Parallel Directory Operations Solution Architecture

Introduction

Unknown macro: 'html'

Increasing performance without re-engineering applications is highly desirable for HPC Users. OpenSFS and Whamcloud are engaged in the removal of a known bottleneck that will deliver this benefit to users on the Lustre filesystem. This document presents the limitation and a proposed solution to provide a step-wise performance boost to a wide class of HPC applications.

Single directory performance is a critical for HPC workloads. In a typical use case an application creates a separate output file for each node and task in a job. As nodes and tasks increase, hundreds of thousands of files may be created in a single directory within a short window of time. Also, on the Lustre OSS the internal object directories can grow to have millions of entries, which can result in significant overhead under some workloads.

Today, both filename lookup and file system modifying operations (such as create and unlink) are protected with a single lock for an entire ldiskfs directory. OpenSFS and Whamcloud will remove this bottleneck by introducing a parallel locking mechanism for entire ldiskfs directories. This work will enable multiple application threads to simultaneously lookup, create and unlink in parallel: Parallel Directory Operations (PDO).

Solution Requirements

The characteristics of successful implementation are described in this section. Parallel Directory Operations (PDO) is concerned with a single component of the Lustre filesysetem: ldiskfs. The Solution Requirements ensure that work on a single component can be verified against the strategic goals of the Lustre community.

Ldiskfs is responsible for storing data to disk and is part of the application stack that a user assumes is completely reliable. In addition, new performance must not come at cost to the system somewhere else: performance must be preserved for single thread operations. Finally, new code will be freely licensed under the GNU GPL, and to be of value to the community it must be easy to maintain.

Parallelize file creation, lookup, and unlink under large shared directory

Ldiskfs uses a hashed-btree (htree) to organize and locate directory entries, which is protected by a single mutex lock. The single lock protection strategy is simple to understand and implement, but is also a performance bottleneck because directory operations must obtain and hold the lock for their duration. The PDO project implements a new locking mechanism that ensures it is safe for multiple threads to concurrently search and/or modify directory entries in the htree. PDO means MDS and OSS service threads can process multiple create, lookup, and unlink requests in parallel for the shared directory. Users will see performance improvement for these commonly performed operations.

It should be noted that the benefit of PDO may not be visible to applications if the files being accessed are striped across many OSTs. In this case, the performance bottleneck may be with the MDS accessing the many OSTs, and not necessarily the MDS directory.

No performance degradation for operations on a small directory

Htree directories with parallel directory operations will provide optimal performance for large directories. However, within a directory the minimum unit of parallelism is a single directory block (on the order of 50-100 files, depending on filename length). Parallel directory operations will not show performance scaling for modifications within a single directory block, but should not degrade in performance.

No performance regressions for single thread operations

In order to be practically useful, any new locking mechanism should should maintain or reduce resource consumption compared to the previous mechanism. To measure this, performance of PDO with a single application thread should be similar to that of the previous mechanism.

Easy to maintain

The existing htree implementation is well tested and in common usage. To avoid deviating from this state, it is not desirable to significantly restructure the htree implementation of ldiskfs. Ideally, the new lock implementation would be completely external to ldiskfs. In reality, this is not possible but a maintainable PDO implementation will minimize in-line changes and maximize ease of maintenance.

Use Case

Alice writes an application for her 100000 core machine. Each hour her application creates a checkpoint file using the job name, thread rank, and timestep number as components the checkpoint filenames. Each thread writes it's output to the same directory in the Lustre filesystem '/parallel/alice/app/checkpoint /'. This can quickly cause a directory to have millions of entries, and can reduce application efficiency.

MDS service threads need to modify the same directory to create the checkpoint files. Without PDO, all of these threads will be serialized on the single directory lock, significantly increasing the time that each checkpoint takes. This is due to threads waiting to create their checkpoint file, as well as underutilizing the IO bandwidth for writing the actual checkpoint data until enough threads have created new files.

The PDO project will:

1. Increase concurrency lookup operations within the shared directory, allowing more disk concurrency and IO merging.
2. Increase concurrency of service threads while they are modifying a shared directory.
3. Reduce threads context switch (sleep/wakeup) due to contention on the single lock.

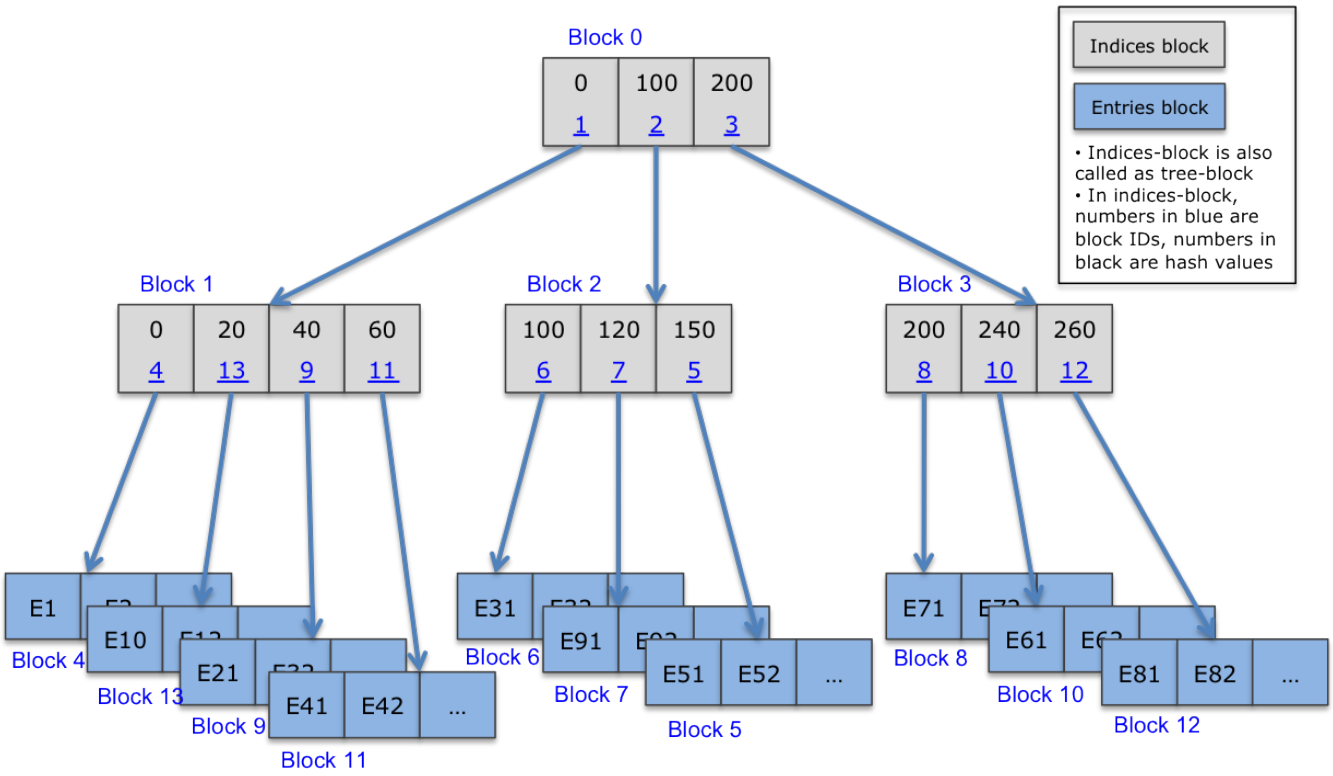
Solution Proposal

Overview of indexed directory structure

There are two types of blocks in an ldiskfs directory:

1. Leaf block (or entries block) stores directory entries (filenames)
2. Tree block (or indices block) stores hash-value/block-ID pairs
    a. Hash value: hash value of the entry name.
    b. Block-ID: either file logical block number of leaf block, or the next level indices block

An Example of ldiskfs abstract directory structure



This figure illustrates how storage and lookup takes place on entries in a htree directory. Gray blocks are tree blocks, and blue blocks are leaf blocks. Value pairs in tree blocks are [hash, block-ID], while strings in leaf blocks are "filenames". For example, if hash-value of name "E52" is 155, first the top level tree block-0 is scaned and the hash value 155 (between hash 100 - 199) is found in the next level tree block-2. Searching block-2 shows that hash 155 (higher than hash 150) is in leaf block-5, where the filename "E52" can be found.

An Ldiskfs directory stores all name entries in an leaf block. The leaf blocks can contain between 15 - 340 entries, depending on the filename length (between 4 and 256 characters), but typically around 100 entries. It is not possible to parallelize locking at a smaller granularity than a single leaf block.

For directories with only a single leaf block, there is no tree block. When the directory grows and name entries overflow one leaf block, ldiskfs will mark the directory as “indexed directory”. A this point, name entries will be sorted by hash-value and the leaf block will split into two leaf blocks at the median hash value. A tree block will be allocated to store the [hash, block-ID] for these two leaf blocks. At this point, the structure is described as a htree.

Leaf blocks are split again and again as the size of the directory grows. Tree blocks can be split as well if the number of leaf blocks increases to the point where their indices overflow one tree block (more than 510 indices).

Today, ldiskfs locks the whole htree whenever a thread needs to modify any block in the tree. The locking mechanism for PDO is designed to lock at the granularity of a block-ID or hash-value for the htree. This new locking mechanism is called the htree-lock.

htree-lock

The htree lock is characterized as follows:

1. htree-lock is an advanced read/write lock.
   five lock modes are available: EX, PW, PR, CW, CR. Compatibility matrix of these lock modes are identical to Lustre Distributed Lock Manager (DLM) lock matrix.

Unknown macro: 'html'

1. htree-lock is a blocking lock
   An application thread may sleep if it fails to obtain a lock immediately.

1. When the lock holder has taken an htree-lock with shared modes (i.e: CW or CR), it can continue to apply child lock for protecting a specific key:

1. Child lock only has two modes: PW and PR
2. Child lock can be used to protect a specific key value, for example, block ID
3. A lock holder can lock 1-N keys, these keys should be used to represent different resources, for example: child-lock::keys[0] is protecting a tree block, child-lock::keys[1] is protecting a leaf block.

1. Lock holder is responsible for child lock ordering to avoid deadlock.
2. htree-lock can be used to protect directory and support parallel directory operations.

ldiskfs and OSD-ldiskfs with htree-lock

1. OSD will create a htree-lock for each directory if PDO is enabled.
2. OSD will pass in the htree-lock into ldiskfs via exported ldiskfs APIs

1. If the directory is not indexed (a small directory), OSD will take htree-lock (mode=EX) for any changing operation, or htree-lock (mode=PR) for any reading/searching operation before calling into ldiskfs APIs. Ldiskfs does not need to take any child lock in this case.
2. If the directory is indexed (a large directory), OSD will take htree-lock(mode=CW) for any changing operation, and htree-lock (mode=CR) for any non-changing operation. ldiskfs have to take child lock (mode=PW/PR) to protect target block (leaf block).
3. ldiskfs may need to take another child lock to protect tree block (or tree block) if a leaf block needs to be split.
4. If the directory htree grows and a tree block requires splitting or a new level to the tree is needed, ldiskfs will relock the htree-lock with EX mode. With EX mode, no other threads can access the htree and the htree can be safely changed. Locking the whole tree with EX mode is expensive but it is expect to be very uncommon. For example, if an assume a leaf block can contain 80 entries, and a tree block can contain 511 [hash, block-ID] pairs. In this case, it is necessary to split the tree block during insertion once every 512 * 80 = 40,960 name entries. This means the chance that an EX lock is required is 1/40,960. This is sufficiently small to ignore as a performance affect.

Graceful Fallback: calling ldiskfs from VFS

If ldiskfs is called directly from VFS without Lustre, htree-lock will be NULL and ldiskfs will assume the directory is well protected by mutex in VFS. This behavior makes PDO ldiskfs gracefully degrade to single directory operations when accessed via the VFS interface (e.g. if using ldiskfs to mount the MDT locally).

N-levels hash-tree

Very large directories can contain many millions of files. Today, current htree of ldiskfs only has two levels, and can have at most about 15 million files. To enable PDO changes to htree, the ldiskfs implementation will be made to support N-levels of htree and even larger directories. This is not in the requirements of PDO, but has been included in scope as it is judged by OpenSFS and Whamcloud to be a worthwhile addition to PDO work.

Big buffer LRU

Least Recently Used (LRU) Buffer is a per-CPU cache used in Linux for fast searching of buffers. The default LRU size is 8. This default value is considered too small for Lustre where support N-level htree for very large directories.

PDO with very large directories as N-level htree is expected to consume LRU for quick reference buffers. As a consequence, in-use buffers may be purged from the LRU cache prematurely if the cache is too small, leading to repeated lookup of the buffer in the block device cache.

Purging of an active buffer will significantly degrade performance as a slow and expensive buffer searching path will need to be traversed. To avoid this scenario, an additional patch to configure the LRU buffer size will be provided and a default value of 16 will be set.

Tunables

A tunable for ldiskfs to allow the user to turn on/off htree-lock at runtime will greatly simplify testing and benchmarking of this feature. This tunable is available for testing purposes only, and is not expected to be useful to administrators, and so will not be extensively documented. PDO is an enhancement that will be enabled by default and no use cases have been anticipated that will require it to be disabled, though this may be used as a short-term workaround in the unlikely case that this feature causes instability.

Unit/Integration Test Plan

Verify PDO performance improvement by mdt-survey.

The mdt-survey tool allows testing multi-threaded metadata operations directly on the MDS without the use of Lustre clients so that it is possible to measure performance of the PDO feature even with a single MDS server. Tests should be run that separately create, lookup+stat, and lookup+unlink a fixed total number of files under a single directory with M={1,2,4,...,1024} threads. Between each of the phases, the MDT filesystem should be unmounted to flush the cache so that the performance of the ldiskfs filesystem is being measured (where PDO is implemented) and not the higher-level cache.

Verify PDO performance improvement by mdtest or mdsrate.

The mdtest or mdsrate benchmarks should be able to verify performance improvement with the whole filesystem stack, including clients. Benchmark runs should create/lookup+stat/lookup+unlink a fixed total number of files under a single directory with M={256,512,1024} MDS service threads and N={1,1024} clients (M <= N). These tests can also be repeated with C={0,1,2,4} stripes per file to measure the impact the stripe count has on PDO.

Verify no performance regression for any other cases by mdtest or mdsrate.

We can use mdtest to verify whether there is performance regression for operations under small (non-indexed) directories, also we can verify whether there is any extra overhead for htree-lock by running mdtest with single thread in a single directory.

Verify no functionality regression by acc-small.

Acceptance Criteria

PDO patch will be accepted as properly functioning if:

1. Improved parallel create / remove performance under large shared directory for narrow stripe files (0 – 4) on multi-core server (8+ cores)
2. No performance regression for other metadata performance tests
3. No functionality regression

Glossary

Ldiskfs: ldiskfs is an ext4 filesystem with additional patches applied specifically to enable use as a backend Lustre filesystem and to improve performance.