Bypass VOL:
A Sketch Design

John Mainzer
Lifeboat, LLC

11/25/22

Introduction

To provide a significant speedup, the bypass VOL requires HDF5 support for multi-threaded VOLs. As design work for the necessary changes to the HDF5 library is still in progress, no more than a design sketch of the bypass VOL is possible at this point. While a more detailed design will be necessary, this sketch allows us to lay out the general structure, and start to identify potential problem areas.

The immediate objective of the Bypass VOL is to allow multiple threads to perform concurrent I/O on a limited subset of HDF5 data sets – specifically either contiguous or chunked data sets of scalar type. In the case of chunked data sets, filtering is disallowed – at least in the initial versions.

However, as the first multi-thread VOL connector, it will serve a second purpose – it will be the initial use case and test vehicle for the HDF5 library modifications required to support multi-threaded VOL connectors.

Conceptual Overview

To date, there have been terminal VOL connectors, and pass-through VOL connectors. Terminal VOL connectors handle I/O to the underlying storage system, and pass-through VOL connectors operate on I/O requests, but don't do any actual I/O. Current examples of pass-through VOL connectors include the Asynchronous VOL that attempts to perform I/O requests in the background, and the Cache VOL, that tries to cache data set data so as to accelerate perceived I/O speed.

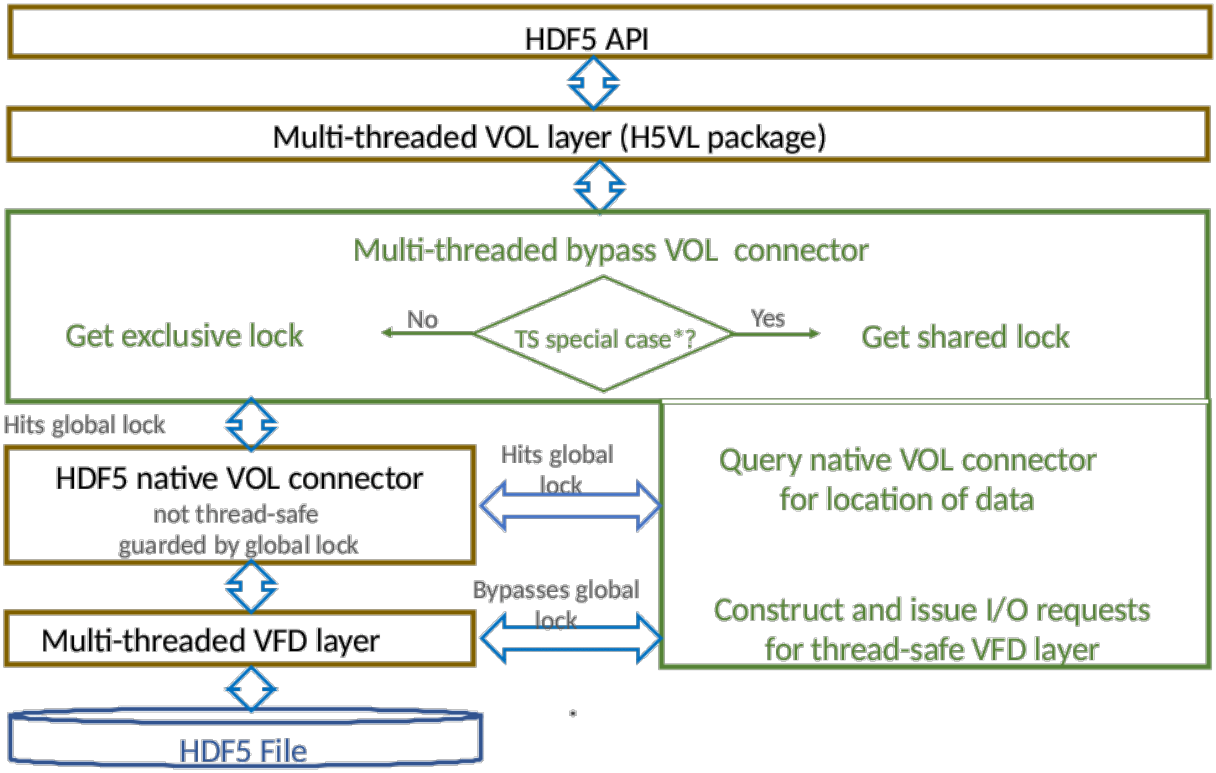
The Bypass VOL will be a hybrid of these two types. API calls for which multi-thread support is not available, are simply passed to the Native VOL for execution. However, for the target data set I/O calls, it will allow multiple calls to proceed simultaneously. As currently conceived, these calls will query the native VOL for the location of the target raw data – which will require

exclusive access to the native VOL. However, once this data is obtained, exclusive access to the native VOL is no longer required, and I/O operations can continue concurrently.

Absent complex selections or metadata not in the metadata cache, this lookup should be quick. The complex selection limitation will be relaxed when a multi-thread safe version of H5S is implemented. The cache miss problem can't be solved in general, although use of paged allocation and page buffering in the native VOL should help greatly in many if not most cases.

Cycle of Operation

The cycle of operation of the Bypass VOL is summarized in the following diagram:



The Bypass VOL Connector examines each API call as it is received.

If the Bypass VOL connector doesn't support multi-threaded execution of the API call in question, the call must obtain an exclusive lock[1] before proceeding. When this lock is obtained, the API call is routed to the native VOL. The exclusive lock is dropped when the API call returns.

1 This exclusive lock must be recursive to accommodate HDF5 API callbacks that may invoke additional HDF5 API calls. Since the exclusive lock must be held throughout the execution of these callbacks, any such API calls will be executed sequentially even if multi-thread operation is normally supported for the call in question.

If multi-threaded operation is supported for this API call, it must obtain the shared lock before proceeding. This allows multiple multi-thread operations to proceed concurrently, while preventing execution of any non-multi-thread enabled API calls from executing during multi-thread operations.

If there have been any non-multi-thread writes since the last multi-thread operation completed, in general it will be necessary to send a flush command to the native VOL – which will hit the Native VOL's global lock, and may take a while. However, this should not be necessary for the first version of the Bypass VOL.

The exact processing from this point depends on the nature of the multi-thread support, but for purposes of this discussion let us restrict ourselves to the initial target of supporting multi-threaded I/O on un-filtered chunked and contiguous data sets of numerical type, without type conversions, variable length data, or references. Given this presumption, processing proceeds as follows:

1. Query the Native VOL Connector to obtain the offset and extent of the target raw data from the target data set in the HDF5 file. This query will hit the Native VOL Connector's global lock, but it should return quickly unless there is a metadata cache miss. It will also be necessary to obtain a pointer to the instance of H5FD_t used by the top level VFD – although this can be done just once at file open.

2. Given the above data, construct VFD I/O call(s) to perform the desired file I/O. Until H5S (selections) is made thread safe, these calls must be either the regular POSIX like calls, or vector I/O calls. Otherwise, the VFD layer would have to call the non-thread safe H5S package to walk the selection, and thus hit the Native VOL's global lock repeatedly.

3. Make the required H5FD I/O call(s). This requires that the VFD layer and the relevant VFDs be thread safe – allowing these calls to bypass the Native VOL's global lock, and thus allowing an arbitrary number of I/O calls to proceed concurrently.

4. On completion, drop the shared lock and return to the caller.

Much of the above functionality has already been implemented in a hybrid VOL used for the initial development and performance testing of the sub-filing VFD.

Note that the shared / exclusive lock mentioned above is different from the global lock in the HDF5 library. While it serves a different purpose, functionally speaking, it may be thought of as a R/W lock.

By the time we are able to test the Bypass VOL, may of the services supplied to VOL connectors by the HDF5 library will have been made thread safe, and the HDF5 global lock will have been pushed down accordingly – thus allowing an initial level of multi-thread operation in VOL

connectors that support it. This will improve as more services (i.e. H5S) are made multi-thread safe.

Finally, observe that most of the differentiation between API calls that either do or don't support multi-thread operation will be performed by the VOL layer proper (i.e. H5VL). Only for H5Dread/write calls will it be necessary to query the native VOL to see if multi-thread access is supported for the target data set.

Known Issues

While it is difficult to offer a more detailed design before the details of HDF5 multi-thread VOL connector support are filled out, there are a number of issues that are visible at this point. While none of these appear to be insurmountable, brief discussions along with proposed solutions will likely be useful, as they will influence the details of multi-thread VOL connector support.

Potential Lock Contention

Determining whether a H5Dread/write call is eligible for multi-thread execution will require several HDF5 API calls, with several more to determine the location of the target data in the HDF5 file. At least as HDF5 is currently constituted, each of these calls will have to gain and drop the HDF5 library global mutex.

In addition to the obvious lock overhead issue, there is a potential lock contention issue.

To see this, suppose a large number of such H5Dread/write calls are made simultaneously. Assuming that the HDF5 global lock is reasonably fair, none of these calls would start its I/O until all of them were largely through their preparatory HDF5 API calls. Depending on lock overhead and possible metadata cache misses, an extended delay before any I/O commences is possible.

The obvious solution to this is to do all the preparatory HDF5 API calls for one H5Dread/write call first, then the next, and so on. This would maximize use of the I/O channel, and hide the costs of the preparatory calls once this channel is saturated.

To do this, we need to allow a thread in the Bypass VOL to acquire the HDF5 global lock, run the HDF5 API calls necessary to determine whether a H5Dread/write call can run multi-thread, and if so, run the additional calls necessary to determine the location of the target data, and then drop the HDF5 global lock.

This is not supported at present. However, it may be convenient to add this facility as part of the multi-thread VOL connector upgrades. Indeed, it may be required, as otherwise, non-multi-thread safe VOL connectors will need to provide their own mutexes to avoid multi-thread operation.

Raw Data Cache Coherency

Depending on circumstances, the HDF5 library caches significant amounts of raw data. For example, chunked data sets have a 1 MB cache by default, and small contiguous data sets can be cached in the page buffer (in the serial case only).

This is not a problem as long as all I/O to any given data set flows through either the Native VOL, or the Bypass VOL, but not both. However, if this is not the case, the possibilities for corruption should be obvious.

In the vast majority of cases, this should not be a problem. But recall that there are HDF5 API calls that invoke user supplied callbacks that may contain arbitrary HDF5 API calls. As mentioned above, these secondary HDF5 API calls must be executed sequentially – and thus at first glance could be passed to the native VOL for execution even if they would normally be executed via the multi-thread branch in the Bypass VOL.

However, this should be avoided due to the above-mentioned cache coherency issues. If it is not unavoidable, the file will have to be flushed before any multi-thread operations commence.

Metadata Cache Miss Problem

Determining whether multi-thread operations is supported for a H5Dread/write call, and if so, where the required raw data resides in the file will hit the HDF5 metadata cache. If this data is not in the metadata cache, the H5Dread/write call (and all other pending HDF5 calls) must wait while the required metadata is fetched from file and decoded. Obviously, under certain scenarios, there is significant potential for performance degradation.

While this is a HDF5 Native VOL issue that can't be addressed from the Bypass VOL, it should be mentioned if only indicate that there are solutions – albeit at the user level.

The first of these is to specify paged allocation at file creation, and to enable the page buffer at file open. This will cause metadata to be loaded in pages – thus reducing the number of metadata read operations.

The problem can be further mitigated through appropriate configuration of the metadata cache.

The particulars of these solutions will be file and use case specific, and thus there is little point in going into further detail here. The take away should be that there are existing ways of mitigating this problem.