Lustre Lockahead: Early Experience and Performance using Optimized Locking

Michael Moore, Patrick Farrell, Bob Cernohous
Cray, Inc.
St. Paul, MN, USA
mmoore,paf,bcernohous@cray.com

***Abstract*—Recent Cray-authored Lustre modifications known as Lustre Lockahead show significantly improved write performance for collective, shared-file I/O workloads. Initial tests show write performance improvements of more than 200% for small transfer sizes and over 100% for larger transfer sizes compared to traditional Lustre locking. Standard Lustre shared-file locking mechanisms limit scaling of shared file I/O performance on modern high performance Lustre servers. The new Lockahead feature provides a mechanism for applications (or libraries) with knowledge of their I/O patterns to overcome this limitation by explicitly requesting locks. MPI-IO is able to use this feature to dramatically improve shared file collective I/O performance, achieving more than 80% of file per process performance. This paper discusses our early experience using Lockahead with applications. We also present application and synthetic performance results and discuss performance considerations for applications that benefit from Lockahead.**

***Keywords*-File systems; Lustre;**

I. Introduction

POSIX I/O file access behavior is usually categorized as file-per-process or single-shared-file. Historically, file-per-process access has provided higher throughput than single-shared-file access due to required file system overhead ensuring consistency during shared write accesses. However, optimal I/O throughput is never the only, and rarely the primary, consideration when writing or using an application. Despite the performance downside, shared files are widely used for a variety of data management and ease of use reasons. To avoid long I/O time caused by shared file performance an application may reduce data output, checkpoint frequency, or number of jobs.

Shared file performance characterization is largely specific to each file system implementation and I/O library. This work focuses on modifications to the Lustre file system for applications using collective MPI-IO operations. A design and implementation in Lustre and Cray MPI libraries to improve shared file write performance by introducing a new Lustre locking scheme is described in this paper. Given MPI-IO allows requesting different file system lock modes via environment variables this work provides a path for applications to improve performance without application code modifications. Code changes within Lustre and MPICH/ROMIO to support Lustre Lockahead have been contributed back to the upstream communities [1], [2].

This paper has been submitted as an article in a special issue of Concurrency and Computation Practice and Experience on the Cray User Group 2017

This paper first describes application I/O behavior and standard libraries used for shared file access. The current Lustre shared file locking implementation is also described to motivate the need for an improved locking mechanism. Second, the implementation of Lustre Lockahead for collective MPI-IO operations is described for both the Lustre file system and the collective MPI-IO library. Next, comparative I/O performance of current file-per-process, independent and collective shared file I/O is presented to evaluate the benefit of this new locking method for Lustre. Finally, we describe early experience using Lustre Lockahead for an application with significant single-shared-file write performance requirements negatively impacting application walltime. Using that experience we discuss evaluating and tuning other applications using Lockahead to improve I/O performance.

A. Shared File Access Behavior

To understand the performance challenges of current single-shared-file performance and I/O workloads that may benefit from Lustre Lockahead a brief description of shared file access behavior within POSIX and MPI-IO is warranted. Although reading shared files is an important component of some workloads it is not specifically addressed in this paper. Many file system implementations, including Lustre, support multiple simultaneous readers.

Single-shared-file application write behavior is typically characterized as either

1) A single rank writing data to a shared file that all ranks may subsequently read
2) All ranks are responsible for writing data to non-overlapping segments within the shared file

Typically, a single rank writing data is used for small write operations that are not significant contributors to overall I/O time. All ranks in the application writing data to non-overlapping segments, as depicted in figure 1 comprises the bulk of shared file write transfers and the source of I/O performance limitations as described in section II-A.

1) POSIX and independent MPI-IO: Applications using standard POSIX (i.e. `write`) or independent MPI-IO calls (i.e. `MPI_File_write`) will generate I/O calls to the underlying file system for each application call on the executing

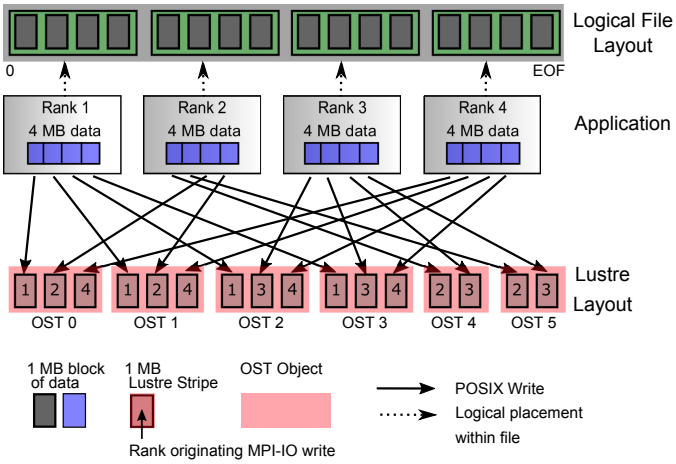

Fig. 1. POSIX and Independent MPI-IO shared file access pattern

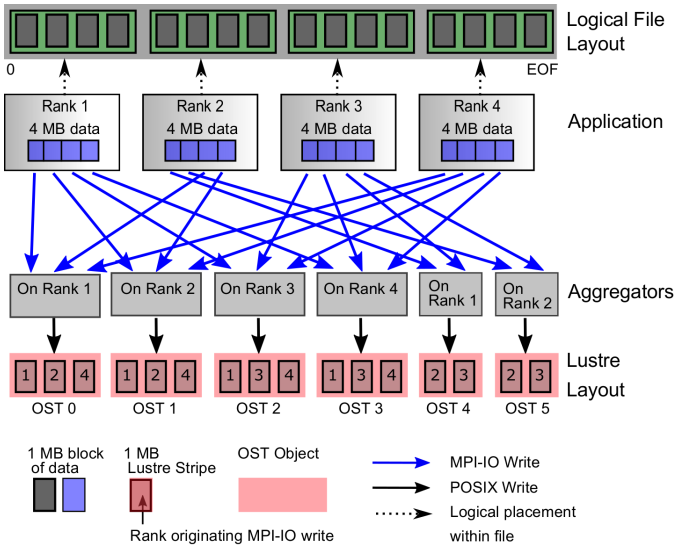

Fig. 2. MPI-IO Collective Buffering shared file access pattern

processing elements (PE) [1]. Access using these interfaces typically leads to significant contention as individual ranks attempt to write segments that are adjacent, although non-overlapping as depicted in figure 1. In specific cases, where guarantees of non-overlapping segments are present for a specific file, the optimal and safe locking behavior would be no locks at all.

2) Collective MPI-IO: In order to address several issues with shared file access, one being artificial write contention, many applications use MPI-IO collective calls (i.e. `MPI_File_write_at_all`). Collective MPI-IO calls allow the MPI library to organize write requests into more optimal requests to the underlying file system using a scheme known as collective buffering. A full review of previous work relating to MPI-IO collective operations is beyond the scope of this paper but many issues are highlighted in [3], [4], [5]. Within the context of locking for large, strided write accesses we focus on the collective buffering feature of collective MPI-IO operations. Collective buffering allows a specific set of ranks (aggregators) to be responsible for performing I/O requests to the underlying POSIX file system for a specific set of data as depicted in figure 2. The benefit of assigning specific ranks to specific sets of segments within a file, from a performance perspective, is allowing a single aggregator to be responsible for all data on a single OST. If the number of aggregators matches the number of OSTs in the file system, the I/O pattern to the underlying POSIX file system is equivalent to a file-per-process workload with a single file per OST [2].

3) Lustre Group Locking: A alternate locking method exists in Lustre currently, known as Group Locking [6], which requests Lustre to no longer acquire write locks. By enabling this locking mode on a file the application assumes responsibility for maintaining consistency of the file since the file system is no longer providing those guarantees. Although this technique addresses all write locking performance concerns it is not possible for most applications to use since expected POSIX consistency is violated. For this reason, collective MPI-IO operations do not support using group locks in the presence of independent I/O. Ensuring the consistency expected by applications accessing a POSIX file system is the reason an additional locking method that addresses both performance and consistency was pursued.

[1]For a single or multiple PE on a single compute node any file system client-side caching behaviors are still present

[2]Although we're accessing a shared file, each lock requested by a Lustre client is OST-specific so only one aggregator is requesting locks per OST using the current locking method

B. Lustre Shared File Access Constraints

Lustre is a high performance network file system which presents a (very nearly) POSIX compliant interface to the clients. As such, it faces significant difficulties in maintaining POSIX semantics around file updates from multiple clients. Unlike the NFS close-to-open consistency model [7], the POSIX consistency model requires that as soon as any I/O operation completes from the perspective of user space, any operations started after that must see the full results of that operation.

Lustre implements this through a distributed locking model under which a client doing an operation on a file is issued a lock on an appropriate region of the file, and when a different client performs a conflicting operation, the existing lock is canceled. As part of lock cancellation, the client holding the lock invalidates any read data, and pushes to the server all writes covered by the lock.

This enables local caching while maintaining POSIX semantics, but it has costs. In particular, only one client can hold a write lock on a particular range of a file at a time. For complex I/O patterns like strided I/O, this poses a challenge. In the case of Lustre and MPI I/O, the cost of dealing with this pattern can be significant. The next section will explain in detail the existing Lustre data locking, what problems it causes, and how Lockahead can use knowledge of the I/O pattern to alleviate this.

II. Lustre Lockahead

A. Lustre Locking Behavior

Lustre's distributed locking (the locking which mediates conflicts between clients) is called the Lustre Distributed Lock Manager (LDLM). LDLM locking is the key to how Lustre implements a single globally visible POSIX file system with standard semantics while allowing updates from many clients at once. LDLM has been described extensively elsewhere so we will limit ourselves to very specific aspects of its behavior [8]. It is enough to know that LDLM locks, their conflicts, and cancellation behavior serve to implement a single consistent view of files being modified by multiple clients.

Two general notes regarding the following description

1) Where not otherwise specified, we are talking about write locks, which cannot overlap.
2) Where not otherwise specified, this discussion assumes a singly striped file. The issues discussed are per stripe, but the language required to explain them is greatly simplified if we limit ourselves to one stripe at a time.

When a client accesses a file, it looks for an LDLM lock on the part of the file it wants to access. If it does not already have such a lock, it sends a request to the OST (Object Server Target, a Lustre data target) which contains that part of the file. (If the region of the file spans multiple OSTs due to striping, the client will request locks from each OST.) The client requests a lock on only the byte range which it is actively trying to read or write. The client is guaranteed to get a lock on at least this byte range, because this is necessary to complete the I/O request. Once the client has received the required lock, it performs the actual I/O. Notice that a lock request adds a network round trip delay to that I/O.

Returning to the lock request, the server can return a lock on a larger byte range than the client requested, and generally grants the largest lock possible. If there are no conflicting locks, this will be a lock on the whole file. In general this is excellent behavior, because a client will usually continue accessing other areas of the file, and if the required lock is already available, the lock request can be avoided.

This behavior is problematic in the case of multiple writers to a single file. Multiple writers to a single file generally use the strided pattern where writers alternate blocks of the file. In a simple two writer example, writer one would write block zero, then block two, etc., while writer two would write block one, block three, etc. Critically, in this pattern, different writers never write to the same block/byte range of a file, and so they should be able to proceed in parallel. Unfortunately, the default Lustre locking optimization behavior prevents this.

Each write operation requests a lock on only the bytes necessary to complete that operation, but when the first write lock request is made, there are no locks on the file. Therefore, the server extends the lock to cover the whole file. Then the second write lock request arrives, and it encounters a conflicting lock. The server must then contact the holder of that lock and call back (i.e. cancel) that lock. Once that lock is canceled, the server processes the second lock request

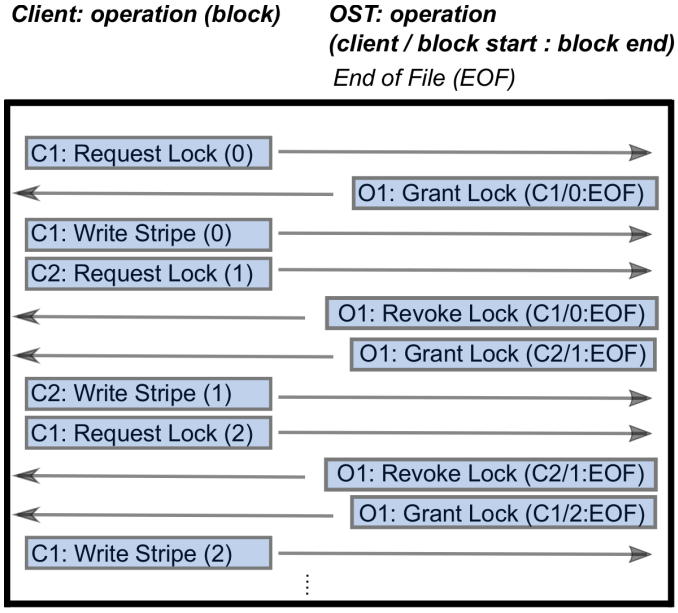

Fig. 3. Default Lustre locking shared file lock conflicts

and sees that there are no locks on the file. It then grants a full file lock to the client which made the second write request. As illustrated in figure 3 this continues throughout write operations, not only serializing the clients but also adding significant latency. This is so severe that two or more writers tend to perform worse than one.

The obvious solution is to disable lock expansion, which allows parallelism between multiple clients. However, this has the effect of adding the extra round trip time for a lock request to every write operation. Testing shows this to be worse than the status quo [9].

This means there are in effect two problems to be solved:

1) Serialization due to false lock conflicts.
2) The latency overhead of the lock requests required to perform our I/O.

Lockahead solves the first by allowing user space to request locks on specific extents separately from performing I/O (The server is not allowed to expand these locks). If an application (or library such as MPI I/O) knows the I/O pattern it will use, it can use this to avoid false conflicts by requesting exactly the locks it needs.

Lockahead solves the second by issuing those requests asynchronously and in parallel, using the same network daemons which handle most Lustre network operations. This means in effect that the round-trip delays for all of the lock requests can be overlapped, making for a total delay comparable to the time for one request.

Implementing this in Lustre required creating a user space interface for the LDLM lock request mechanism, separate from the requests automatically created as part of I/O. It also required adding an LDLM level flag to tell the server not to expand the lock beyond the requested extent. Creating the possibility of locks not explicitly associated with I/O

had interesting implications for the methods Lustre uses to determine file size, but a detailed discussion of those issues is out of scope for this paper.

For those wishing to know more, internal implementation details are available in the references cited immediately below. Of particular interest are the changes required to handle file size efficiently, referenced in those presentations, and found in `ofd_dlm.c` in the Lustre source [9] , [10], [11]. The code itself is available [12].

B. User space Interface and MPI-IO

The user space interface for Lockahead is either, in current Cray Lustre clients, a dedicated ioctl, or in Lustre 2.10 (release estimated in summer 2017), the ladvise interface. In either case, the interface requires a description of the locks to be requested, in to a structure defined by the Lustre headers. The description of a lock is three parts:

1) Lock mode (read or write)
2) Start (file offset, bytes)
3) End (file offset, bytes)

All of these should be self-explanatory. User space provides a count and an array of these descriptions, then Lustre makes lock requests as specified in the descriptions. The result of each request is stored in a location in the description structure. Note that if a request results in an actual lock request to the server, we cannot provide the status through this interface. This is because those requests are made asynchronously, and waiting for the results would defeat the utility of this interface [3].

MPI-IO, as previously described, collects the data to be written to the file system and only a few aggregators will make the POSIX calls to write the data. MPI-IO uses information from the file system to setup a non-overlapping I/O pattern so that each aggregator writes to different sections of the file. For Lustre, it's a simple algorithmic assignment –with *N* aggregators each aggregator *n = 0..N-1* starts at the *n* stripe and locks every *Nth* stripe from there.

With this pattern in Lustre, each aggregator optimally writes to different stripes and different OSTs when possible. MPI-IO expects that this I/O pattern will enable the aggregators to be able to proceed with parallel non-conflicting writes. But as previously described, Lustre servers often return locks larger than the requested byte range which then cause contention and must be canceled for the next aggregator's lock requests.

After all aggregator locks are issued and resolved, and the data is written, an analysis of the Lustre file locks remaining would show the expected ideal pattern where individual aggregators lock non-overlapping stripes. But creating this pattern of locks was a slow, often serialized and contentious process. With Lustre Lockahead, MPI-IO can create this pattern by request using the calculated offsets and lengths for each extent that each aggregator expects to write. MPI-IO collective aggregators are guaranteed not to write to any other aggregator's locked extents. Ideally this eliminates all lock contention and lock canceling.

There is a limitation to using Lustre Lockahead in MPI-IO. Lustre Lockahead is implemented as an array of byte offsets and lengths called extents. MPI-IO has to predict how many extents and ranges to lock ahead of the writes. Currently MPI-IO uses Lockahead assuming that the application will write from the beginning of the file and proceed in a mostly sequential pattern through the file. So initially, for *N* aggregators, MPI-IO starts locking stripes *0* through *N* up through a configurable limit which currently defaults to 500. Aggregator *0* locks every *Nth* stripe from 0 to 499. Aggregator *N* locks every *Nth* stripe from *N* to *N+499*.

If MPI-IO predicts the wrong range of extents and a write occurs before or after the locked extents, then a new range of extents must be locked and the overhead of requesting the unused locks is potentially detrimental to overall performance. As a result, an application that does I/O to widely scattered or at random offsets is unlikely to benefit as much from Lockahead.

III. I/O Performance

A. Test Environment

Tests were performed on an XC30 with 252 compute nodes connected to a 12 SSU Sonexion 2000 which contains 24 OSSes and OSTs with approximately 1.4 PiB in capacity. The XC compute node image was based on CLE 5.2 UP04 but included a custom Lustre client, version 2.7, in order to support the Lockahead. Lustre client version 2.7 is available in CLE 6.0. The Sonexion 2000 was running NEO 2.0 SU23 with a special patch to address a specific issue related to Lockahead for large aggregator-per-OST tests. Cray MPT version 7.6.0 pre-release was used in testing. The minimum required Cray software versions to support Lustre Lockahead are CLE 6.0 with Lustre Client 2.7.1 and Cray MPT 7.4.0. The upcoming release of NEO 2.0 SU24 is expected to contain the patch used in testing.

The system used for testing is primarily sized for I/O testing. For that reason it has an atypical node memory to storage ratio compared to many customer environments. The compute nodes are Ivy Bridge processors with 64 GB of memory per compute node As mentioned above, the file system under test is a 12 SSU Sonexion 2000. This provides a compute node to OST ratio of *9.3 : 1* or, using another metric, expected file system throughput allows all nodes to write memory to stable storage in under 3 minutes. There are, as designed, adequate compute nodes to achieve maximum file system performance but this can provide a skewed view of typical application performance. For synthetic I/O tests all OSTs are used. For application performance measurements the number of SSUs is varied to provide a range of compute to storage resource ratios.

B. IOR

IOR was used as a synthetic benchmark to measure I/O performance. A fixed amount of data is written for each test

[3]The best simple example of both interfaces are the unit tests written for them [13], [14]

for a single iteration. File-per-process (FPP) tests use balanced file counts per OST and the Lustre stripe size is 1 MiB and file stripe count is 1. All shared file tests are striped across all OSTs (24) and the stripe size is set to the transfer size. Additionally, shared file tests set the IOR block size equal to the IOR transfer size to more accurately simulate strided access patterns.

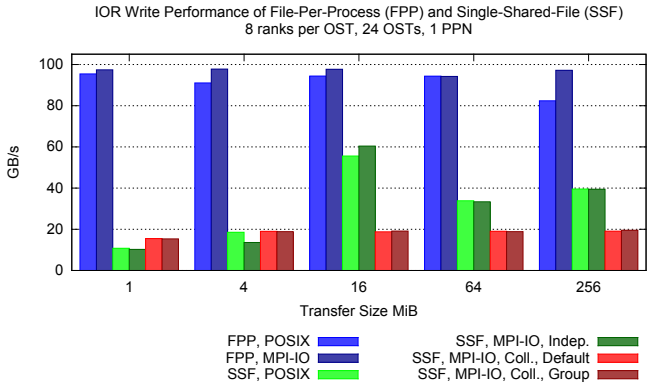

Fig. 4. Comparison of existing file-per-process and shared file performance

To begin, we illustrate the current performance of typical file-per-process and shared file I/O performance in figure 4. The overhead imposed for POSIX consistency on shared writes, in both shared POSIX and independent MPI-IO tests, is significant. The gap between file-per-process and shared file collective MPI-IO is the challenge that motivated the need for strided locking as implemented in Lockahead.

Collective MPI-IO tests in figure 4 used the same total number of ranks as other tests but only used a single aggregator per OST since that is the standard number of aggregators used for collective MPI-IO today. Collective MPI-IO performance is consistent regardless of the transfer size which illustrates that large transfer sizes are constrained by the single aggregator, relative to independent I/O. Smaller transfer sizes tend to benefit from the reduced lock contention provided by collective MPI-IO. However, for these specific tests, shared file performance is typically 20% and not better than 65% of file per process performance [4].

As described in section I-A1 and section II-A, POSIX and independent MPI-IO, in the context of Lustre, are problematic due to multiple writers per OST causing false lock sharing. Figure 5 illustrates the limitated performance for file-per-process IOR tests with a single writer per OST and the similar performance achieved by POSIX and independent MPI-IO IOR tests when lock conflicts are avoided by a single rank accessing each OST. Single-shared-file tests are within 10% of POSIX and MPI-IO file-per-process performance excluding the anomaly of POSIX single-shared-file 64 MB transfers. However, collective MPI-IO performance is significantly lower than file-per-process and independent single-shared-file perfor-

[4]These tests do not represent optimal shared file performance, rather, a comparable test to illustrate factors influencing shared file performance

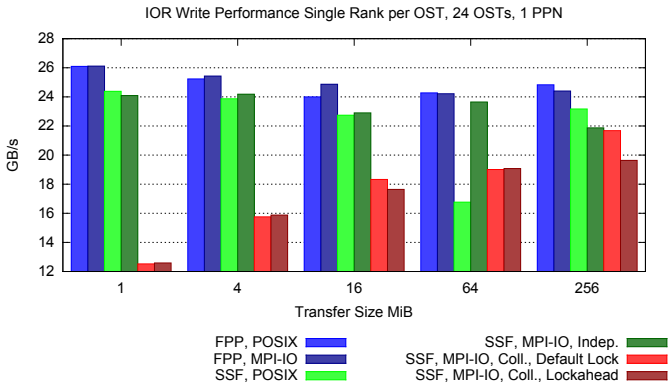

Fig. 5. Comparison of single writer per OST file-per-process and single-shared-file performance

mance due to the limited number of ranks performing MPI-IO collective buffering and ranks issuing collective MPI-IO calls. A total of one rank per OST for collective MPI-IO workloads is not a meaningful workload but is intended to illustrate the role of locking and lock expansion relative to file-per-process workloads.

The collective MPI-IO write performance in figure 6 depicts a typical single-shared-file workload –192 compute nodes with 4 processes per node (PPN). The default locking method shows that for small, $\leq$ 4MB transfer sizes, using a single aggregator avoids lock contention that can limit performance. Transfer sizes $\geq$ 16MB achieve increased performance with multiple aggregators per OST due to the larger Lustre stripe size and MPI-IO aggregators issuing striped sizes write requests. The overhead of lock contention is mitigated by increasing the size of the locks.

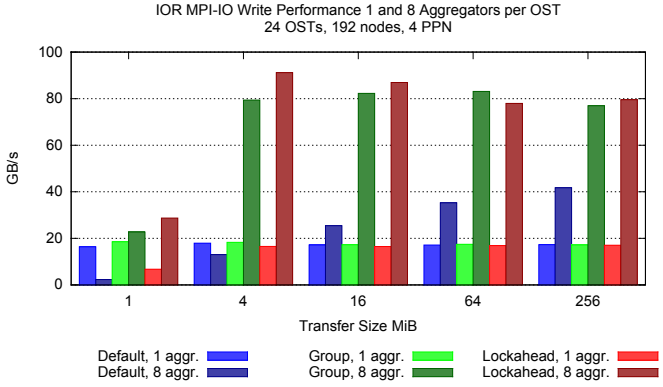

Fig. 6. Comparison of MPI-IO performance for Lustre locking schemes

As an evaluation of the Lustre Lockahead implementation through MPI-IO figure 6 shows that Lustre Lockahead achieves similar performance to group locking, where no write locks are required, see section I-A3 for additional details. Lockahead with collective MPI-IO is able to asynchronously acquire the necessary write locks, prior to when the write is issued, to avoid both lock acquisition latency and lock

contention present in default Lustre locking.

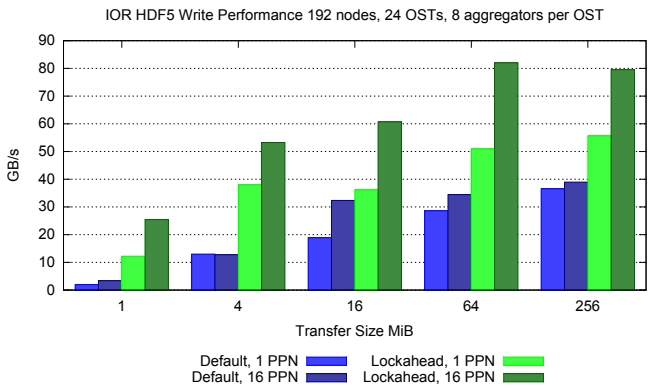

Fig. 7. Comparison of HDF5 Write Performance using Default and Lockahead Locking Schemes

The comparative performance of default and Lockahead locking in Lustre for a higher-level I/O library is shown in figure 7. As mentioned, HDF5 requires small, independent writes so group locking is possible. The IOR HDF5 results include tests using one and 16 processes per node (PPN) with a fixed count of 192 nodes which illustrates that collective MPI-IO with collective buffering is able to achieve higher I/O throughput when more processes are making MPI-IO calls for a given number of aggregators[5]. Since the Lustre stripe size is modified to match the transfer size the independent effect of transfer size and Lustre stripe size cannot be inferred from these tests.

Synthetic I/O benchmarks demonstrate the improvement Lockahead can provide single-shared-file I/O write performance compared to default locking. Next, we evaluate a candidate application and describe our early experience using Lockahead.

IV. Application Performance Early Experience

Selecting an application to use for evaluating Lockahead required using prior knowledge of an application's I/O workload. Given previous investigations into I/O performance, The Weather Research & Forecasting Model, WRF, was selected as the application for early experience testing[15], [16]. Study of a specific weather system that has significant I/O requirements was selected.

A. WRF

1) I/O Workload: WRF was configured to use Parallel NetCDF (PNetCDF) [17] for relevant input and output files. The application testing focused on the write time of forecast history (`auxhist`) and restart (`wrfout`) files. The communication, computation, and file read time are collectively referred to as `other` in subsequent figures. The use of PnetCDF allows testing Lockahead performance only modifying the MPI-IO hints specified in the job script.

[5]A maximum of one rank per node is used as an MPI-IO aggregator

This particular WRF workflow, from the Arctic Region Supercomputing Center benchmark suite [18] accesses a 77 GB input file and generates a total of 11 history files of 28 GB each and 10 restart files of 81 GB each. Depending on the I/O performance the application walltime is between 15 and 60 minutes. The I/O requirements for a WRF work flow is dependent on the specific data set and data output requirements; this particular work flow is intentionally I/O intensive but is not performing an unrealistically excessive amount of I/O.

2) Application Environment Changes: Minimal changes were required to specify the Lustre locking mode and number of aggregators per OST to use. MPI-IO hints were specified for write locking mode and aggregators per OST. The number of OSTs to use for a given test were specified by setting Lustre striping parameters on the output in the job script. Additionally, a Lustre pool was assigned to the directory matching the desired number of OSTs to ensure the pairs of OSTs were selected from SSUs. For that reason, the striping factor was not specified in the MPI-IO hints and the Cray MPI library selected striping attributes based on the directory striping information. The total number of requested aggregator ranks for a given file is $striping\_factor * cray\_cb\_nodes\_multiplier$ and, by default, MPICH attempts to scatter aggregator ranks when selecting placement. The environment variable used to specify MPI-IO hints was *MPICH_MPIIO_HINTS*. Each output file expression for `auxhist` and `wrfout` specified the following two attributes:

- `cray_cb_write_lock_mode`
- `cray_cb_nodes_multiplier`

Application runs depicted in the figures use one of three `cray_cb_write_lock_mode` values: "0" for default locking, "1" for group locking, or "2" for Lockahead locking. The string `LOCKMODE` is replaced with the appropriate integer value for the environment variable . The string `AGGREGATOR_MULTIPLE` is replaced with the integer of the desired number of aggregator ranks per OST. The general format of the environment variable used to specify MPI-IO hints in the job script is specified in listing 1.

```
setenv MPICH_MPIIO_HINTS "wrfout*:cray_cb_write_lock_mode=
    LOCKMODE:cray_cb_nodes_multiplier=AGGREGATOR_MULTIPLE,
     auxhist*:cray_cb_write_lock_mode=LOCKMODE:
     cray_cb_nodes_multiplier=AGGREGATOR_MULTIPLE"
```

Listing 1. Example MPICH_MPIIO_HINTS environment variable

3) Application Runtime: The application run time is measured using the wall time as the total run time. The time spent for writing each copy of the `auxhist` and `wrfout` files is combined in the following figures. The remainder of the walltime (compute, communication, I/O reading) is given in the `other` segment. Across all tests, regardless of Lustre locking scheme, the `other` time is relatively constant as illustrated in figure 8. However, the `wrfout` and `auxhist` time is relatively equal across all stripe counts using Lustre

Lockahead which indicates that, in this system, $\geq$ 4 OSTs performs similarly to using all OSTs. In the case of default locking, there are incremental improvements by striping the single-shared-file output data over larger stripe counts. Shifting the dominant time from I/O to computation or communication allows for future application walltime reductions as other system components, such as processors or memory, improve in performance.

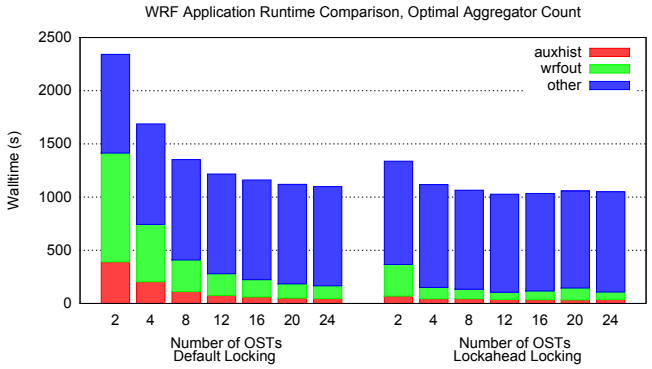

Fig. 8. Comparison of WRF application run time using optimal aggregators per OST

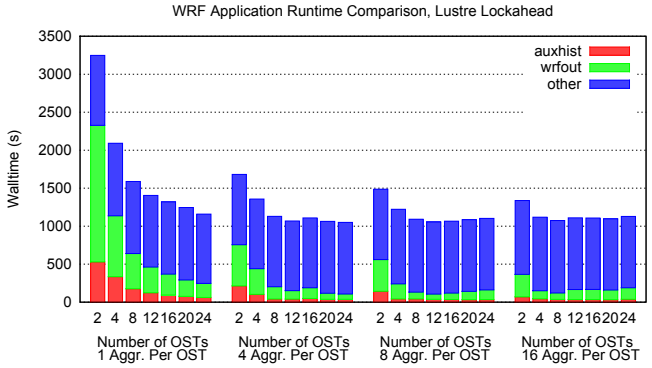

Fig. 9. WRF run time evaluating effect of aggregators per OST using Lustre Lockahead

Figure 9 depicts the performance for a range of aggregator counts and OSTs to further investigate the effect of changing aggregators per OST. Although the application `other` time dominates overall run time, once an adequate number of aggregators were used (8) the addition of more aggregators per OST caused no reduced performance as would be typical of default locking.

For this specific application, when using 4 OSTs, the single-shared-file write I/O time using Lockahead is 20% of the time using default locking (150 seconds compared to 743 seconds). Assuming a standard computation to storage ratio, such as 192 nodes to 4 OSTs, and a similar weather system study, a significant improvement of application walltime and storage throughput utilization would be expected. In the next section we discuss evaluating an application for use with Lockahead and some of the possible parameters for optimizing Lockahead for collective MPI-IO write workloads.

V. Evaluating Lockahead for Application I/O Improvements

In addition to the basic collective MPI-IO write operations and environment variable requirements to enable Lustre Lockahead there are several factors that determine if a given application using a given file system will experience decreased application walltime. In this section we discuss evaluating an application's potential single-shared-file write performance improvement for collective MPI-IO write operations and tuning options to maximize single-shared-file write performance with Lockahead.

A. Evaluating Potential Application Performance Improvement

Potential improvement in throughput and application walltime using Lockahead for collective MPI-IO writes is determined by the number of ranks performing collective writes, the number of ranks used as aggregators per OST, the total number of collective writes, the size of collective writes and the number of system writes or Lustre stripe size. If performing application tests with Lockahead to determine application walltime improvement is not possible, running the application using default locking with Cray MPI-IO statistics and timers enabled can provide an indication of potential improvement.

Adding the appropriate environment variables to the application's job script will enable collection and output of the MPI-IO statistics. The MPI-IO statistic related environment variables used in tests for this paper, and recommended for investigation of MPI-IO behavior are given 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. MPI-IO statistic reporting environment variables

```
MPIIO write access patterns for IOR_HDF5
independent writes      = 85
collective writes       = 196608
independent writers     = 84
aggregators             = 192
stripe count            = 24
stripe size             = 16777216
system writes           = 196757
stripe sized writes     = 196544
aggregators active      = 0,0,0,196608 (1,<=96,>96,192)
total bytes for writes  = 3145728 MiB = 3072 GiB
ave system write size   = 16764511
read-modify-write count = 0
read-modify-write bytes = 0
number of write gaps    = 18
ave write gap size      = 642300800454
```

Listing 3. Example IOR HDF5 write access pattern using default locking

```
MPIIO write access patterns for wrfout_d01
independent writes      = 2
collective writes       = 638400
independent writers     = 1
aggregators             = 64
stripe count            = 4
stripe size             = 1048576
system writes           = 82334
stripe sized writes     = 82091
aggregators active    = 177840,0,0,460560 (1,<=32,>32,64)
total bytes for writes = 82192 MiB = 80 GiB
ave system write size   = 1046770
read-modify-write count= 0
read-modify-write bytes= 0
number of write gaps    = 2
ave write gap size      = 524284
lock ahead extent count= 96000
lock ahead write hits   = 82204
lock ahead write misses= 128
```

Listing 4. Example WRF pNetCDF write access pattern

1) MPI-IO Access Pattern Statistics: MPI-IO stats and timer output are written to the job's standard error for each accessed file. Initially, the write access pattern details can be used to validate the intended MPI-IO hints were used. Listing 3 was created by an IOR HDF5 write test using a 16 MiB stripe size, 24 OSTs, and 8 aggregators per OST. Inspecting the attributes `aggregators`, `stripe count`, and `stripe size` confirms that 192 aggregators $8 * 24$, 24 OSTs, and a 16 MiB Lustre stripe size were used.

Assuming the expected Lustre striping and aggregator attributes are present, two other statistics provide an indication of the application's potential collective MPI-IO write performance improvement using Lustre Lockahead. First, the number of `collective writes` represent the number of MPI-IO writes that will be sent to aggregators which could benefit from Lockahead. The independent write calls do not benefit from Lockahead and can negatively impact collective writes due to cancellation of Lockahead locks. Listing 3 represents a typical number of independent writes for an HDF5 file. I/O libraries using MPI-IO typically have a small number of independent writes near file open or close and cause minimal lock cancellations.

Second, the `aggregators active` field shows how many aggregators are active using 4 "buckets"; 1 aggregator, > 1 and $\leq 50\%$ of aggregators, $> 50\%$ and $<$ all aggregators or all aggregators were active issuing I/O requests. A higher count of more aggregators active indicate that adequate data is available to write to the file system and aggregators are active writing the data. For IOR, all aggregators are typically active since the ranks are only issuing I/O. Applications will report less idealized `aggregator active` statistics such as listing 4 collected from a WRF test in section IV. Early experience evaluating applications for use with Lockahead indicates that a good candidate application will have some counts in the all aggregators bucket.

Finally, although it requires enabling Lockahead, the statistics section includes counters specific to Lockahead behavior. The counters for `lock ahead write hits` and `lock ahead write misses` give a direct indication of how effective Lockahead was in acquiring locks for writes prior to when they were necessary for writes.

```
MPIIO write by phases, writers only, for wrfout_d01
                                          min          max          ave
  file write          time      =       50.28        71.02        61.69

  time scale: 1 = 2**9    clock ticks     min          max          ave
  total                       =                                621076237

  imbalance                   =       13702        17650        15896  0%
  local compute               =     2904270      3954987      3344989  0%
  wait for coll               =    27883170     38036293     33823494  5%
  collective                  =      216151       311404       265380  0%
  exchange/write              =      432356       484632       463949  0%
  data send                   =    52859942    101826753     75238921 12%
  file write                  =   265243604    374646356    325433382 52%
  other                       =   154855180    212509150    181886450 29%

  data send BW (MiB/s)        =                                 80.916
  raw write BW (MiB/s)        =                               1332.366
  net write BW (MiB/s)        =                                698.137
```

Listing 5. WRF wrfout timer statistics for writers using default Lustre locking

```
MPIIO write by phases, writers only, for wrfout_d01
                                          min          max          ave
  file write          time      =        2.90         6.31         4.72

  time scale: 1 = 2**7    clock ticks     min          max          ave
  total                       =                                874687367

  imbalance                   =       52981        69529        62477  0%
  local compute               =    11564823     15692947     13303061  1%
  wait for coll               =    13154931     21812791     19040471  2%
  collective                  =      866203      1252425      1064713  0%
  exchange/write              =     1748750      1934323      1870153  0%
  data send                   =    69631810    145128634     97582694 11%
  lock mode                   =      294934       423117       365912  0%
  file write                  =    61251755    133053597     99598729 11%
  other                       =   524346902    724441695    635362122 72%

  data send BW (MiB/s)        =                                249.554
  raw write BW (MiB/s)        =                              17413.726
  net write BW (MiB/s)        =                               1982.863
```

Listing 6. WRF timer statistics for writers using Lustre Lockahead

2) MPI-IO Timer Statistics: Additional MPI-IO statistics, enabled using `MPICH_MPIIO_TIMERS`, were added in Cray MPT 7.5.1. The timer statistics provide detail on how time is spent by MPI-IO ranks and MPI-IO aggregators in the various phases of MPI-IO library activities. For each file accessed a section of statistics is generated for reads and writes for all ranks and only writers (aggregators), listing 5. For our purposes we focus on the percentage of time spent between phases in the `MPIIO write by phases, writers only` statistics.

In listing 5, where default locking is used, a majority of the aggregator's time was spent performing file writes (52%) while only 11% of time was needed for sending data to other aggregators for collective buffering. The imbalance in time between `data send` and `file write` indicate that improving file write performance could yield reduced time for collective MPI-IO writes (and reduced application walltime). The same statistics collected using Lockahead show an even percentage of time (11%) for data send and file write.

MPI-IO statistics and timers provide an indication if an application can benefit from improved I/O throughput using Lockahead. An application using collective MPI-IO may not

write enough data to each OST, frequently issue independent writes, or have several other behaviors that limit the actual improvement from Lockahead. Beyond the I/O performance improvement, it's equally important to evaluate the amount of time spent performing I/O relative to total application walltime. As seen in section IV Lockahead greatly improved the MPI-IO write portion of the application, reducing write time to 20% of write time using default locking, but, the computation and communication portion of the application remained constant.

B. Lustre Lockahead Tuning

The synthetic benchmark results in section III and application early experience in section IV evaluated the most important tunings for an application with Lustre Lockahead. The two major parameters are stripe count and number of aggregators per stripe.

As depicted in figure 9, achieving optimal throughput requires specifying adequate aggregators per OST. The optimal number of aggregators is dependent on the performance of the Lustre OSTs. For example, a Sonexion 1600 OST requires fewer aggregators than a Sonexion 2000 OST. There is little to no negative impact on I/O performance by using too many aggregators per OST, however, there is also no benefit.

The minimum necessary stripe count is also an important tunable for an application to specify. Figure 9 illustrates there is no benefit to striping shared files beyond 4 OSTs. Although there is no observed negative performance impact when using more stripes on a dedicated system that may not hold true on a shared system where other jobs are accessing other OSTs.

VI. Conclusion

The current locking methods available in Lustre limit single-shared-file performance relative to file-per-process performance and the capabilities of the underlying storage system. A new locking scheme for Lustre was described and the Lustre and Cray MPI-IO implementations were evaluated using synthetic benchmarks and a real application. Synthetic benchmark tests showed optimal single-shared-file performance within 10% of optimal file-per-process performance. Additionally, single-shared-file performance using Lockahead was equivalent to performance of single-shared-files using no write locks (group locking). The importance of selecting Lustre stripe count and aggregators per OST parameters in achieving optimal write performance with Lockahead was discussed. Finally, several values from MPI-IO statistic and timer output were used to evaluate an application's potential benefit from Lockahead for single-shared-file output.

An application is able to gain benefits from Lockahead by specifying MPI-IO hints through an environment variable without other application changes. Any application using collective MPI-IO, either directly or through an I/O library, can leverage Lockahead in place of default locking. Generally, if an application is spending a significant amount of time performing single-shared-file writes it will likely benefit from using Lockahead. However, the gains a specific application will experience are dependent on a variety of application I/O factors in addition to file system characteristics.

Acknowledgment

In memory of David Charles Knaak, a long time Cray MPI-IO designer and engineer. His design for strided locking was the basis and inspiration for what has now evolved into Lustre Lockahead.

We would like to acknowledge the contributions to this work of the following people

Bob Fielder for his manuscript input and WRF application support including identifying and staging data sets, configuring and sizing jobs on the test system used for data collection in this paper, and explanations of WRF behavior.

Joe Glenski for his valuable feedback throughout the process from proposal to presentation.

Peter Johnsen for his WRF application support including early staging data sets, configuring and sizing jobs prior to data collection for this paper, and explanations of WRF behavior.

Norm Troullier for his valuable feedback on the manuscript.

Richard Walsh for his valuable feedback on the abstract and manuscript.

REFERENCES

[1] (2017) Lustre. [Online]. Available: http://lustre.org/

[2] (2017) Mpich — high-performance portable mpi. [Online]. Available: http://www.mpich.org/

[3] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in ROMIO," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–.

[4] K. Coloma, A. Ching, A. Choudhary, W. k. Liao, R. Ross, R. Thakur, and L. Ward, "A new flexible MPI collective i/o implementation," in *2006 IEEE International Conference on Cluster Computing*, Sept 2006, pp. 1–10.

[5] R. Latham, R. Ross, and R. Thakur, *The Impact of File Systems on MPI-IO Scalability*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 87–96.

[6] (2006) [lustre-devel] group locks design document. [Online]. Available: https://thr3ads.net/lustre-devel/2006/05/2056557-Group-locks-design-document

[7] (2001) Close-to-open cache consistency in the linux NFS client - draft. [Online]. Available: http://www.citi.umich.edu/projects/nfs-perf/results/cel/dnlc.html

[8] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding Lustre Filesystem Internals," Oak Ridge National Lab., National Center for Computational Sciences, Technical Report ORNL/TM-2009/117, 2009.

[9] (2015) Shared file performance improvements LDLM lock ahead. [Online]. Available: http://cdn.opensfs.org/wp-content/uploads/2015/04/Shared-File-Performance-in-Lustre_Farrell.pdf

[10] (2015) Shared file performance in Lustre: Challenges and solutions. [Online]. Available: http://youtu.be/ITfZfV5QzIs

[11] (2015) Shared file performance improvements LDLM lock ahead deep dive. [Online]. Available: http://wiki.lustre.org/images/2/2d/Lock_Ahead_-_Technical_Discussion.pptx

[12] (2017) Lu-6179 llite: Implement ladvise lockahead. [Online]. Available: https://review.whamcloud.com/#/c/13564/

[13] (2017). [Online]. Available: https://review.whamcloud.com/#/c/13564/24/lustre/tests/lock_ahead_test.c

[14] (2017). [Online]. Available: https://review.whamcloud.com/#/c/13564/56/lustre/tests/lockahead_test.c

[15] T. Balle and P. Johnsen, "Improving i/o performance of the weather research and forecast (WRF) model," presented at CUG 2016, 2016.

[16] (2017) The weather research & forecasting model. [Online]. Available: http://http://www.wrf-model.org/index.php

[17] (2017) Parallel netCDF. [Online]. Available: http://cucis.ece.northwestern.edu/projects/PnetCDF/

[18] ARSC WRF benchmark suite. [Online]. Available: http://http://weather.arsc.edu/BenchmarkSuite/