment count was specified to write adequate data for each test. The Lustre stripe size matched the IOR transfer and block size. In the case of tests using the POSIX interface any dirty data was flushed via an fsync call as part of the measured test time to avoid any client-side cache effects. For POSIX Lustre group lock tests IOR 3.3.0 was modified to use Lustre group locks through the Lustre API. The modified version of IOR was only used for POSIX tests with Lustre group locks. There was no writeback, writethrough or readback cache enabled on the OSSes. A sample IOR invocation used for testing using the MPI-IO collective interface, with a 1MB transfer and block size, and 512 segments per rank which

Identify applicable funding agency here. If none, delete this text box.

equates to 512MiB of data written per rank is documented in Listing 1. The aggregate file size was tuned to ensure adequate test duration based on MPI rank count although tests were relatively short to expedite data collection and minimize the amount of NVMe OST trimming that was required for optimal write performance. During testing we identified an issue with the HDF5 API support in IOR that the alignment of I/O requests did not honor the value of the '-J' argument. Without alignment set on at least page boundaries the I/O requests prevented using Direct I/O with HDF5 – since direct I/O requests must be page aligned in Lustre. We will further investigate this issue. A test listed as "Direct" refers to opening the file with the O_DIRECT flag, for POSIX this is done with an IOR argument and for MPI-IO and HDF5 API, this is enabled via the `direct_io=true` MPI-IO hint and not directly as an IOR argument.

All tests using Lustre group locks were impacted, to varying degrees, by a Lustre issue which causes very long times for Lustre Group lock unlock calls (`llapi_group_unlock`). The issue is tracked in LU-16046 [6]. The fix is not present in COS 2.4 but is planned for COS 2.6. The additional time spent in the Lustre group lock unlock dramatically reduced the reported throughput from the IOR benchmark since close time is included in the test duration. The minimal difference, and variability of, the extra file close timing effects both collective buffering with group locks and Local Aggregation as collective buffering similarly so comparative results are still meaningful.

The initial tests of a single node were previously detailed above in Figure 1 and 2. The experiments in Figure 1 demonstrate the best-case single process shared file performance, or more specifically, the best-case performance of a single MPI rank operating as a collective buffering aggregator. These tests use POSIX, instead of MPI-IO, because the MPI ranks serving as aggregators use the POSIX interface to write to the file system. The single node tests show the lack of performance scaling with additional ranks on a node using buffered I/O. As noted above, using multiple processes on a single node yields no performance improvement when using buffered I/O. The impact of the latency in direct I/O calls is evident by the relatively lower performance at smaller transfer sizes which matches expectations. However, at higher process counts and larger transfer sizes using direct I/O appears to provide a way to increase per-node performance when doing shared file writes.

Next, we evaluate the shared file write performance, still using the POSIX interface, to emulate how collective buffering aggregators would be writing. The single node tests used multiple OSTs since the single client was under test, we change to single OST tests when using multiple nodes to evaluate performance of the scalable unit of storage– an OST. Figure 3 shows measured shared file POSIX write performance using Lustre group locking and Figure 4 shows the improvement in throughput between default Lustre Locking and Lustre group locking. An obvious but still important observation from these results is that the transfer and Lustre stripe size matter, even when using default Lustre locking. Further, the observed benefits of direct IO at larger transfer sizes are still observed with many writers. However, direct I/O shows small to no improvements using Lustre group locking while buffered I/O, especially for NVMe OSTs, shows very large improvements, greater than 200%. The tests that appear to be not depicted reported a 0% or few percentage point reduction in performance and were not plotted for clarity and should be considered as equal in performance.

For the final single OST tests we report the performance

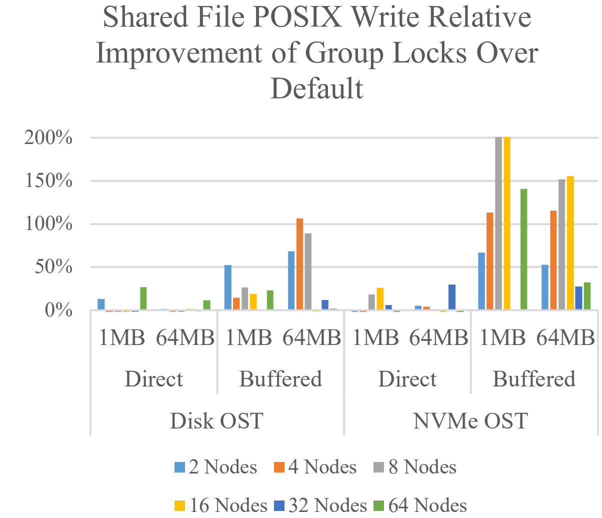

Figure 3. Single OST Group Locking Percentage Improvement

combining the benefits of Lustre group locks with Overstriping. Considering the use of a single process per rank and the additional overhead of Lustre group lock unlock with this Lustre client, these results demonstrate performance very near optimal E1000F OST performance.

The end goal of optimizing shared file workloads is to

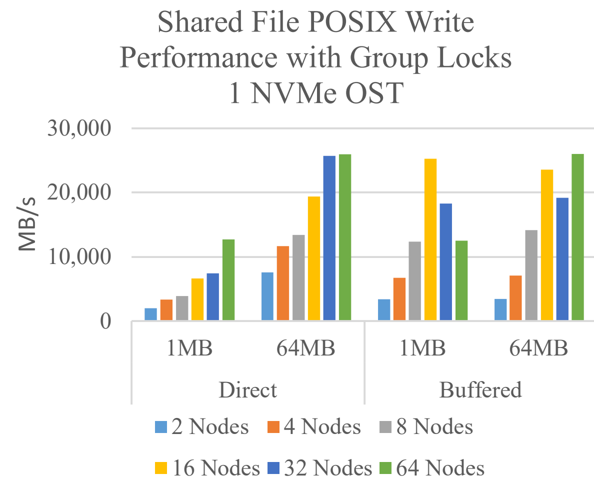

Figure 4. Single OST Write Performance with Group Locks

achieve close to the peak expected performance of an OST – which was demonstrated for some specific workloads in figure 5. As previously mentioned, this workload is very similar to that of collective buffering aggregators. Although we provide additional guidance later these results indicate for E1000 disk

OSTs using 4-8 aggregators for smaller stripe sizes or 8 or more aggregators for larger stripe sizes will achieve near peak performance. For NVMe OSTs, larger node counts (32 or 64 per OST) are needed. For both OST types, direct IO with a single rank per-node is only able to reach peak performance for large transfer sizes.

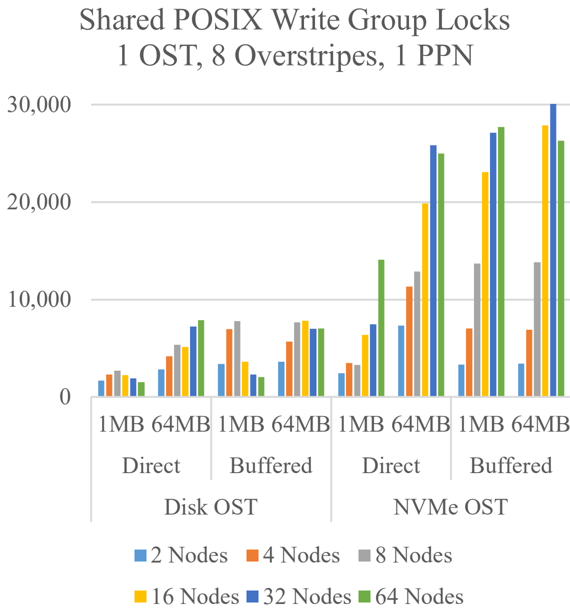

Figure 5. Single OST Group Locking, Overstriping

Finally, moving beyond the POSIX API we move to evaluating performance using HDF5 with collective metadata through IOR and compare the original collective buffering solution and the experimental Local Aggregation as collective buffering feature. The HDF5 tests use many ranks per node (32) to measure performance more realistically for an application. The collective buffering (CB) tests use lock mode 1 (Lustre group locks) and all nodes have one rank used as an aggregator

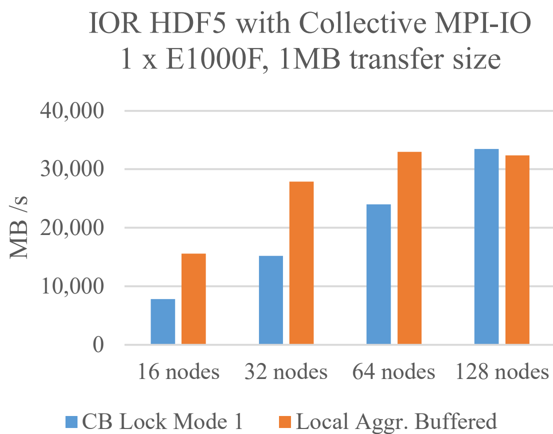

Figure 6. HDF5 Write Performance Comparison with Local Aggregation

e.g. the 64 node test uses 64 aggregators. Figure 6 compares the write performance to a shared HDF5 file using collective MPI-IO, including HDF5 collective metadata, with traditional collective buffering using Lustre group locks to Local Aggregation as collective buffering. The results indicate that, especially at smaller transfer sizes and smaller node counts, using Local Aggregation as collective buffering provides a significant performance improvement. Furthermore, given the limitation of single-node buffered performance the performance improvements between collective buffering and Local Aggregation using buffered I/O are solely from removing the collective buffering overhead. We expect there to be additional performance improvement once we're able to run similar tests use the MPI-IO hint `direct_io=true` successfully. Despite these encouraging results the overall measurements are well below expected or peak performance. As previously discussed, these are reported IOR results which include the large overhead of file close caused by LU-16046. Excluding the close time, there are significant throughput improvements relative to the results depicted in Figure 6. However, without the ability to run comparable Direct I/O tests there isn't a straightforward way to report bandwidth due to client-side buffering. To provide some idea of the time spent in file close relative to writing in these tests, Figure 7 shows the time split between write and close. File close accounts for at least 10% of total time and tends to impact Local Aggregation more severely, since all MPI ranks are taking a group.

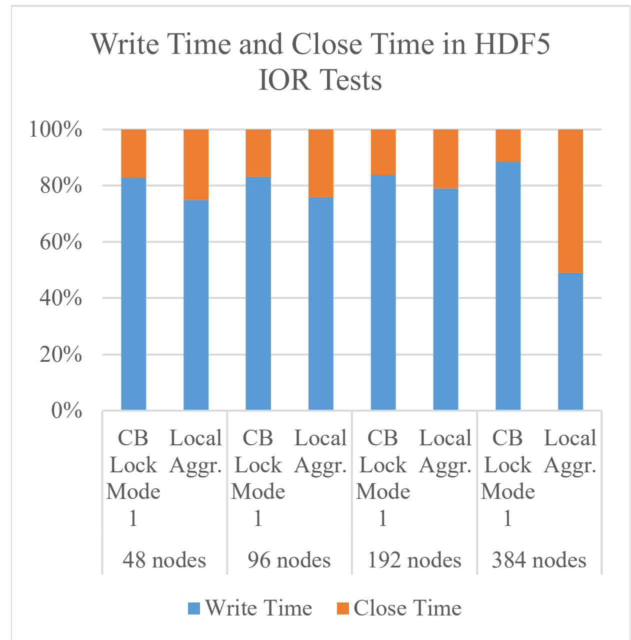

Figure 7. HDF5 Write and Close Time Comparison

Scaling up the testing from a single E1000F scalable unit to three E1000Fs Figure 8 shows the percent improvement of Local Aggregation relative to an optimal collective buffering with Lustre group locks test. Like the smaller scale tests, Local Aggregation shows significant performance improvements at lower node counts and smaller transfer sizes. As previously mentioned, we expect that tests using direct IO will provide an even more compelling case for the feature at larger transfer sizes and higher node counts. Having demonstrated existing collective MPI-IO performance using collective buffering and local aggregation we discuss analyzing your collective MPI-IO workload using statistics from HPE Cray MPICH in the next section.

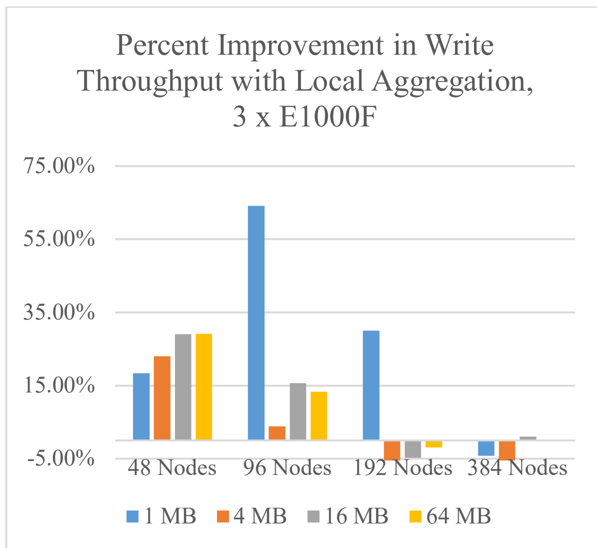

Figure 8. HDF Write Performance Improvement With Local Aggregation on 3 E1000F Configuration

IV. Analysizing Collective MPI-IO Workloads

HPE Cray MPICH provides several collective MPI-IO statistics and informational debugging options. Enabling these low overhead debugging options can provide insight into the characteristics of your collective MPI-IO, where time is spent during the collective calls, and help identify sources of imbalance. There are a set of 4 environment variables that will enable relevant debugging output (to stderr) provided in Listing 2.

```
export MPICH_MPIIO_HINTS_DISPLAY=1
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
export MPICH_MPIIO_STATS=1
export MPICH_MPIIO_TIMERS=1
```

Listing 2. Recommended Collective MPI-IO Information Environment Variables

A. MPI-IO Hints Display

This environment variable (`MPICH_MPIIO_HINTS_DISPLAY=1`) will report the MPI-IO hints applied when each file is opened. This is helpful to confirm which hints are being used. In the case of erroneous or conflicting hints, this identifies which hints are used. The example in Listing 3 reports the hints of a file opened with enabled collective buffering using Lustre Group Locks and 64 aggregators per OST on a file striped across 6 OSTs using a 16 MiB stripe size.

```
PE 0: MPIIO hints for
/lus/flash/mmoore/testdir.200/IOR_HDF5:
          romio_cb_pfr              = disable
          romio_cb_fr_types         = aar
          cb_align                  = 2
          cb_buffer_size            = 16777216
          romio_cb_fr_alignment     = 1
          romio_cb_ds_threshold     = 0
          romio_cb_alltoall         = automatic
          romio_cb_read             = enable
          romio_cb_write            = enable
          romio_no_indep_rw         = true
          romio_ds_write            = automatic
          ind_wr_buffer_size        = 524288
          romio_ds_read             = disable
          ind_rd_buffer_size        = 4194304
          direct_io                 = false
          striping_factor           = 6
          striping_unit             = 16777216
          romio_lustre_start_iodevice = -1
          aggregator_placement_stride = -1
          abort_on_rw_error         = disable
          cb_config_list            = *:*
          cray_cb_nodes_multiplier = 64
          cray_cb_write_lock_mode  = 1
          cray_fileoff_based_aggr  = false
          romio_filesystem_type     = CRAY ADIO:
          cb_nodes                  = 384
```

Listing 3. MPI-IO Hints Display Example

B. MPI-IO Aggregator Placement

This environment variable (`MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1`) reports the placement of aggregator ranks by MPI rank number and node name. This output is helpful to confirm aggregator count but also makes identifying which node a slow aggregator resides on easier. Details of the rank reordering and aggregator placement stride are available in the HPE Cray MPICH `mpi` man page. Abbreviated output showing only 6 aggregators out of the 384 below is provided in Listing 4.

```
Aggregator Placement for
/lus/flash/testdir.200/IOR_HDF5
  RankReorderMethod=1  AggPlacementStride=-1
  AGG    Rank      nid
  ----  ------   --------
     0       0   nid00000
     1      32   nid00001
     2      64   nid00002
     3      96   nid00003
     4     128   nid00004
     5     160   nid00005
     6     192   nid00006
...
```

Listing 2. MPI-IO Aggragator Placement Example

C. MPI-IO STATS

This environment variable (MPICH_MPIIO_STATS=1) variable provides a synopsis of several important characteristics of what MPI-IO operations were performed to a file. Specifically, the rank count, independent and collective writes, number of aggregators, striping information, if stripe-size (optimal) writes happened and how many aggregators were active. For performance considerations the number of stripe-sized writes relative to the total number of writes is important. In the case of HDF5, there will always be some number of smaller writes due to HDF5 metadata or the write size not being an exact multiple of the Lustre stripe size. The important items are that most writes are equal to the Lustre striped size and that a majority of system writes report all aggregators being active (e.g. the 384 aggregator "bucket" in the example below). See the "Collective Buffering and Aggregators" section for more discussion about collective write sizes, aggregator counts and Lustre stripe sizes. When comparing results between collective buffering enabled and local aggregation (collective buffering disabled) the "system writes" count will have a higher value in the case of local aggregation and will typically have lower "ave system write size" as well as "stripe sized writes". The exception to these observations is the case when the I/O block size matches or exceeds the Lustre stripe size. Finally, for the local aggregation case "number of write gaps" and "ave write gap size" counts will typically be higher than when collective buffering is enabled. If more in-depth analysis of the workload is of interest using a tool like Darshan is recommended [7].

```
+---------------------------------------------+
| MPIIO write access patterns for
/lus/flash/testdir.200/IOR_HDF5
|   ranks in communicator   = 12288
|   independent writes      = 0
|   collective writes       = 34
|   independent writers     = 0
|   aggregators             = 384
|   stripe count            = 6
|   stripe size             = 16777216
|   system writes           = 393268
|   stripe sized writes     = 393184
|   aggregators active      = \
|      24576,0,0,393216 (1, <= 192, > 192, 384)
|   total bytes for writes  = 6597069779872 \
|                           = 6291456 MiB \
|                           = 6144 GiB
|   ave system write size   = 16774997
|   read-modify-write count = 0
|   read-modify-write bytes = 0
|   number of write gaps    = 16
|   ave write gap size      = 1156428335687
+---------------------------------------------+
```

Listing 5. MPI-IO Stats Example Output

D. MPI-IO TIMERS

The last set of debugging information contains the most detail. A section of timer information for "all ranks" is reported for each file accessed with a series of timers shown below. In the case of collective buffering a second section is printed with statistics relevant only to aggregator ranks. When using Local Aggregation as collective buffering only the "all ranks" section is reported since collective buffering isn't used.

The example output is from a test using Local Aggregation and buffered I/O. All ranks are writing data and using buffered I/O. This output shows that a significant amount of time is being spent in file open and close. Unexpectedly, there is no time in the group unlock but instead was spent in close. This is likely related to LU-16046 since it appears to be a single rank taking a disproportionate amount of time but this will require further investigation. This testing also identified a bug where the 'close

```
+---------------------------------------------+
| MPIIO write by phases, all ranks,\
|  for /lus/flash/testdir.201/IOR_HDF5
|   number of ranks writing        = 12288
|   number of ranks not writing    =     0
|                               min  max  ave
|                               ---  ---  ---
|   open/trunc  time        = 4.89  4.89 4.89
|   close sys time          = 0.00  5.76 0.12
|   close fsync time        = 0.00  5.76 0.12
|   close group-unlock time = 0.00  0.00 0.00
|   close other + wait time = 0.00  0.00 0.00
|   file write        time  = 6.09 11.02 7.09
|
|   time scale: 1 = 2**7     clock ticks
                          min max         ave
|                                   ----------
----------  ---------- ---
|   total              =          679365650
|
|   imbalance     = \
           3661455     6419569   4608620  0%
|   open/trunc    = \
           124220702 124275751 124237723 18%
|   close sys = \
               348   146546323   2958218  0%
|   close fsync       = 0  0   0            0%
|   close group-unlock = 0  0   0           0%
|   close other + wait = 1 24   5           0%
|   local compute     = \
           3664      11145     4856       0%
|   wait for coll = \
|          270489159 394372058 367283785 54%
|   file write    = \
           154887311 280306349 180272438 26%
|   other            = 0   0   0            0%
|
|   raw write BW (MiB/s) = 887322.425
|   net write BW (MiB/s) = 235454.615
+---------------------------------------------+
```

Listing 6. MPI-IO Timers Example Output

fsync time' is incorrectly reporting the 'close sys time'. This issue will be fixed in an upcoming HPE Cray MPICH release. The raw and net write BW reporting is also helpful, include the data send rate for collective buffering, although understanding what is being reported is important as it can lead to misleading results. In this case both raw and net write rates are achieving well above the possible performance of the OSTs the file is striped across. This is a buffered I/O test which means data may held in page cache and not necessarily on disk. The raw write rate is the amount of data written divided by the average write time (6TiB / 7.09 sec in this example). This gives a general idea of the performance ranks or nodes are achieving but is optimistic

when the average and maximum are not equal. The net write uses the sum of the average times of all the components which should be closer to the real, application observed write rate however, again, a significantly larger maximum compared to the average makes the rate optimistic. The application visible write rate would be closer to total data written divided by the sum of all the maximum times.

When comparing timer output between collective buffering and local aggregation note that the "number of ranks writing" will include all ranks actually performing collective writes. Additionally, the time spent in "open/trunc" will account for a higher share of the average time spent; typically it will be second to "file write", depending on IO size, since all ranks and not just aggregators now need to "open" the file. Finally, with local aggregation the timer categories "close group-unlock" and "close other + wait" are likely to account for a notable share of average time spent. Significant time spent in "close other + wait", especially when their min and max vary by several orders of magnitude, is usually an indication that some ranks are waiting for one or more other ranks to complete the group-unlock operation as part of close(). The example provided in Listing 6 is heavily formatted to fit printing constraints.

V. General Guidance For Shared File Optimizations

Using the above diagnostic tools within HPE Cray MPICH, the goal is for an application user to be able to understand how to optimize their collective MPI-IO shared file workload. The following guidance is meant as a starting point when optimizing a shared file workload on a system.

For POSIX or MPI-IO independent shared file writes, we recommend Lustre Overstriping. The number of Overstripes will depend on the specific workload. Generally, if Lustre Overstriping will provide performance improvements you will see an increase in performance with even one additional Overstripe per OST (2 stripes per OST). In general, there are diminishing returns at Overstripe counts above 8 except for very specifically aligned workloads. For these workloads there are no locking optimizations available so selecting a Lustre stripe size and stripe count that aligns with the access sizes and patterns is important. As an example, for some workloads rank re-ordering or rank placement that allows for adjacent I/O requests from different ranks can allow those requests to be coalesced to a write on the same Lustre stripe. Considering the total amount of contiguous data written by all ranks on a node and selecting an equivalent Lustre stripe size and significantly reduce lock contention. Without direct I/O the performance from a single node doing shared file writes is currently limited to around 1.5 GB/s per node regardless of the number of ranks on a single node that are writing data.

In the case of collective MPI-IO it is recommended to enable either collective buffering or to use the experimental new feature of Local Aggregation as collective buffering. The optimized locking that these two options provide is important in achieving peak shared file write performance. If your application does a mix of independent and collective MPI-IO calls to the same file use Lustre Lockahead (`cray_cb_write_lock_mode=2`). Otherwise, using Lustre Group Locks (`cray_cb_write_lock_mode=1`) is recommended. Regardless of OST type it is recommended to use multiple aggregators per OST with current HPE Cray ClusterStor E1000 OSTs. One way to control this with MPI-IO hints is using the `cray_cb_nodes_multiplier` hint which represents the number of aggregators to be assigned per Lustre stripe. A value of 4 to 8 for disk OSTs and a value of 32 to 64 for NVMe OSTs are good starting points. If your application doesn't use enough compute nodes for such a high number of aggregator ranks considering using the `direct_io=true` MPI-IO hint to see if that provides improved performance. Direct I/O typically performs best at 16 MiB and larger I/O request and Lustre stripe sizes.

VI. Future Work

After the initial implementation and testing of the new feature there are several items identified throughout the paper that will be addressed in the future. We intend to continue testing once the fix for LU-16046 is available. First, we will investigate the data validation errors seen in early testing. Investigating and resolving the alignment issue with HDF5 in IOR will allow for direct I/O testing. Longer range work supporting collective MPI-IO write workloads includes allowing a configurable number of ranks per node to act as local aggregators in a similar approach to [8]. Highly packed nodes may benefit from fewer ranks performing I/O and yield improved write performance. Finally, applying the finding to at-scale applications to identify and quantify which workloads benefit from this optimization.

VII. Conclusion

The current generation of NVMe-based OSTs highlighted the need to re-visit shared file write performance, specifically collective MPI-IO workloads. Although the same file system optimizations are available on Lustre the way they are currently leveraged for collective MPI-IO is showing diminishing returns and requiring a high compute node to storage target ratio which isn't always feasible. The underlying file system optimizations are still still relevant and necessary to optimize collective MPI-IO write workloads, but they need to be applied in a new way. With NVMe-based OSTs there are lower latency request times and no penalty for seeking – both factors that made this approach non-performant in the past. A new optimization, Local Aggregation, uses the Lustre file system optimization of Group Locks to provide an optimized path to high throughput OSTs using collective MPI-IO. With each MPI rank taking a group lock and submitting its own I/O, the overhead of collective buffering, how group locks were previously used, is avoided. This experimental feature can be enabled using MPI-IO hints requiring no application code changes. Initial testing shows a significant performance improvement for smaller node counts and smaller I/O request sizes. Since this testing uses buffered I/O, which currently has a Lustre imposed low per-node performance limit, the results

suggest that collective buffering overhead is limiting performance since both tests have the same per-node shared file performance limit. With future testing using direct I/O we expect to see larger performance improvements at lower node counts as demonstrated in single node direct I/O tests.

ACKNOWLEDGMENT

Thanks to Doug C. for his assistance with system configuration tasks that enabled this testing.

REFERENCES

[1] P. Carns et al., “"Understanding and improving computational science storage access through continuous characterization," 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), Denver, CO, USA, 2011, pp. 1-14, doi: 10.1109/MSST.2011.5937212.

[2] (2006) “[lustre-devel] group locks design document. [Online]. Available: https://thr3ads.net/lustre-devel/2006/05/2056557-Group-locks-design-document

[3] M. Moore, P. Farrell, and B. Cernohous, “Lustre Lockahead: Early Experience and Performance using Optimized Locking,” presented at CUG 2017, 2017.

[4] M. Moore, P. Farrell, “Exploring Lustre Overstriping For Shared File Performance on Disk and Flash, “ presented at CUG 2019, 2019.

[5] (2017) “HDF5 Collective Metadata I/O Documentation” [Online]. Available: https://docs.hdfgroup.org/archive/support/HDF5/docNewFeatures/NewFeaturesCollectiveMetadataIoDocs.html.

[6] (2022) “[LU-16046] Shared-file I/O performance is poor under group lock - Whamcloud Community JIRA “ [Online]. Available: https:// https://jira.whamcloud.com/browse/LU-16046 .

[7] “Darshan – HPC I/O Characterization Tool” [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/

[8] Q. Kang et al., "Improving MPI Collective I/O for High Volume Non-Contiguous Requests With Intra-Node Aggregation," in IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 11, pp. 2682-2695, 1 Nov. 2020, doi: 10.1109/TPDS.2020.3000458.