Lock Ahead

Overview

This feature is meant to improve shared file performance of MPI-IO collective buffering. This method aggregates IO to a specific set of nodes, each of which handles parts of a file. Writes are strided and non-overlapping. In this pattern, the lock count per OST is equal to the file size in MB divided by the numbers of OSTs, assuming 1MB striping.

Lock ahead requests are non-blocking. If client B holds a lock on part of the file, and client A requests a lock ahead on the same part of the file, client A's request will fail. Lock ahead requests are asynchronis.

Write or read lock requests are blocking lock requests, with one exception which is group lock.

Group locks are using the library to clearing out any other locks before requesting group locks. They'r exclusive, blocking, and absolute. That guarantees the file is clean.

Reference

- LU-6179, Lock ahead - Request extent locks from userspace
- Code review of LU-6179 llite: Implement lock ahead
- LUG15 Presentation
- LUG15 Presentation Video

Usage

Lockahead is enabled by setting cray_cb_write_lock_mode=**4** in MPICH_MPIIO_HINTS

cray_cb_write_lock_mode=0 is standard Lustre file locking.

cray_cb_write_lock_mode=1 is a Lustre group lock.

cray_cb_write_lock_mode=2 is request only.

cray_cb_write_lock_mode=2 is strided.

cray_cb_write_lock_mode=4 is lock ahead.

Note: cray_cb_nodes_multiplier must be set to greater than 1, otherwise the lock mode is ignored and defaults to 0, standard Lustre locking.

cray_cb_nodes_multiplier is the number of collective buffering aggregators per OST.

```
# Example MPICH_MPIIO_HINTS that enables Lockahead
export
MPICH_MPIIO_HINTS="*:romio_no_indep_rw=true:romio_cb_write=enable:cray_cb_nodes_multip
lier=2:cray_cb_write_lock_mode=2"
```

Text for the intro_mpi man page, in the MPICH_MPIIO_HINTS section (note: it has not been updated to reflect Lockahead yet):

```
cray_cb_write_lock_mode

Specifies the file locking mode for accessing Lustre files.
Valid values are:

0 Standard Lustre file locking. Extent locks are held by each MPI
rank accessing the file, and locks are revoked and reissued when
the extent of one lock conflicts with the extent of another lock.

1 A single lock is shared by all MPI ranks that are writing the file.
This locking mode is only valid if the only accesses to the file
are writes and all the writes are done by the aggregators doing
collective buffering. The romio_no_indep_rw hint must be set to
true to use this locking mode. This is an explicit assertion that
all file accesses will be with MPI collective I/O. Setting the
romio_no_indep_rw hint also sets romio_cb_write and romio_cb_read
to enable. Any other MPI I/O accesses will cause the program to
abort and any non-MPI I/O access may cause the program to hang.
Both HDF5 and netCDF do both collective and independent I/O so
this locking mode is not appropriate for these APIs.

This locking mode reduces lock contention and therefore supports
greater parallelism by allowing multiple aggregators per OST to
efficiently write to the file. Set the cray_cb_nodes_multiplier
hint to 2 or more to get the increased parallelism. The optimal
value depends on file system characteristics.

Default: 0

cray_cb_nodes_multiplier

Specifies the number of collective buffering aggregators (cb nodes)
per OST for Lustre files. In other words, the number of aggregators
is the stripe count (striping_factor) times the multiplier. This may
improve or degrade I/O performance, depending on the file locking
mode and other conditions. When the locking mode is 0, a multiplier
of 1 is usually best for writing the file. When the locking mode is
1, a multiplier of 2 or more is usually best. If a locking mode is
specified and both cb_nodes and cray_cb_nodes_multiplier hints are set,
the cb_nodes hint is ignored. See cray_cb_write_lock_mode.

Note: If the number of aggregators exceeds the number of compute nodes,
performance generally won't improve over 1 aggregator per compute node.
When reading a file with collective buffering, a multiplier of 2 or
more often improves read performance.

Default: 1
```

Application Testing

IOR

NERSC and HLRS use both MPI-IO and HDF5 heavily for acceptance criteria. IOR is capable of both MPI-IO and HDF5, so we can do a lot of testing with IOR. IOR is also able to verify data.

At a minimum, we need to demonstrate correctness and performance using IOR and these interfaces.

- MPI-IO
- HDF5

A note about buffer cache. The way NERSC specifies using IOR, they us large files so that the read isn't using the cache (theoretical exhaustion of caches). David does one write job, one read job. That makes sense because we clear caches between jobs. A write/read in the same job can be quite a bit slower (lock efficiency). So we need to cover the NERSC base, but can test faster with separate codes.

- Explore using the -C flag to IOR to reorder tasks
    - Reads from different nodes to avoid reading from cache on node that wrote the data

A note about MPI-IO support for Lock Ahead functionality. It will be in the Cray ADIO before it is in the open source ADIO from Argonne. When using the Cray Programming Environment, we will need to specify the Cray ADIO and use hints to specify a lockmode.

Cray ADIO has a lock mode that you promise you're only going to write through MPIIO collective buffering. Difficulty is job then reads or does non-collective write then it will hang. Irregular pattern that won't work with group lock but should work with Lock Ahead. HDF5 does need to do

writes. We should understand relative performance of group lock and Lock Ahead.

Customer Application

Pick a user application. Make sure it works. Before and after performance comparison. David has some notes on what a good candidate might be. Also has some friendly sites that might share something.

Custom Applications

Create a user level MPI job that doesn't use MPI-IO that directly accesses the ioctl interface. Bob's code could potentially be used as a starter here.

Verify File Size and Correctness

Read disruption

Take the size of the file (total size we know we're going to write, not from stat), read half a page to 2M from a random location (ahead or behind lock ahead) in the file. We expect that it doesn't disrupt the lock ahead writing of data. We don't care about the contents read or if the read succeeds necessarily, we want to verify through lock ahead the data written is consistent.

Random points, random amounts. Performance impact is going to be significant. We're trying to gum up lock ahead.

We may need to rate limit the reads. Try to not be overly disruptive, aim for moderately.
Keep a count of how many reads, total size

doio_mpi could probably be modified to do these reads

Write visibility testing

Consistency between clients. On client A, we write 1MB. Write call returns to user space. At that point in time, we send MPI message to a reader on another node (same node not as interesting), that client tries to read that data. Lustre layers tell writer to write data to disk ASAP, gives lock back to server, server grants read lock to other client and they should be able to read the data. Reads attempted 100th of a millesecond behind the write should get consistent.

Stat performance

Repeatedly stat the file during I/O (from different clients). Confirm that performance doesn't tank.

Mix Lock Modes

Check non-Lock Ahead and Lock Ahead workload on the same OST. Don’t want performance to tank more than just contention and sharing
Run one mpiio collective job to an ost, and another job to the same OST, shouldn't see >60% decrease

- Explore using the -F flag for IOR
    - Each node writes to its own file, could setup directories to use particular OSTs
    - Two different IOR jobs (one Lock Ahead, one with -F) could accomplish this

Lock Volume

Get a LOT of locks out there. What is the upper limit of lock counts for the OST?
MDS can handle a lot of them.

Torture Test

Torture test where we're ping-ponging locks on purpose. Can you trigger client evictions?
Patrick doesn't see this as an issue.

Create incorrect/other patterns

Request lockahead with a particular pattern in mind, then do a different mind. Request lock ahead locks that would match

Two clients requesting lock. Strided writing. Have client A be off by one and client B off by one. Every write would fall in lock space requested by other.

You have 10 clients writing out a 10GB file, they should each be taking each 10th lock. Have every client request locks at random file size, they try to write out file. Performance should be bad, but it should still have consistent data.

Failover and Recovery

RPCs should be resent by the stand rules of resending a lock request, these aren't any different. Not super interesting, but worth doing to make sure we get consistent results.

Lock volume goes up quite a bit, so failover might take longer. We could just put a high lock volume as stress. If every job on the system uses Lock Ahead, how much does failover slow down?

Identifying Recommended Parameters

More of a tuning thing with the library. How far ahead do you go. How tight is the bounds of how many locks we request in advance?

API feature. Actual lock requests sent out of the network by the API is not visible to you.

When you build lock request, you check to see if another lock available matches or is greater than the area you requested. No asynch component if it matches a local lock.

With lock ahead you want to request 500, 1000, 2000 ahead. You want to know that the first locks will be granted by the time you're ready to write.

Starting to write after 5 locks granted probably isn't enough.

What you do is, request 100 locks, check to see if the first one is back yet. Only way to check for a lock is to make a request, so it could bounce around and cause trouble.

Get a return code
0 - I made a request to server
1 - I found exactly the lock you asked for byte for byte
2 - I found some lock of different dimension that matches
3 - I found lock that doesn't match but since this is an async request I am not going to grant it to you

Debugging

You can make Lustre dump a list of the locks. dlmtrace debug trace. clear, echo 1 into the proc entry, it will dump the lock list into the dk buffer, turn off dlmtrace and clear

You could compare the number of locks that were there at the end. Can compare to the number of locks you requested.