Shared File Performance Improvements

LDLM Lock Ahead

Patrick Farrell (paf@cray.com)

Shared File vs FPP

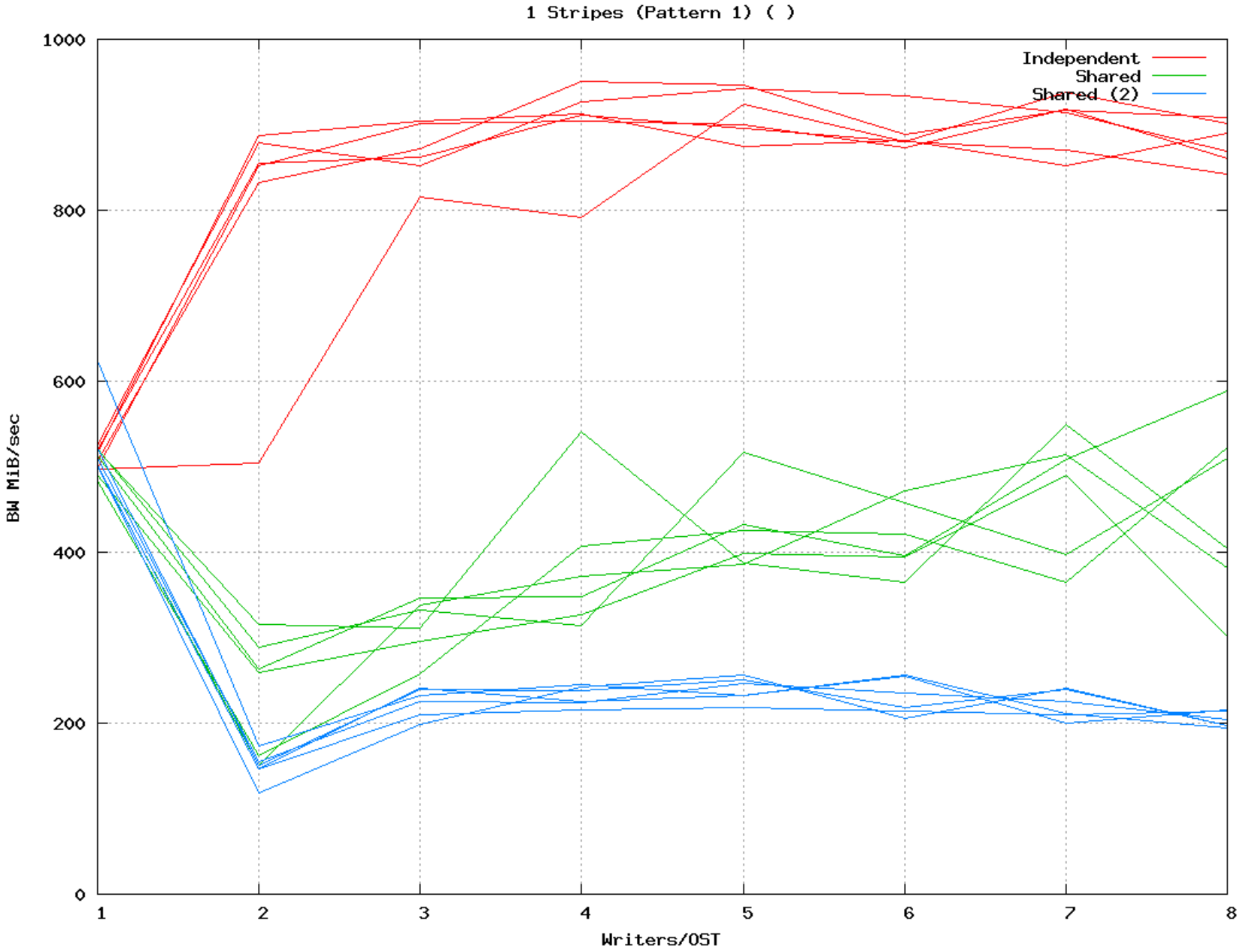
- Two principal approaches to writing out data: File-per-process or single shared file
- File-per-process scales well in Lustre, shared file does not
- File-per-process has problems:
- Heavy metadata load for large jobs
- Many cores → Many files
- Getting worse: ~250,000 cores on current top 10 x86 machines

Shared file IO

- Common HPC applications use MPIIO library to do ‘good’ shared file IO
- Technique is called collective buffering
- IO is aggregated to a set of nodes, each of which handles parts of a file
- Writes are strided, non-overlapping
- Example: Client 1 is responsible for writes to block 0, block 2, block 4, etc., client 2 is responsible for block 1, block 3, etc.
- Currently arranged so there is one client per OST

Shared File Scalability

- Bandwidth best at one client per OST
- Going from one to two clients reduces bandwidth dramatically, adding more after two doesn't help much
- In real systems, OST can handle full bandwidth of several clients (FPP hits these limits)
- For example, latest Seagate system OSTs have enough bandwidth for 8+ Cray clients per OST

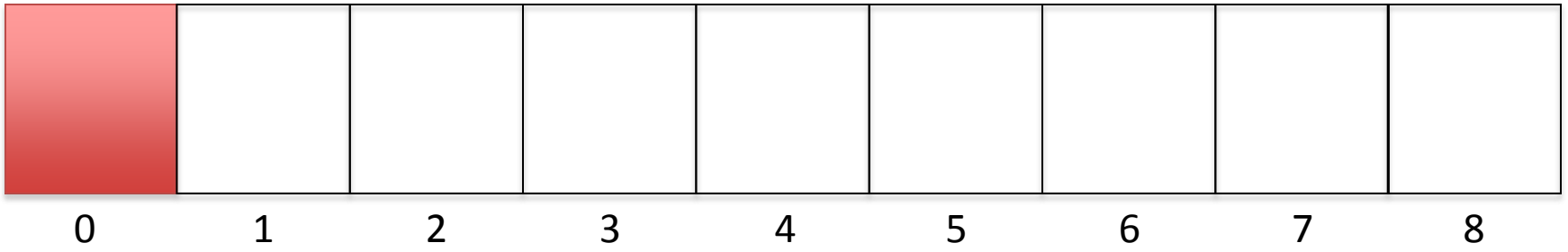

1 Stripes (Pattern 1) ( )
Independent
Shared
Shared (2)
BW MiB/sec
Writers/OST
0
200
400
600
800
1000
1
2
3
4
5
6
7
8

Why doesn't shared file IO scale?

- In 'good' shared file IO, writes are strided, non-overlapping
- Since writes don't overlap, should be possible to have multiple clients per OST without lock contention
- With > 1 client per OST, writes are serialized due to LDLM* extent lock design in Lustre
- 2+ clients are slower than one due to lock contention

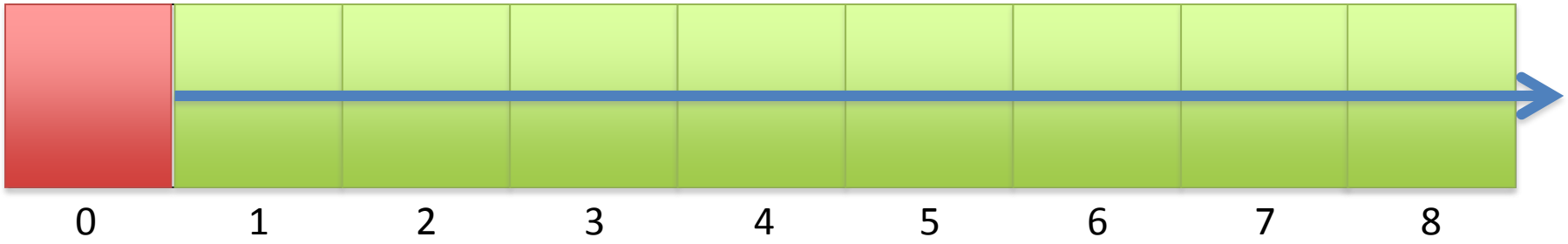
*LDLM locks are Lustre's distributed locks, used on clients and servers

Extent Lock Contention



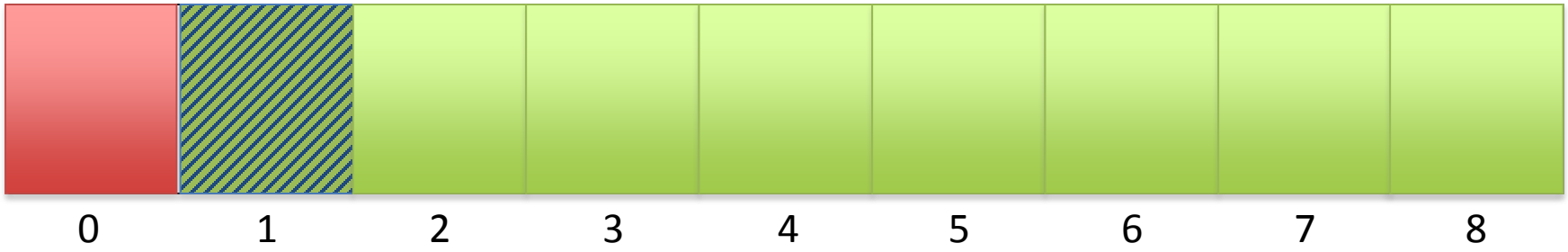
- Single OST view of a file, also applies to individual OSTs in a striped file
- Two clients, doing strided writes
- Client 1 asks to write segment 0 (Assume stripe size segments)

Extent Lock Contention



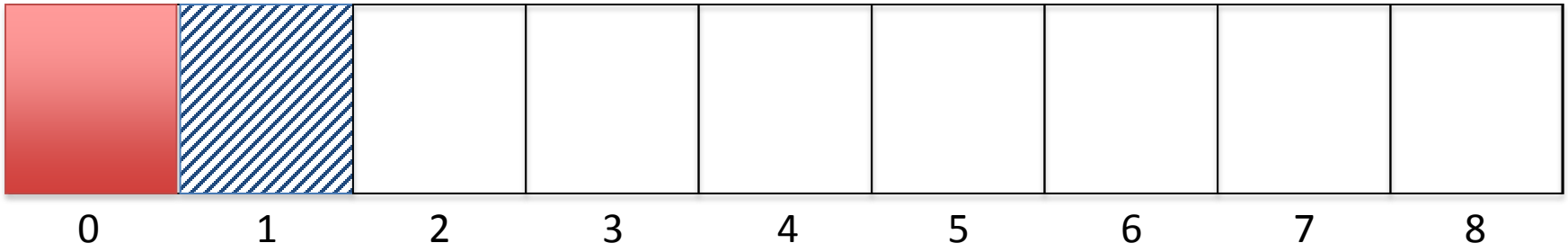
- No locks on file currently
- Server expands lock requested by client 1, grants a lock on the whole file

Extent Lock Contention



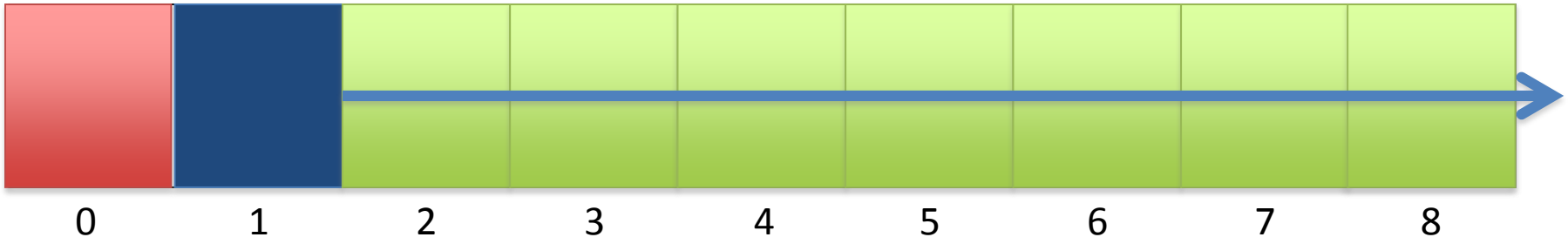
- Client 2 asks to write segment 1
- Conflicts with the expanded lock granted to client 1

Extent Lock Contention



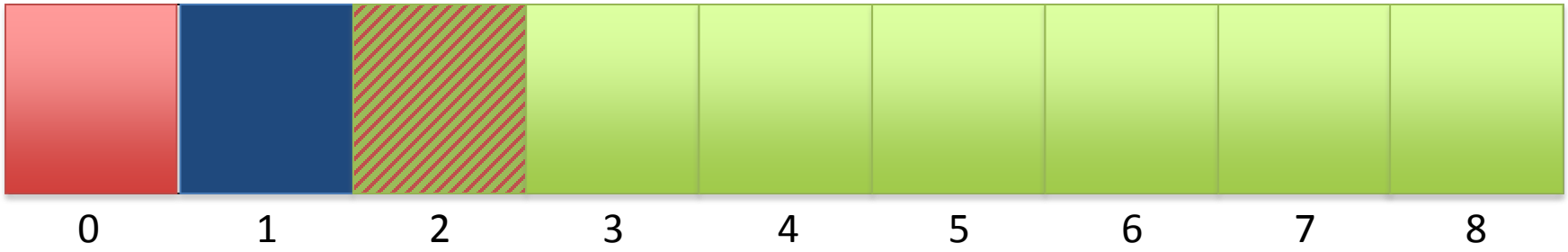
- Lock assigned to client 1 is called back
- Client 2 lock request is processed…

Extent Lock Contention



- Lock for client 1 was called back, so no locks on file currently
- OST expands lock request from client 2
- Grants lock on rest of file…

Extent Lock Contention



- Client 1 asks to write segment 2
- Conflicts with the expanded lock granted to client 2
- Lock for client 2 is called back…
- Etc. Continues throughout IO.

Extent Lock Contention

- Multiple clients per OST are completely serialized, no parallel writing at all
- Even worse: Additional latency to exchange lock
- Mitigation: Clients generally are able to write > one segment before giving up lock

Extent Lock Contention

- What about not expanding locks?
- Avoids contention, clients can write in parallel
- Surprise: It's actually **worse**
- This means we need a lock for every write, latency kills performance
- That was the blue line at the very bottom of the performance graph…

Proposal: Lock Ahead

- Lock ahead: Allow clients to request locks on file regions in advance of IO
- Pros:

Request lock on part of a file with an IOCTL, server grants lock only on requested extent (no expansion)

Flexible, can optimize other IO patterns

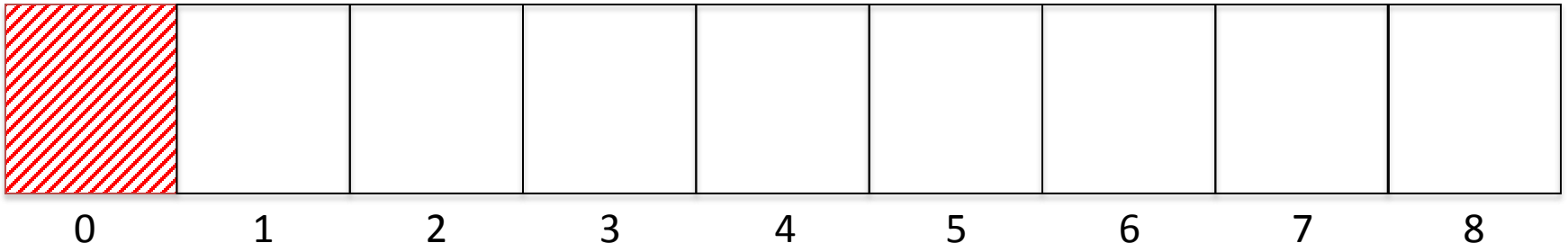
Relatively easy to implement

- Cons:

Large files drive up lock count and can hurt performance

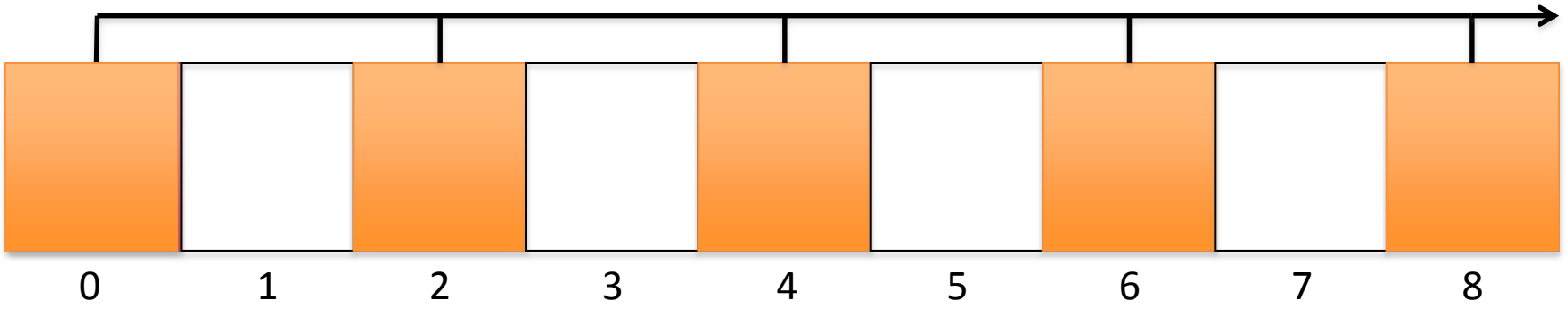
Pushes LDLM in to new areas, exposes bugs

Lock Ahead: Request locks to match the IO pattern



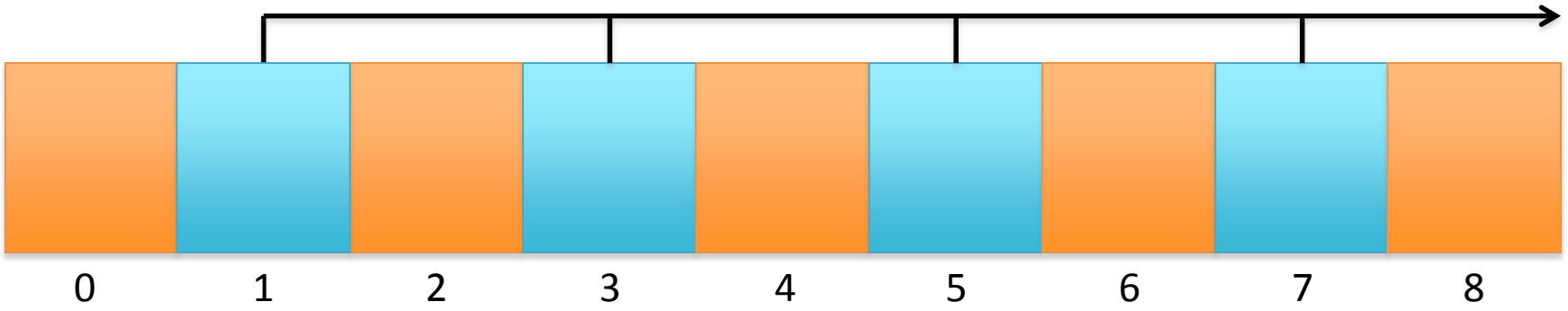
- Imagine requesting locks ahead of time
- Same situation: Client 1 wants to write segment 0
- But before that, it requests locks…

Lock Ahead: Request locks to match the IO pattern



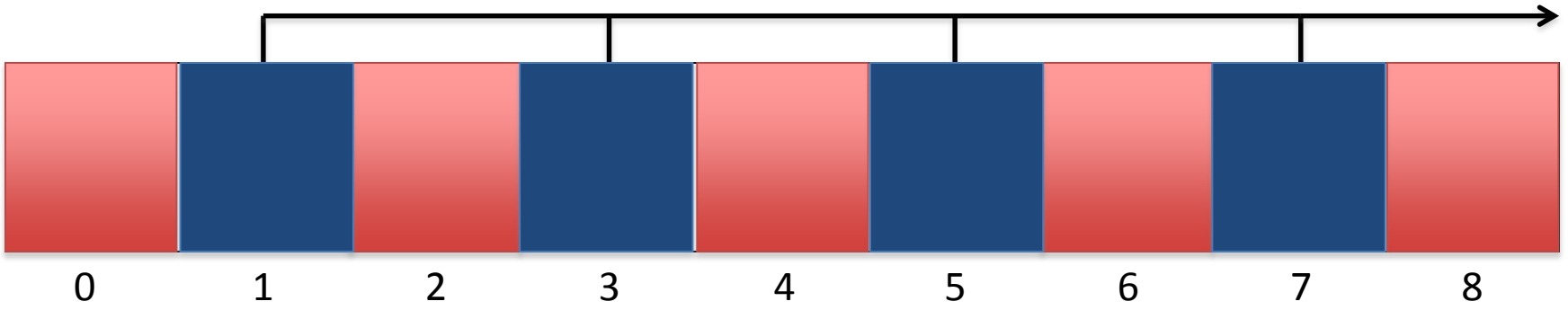
- Request locks on segments the client intends to do IO on
- 0, 2, 4, etc.
- Lock ahead locks are not expanded

Lock Ahead: Request locks to match the IO pattern



- Client 2 requests locks on its segments
- Segments 1,3,5, etc.

Lock Ahead: Request locks to match the IO pattern



- With locks issued, clients can do IO in parallel
- No lock conflicts.

What about Group Locks?

- Lustre has an existing solution: Group locks
- Basically turns off LDLM locking on a file for group members, allows file-per-process performance for group members
- Tricky: Since lock is shared between clients, there are write visibility issues (Clients assume they are the only one with a lock, do not notice file updates until the lock is released and cancelled)
- Must release the lock to get write visibility between clients

What about Group Locks?

- Works for some workloads, but not OK for many others
- Not really compatible with HDF5 and other such file formats:
  In file metadata updates require write visibility between clients during the IO
- It’s possible to fsync and release the lock after every write, but speed benefits are lost

Lock Ahead: Performance

- Early performance results show performance equal to file-per-process or group locks
- Unable to test large files (200 GB+) due to bugs in current code

Lock Ahead: Performance

- Intended to match up with MPIIO collective buffering feature described earlier
- Freely available in the Lustre ADIO, originally from Argonne, improved by CFS/Sun
- IOR –a MPIIO –c
- Cray will make a Lustre ADIO patch available
- Codes need to be rebuilt but not modified

Lock Ahead: Implementation

- Re-uses much of Asynchronous Glimpse Lock (AGL) implementation
- Adds an LDLM flag to tell server not to expand lock ahead locks
- Other issues will be covered in detail at a developer's day talk after LUG

Lock Ahead: When can I have it?

- Targeted as a feature for Lustre 2.8
- Depends on client & server side changes: No using this feature with new clients with old servers

What's up with Strided Locks?

- Proposed previously, lock ahead is a simpler solution
- Incomplete prototype is still up at **LU-6148**
- Work on hold: Lock ahead locks are simpler and may meet our needs
- We'll see…

Other Information

- Thank you to Cray engineers David Knaak Bob Cernohous for inspiration and assistance testing
- Thanks to Jinshan Xiong and Andreas Dilger of Intel for suggestion of lock ahead and assistance with design

Finally:

- Lock ahead work in **LU-6179**
- MPI-IO ADIO patch will be linked from there (will try to submit upstream)
- For ugly implementation details, come to the developer's day discussion
- **Any questions?**
- Happy to answer questions later or by email (paf@cray.com)