Making H5VL Multi-Thread Safe:
A Sketch Design

John Mainzer
Lifeboat, LLC
Neil Fortner
The HDF Group

6/6/24

Introduction

Conceptually, H5VL is fundamentally a straight forward module. Its purpose is to route HDF5 API calls to the appropriate VOL connector for execution. Thus, the majority of its multi-thread issues are inherited either from its dependencies or from the underlying VOL connectors.

It will be impossible to fully address the multi-thread issues of the VOL Layer (H5VL) until the multi-thread issues of the supporting packages are well understood. Even then, much effort will be required to move the global mutex down into the VOL Layer (and possibly into the VOL connectors) to allow the desired multi-thread operation in VOL connectors that support it.

Thus, for now, this document is little more than a list of known issues, combined with some musings on how the mutex issue might be addressed. The purpose of this initial development is to allow concurrent access from the API to the VOL interface in order to support multi thread safe VOL connectors. This document therefore focuses on H5VL code above the actual VOL interface, thus code in H5VLnative*.c and H5VLpassthru.c will not be considered for now and will be covered by the global mutex.

A Quick Overview of H5VL

In essence, H5VL is a switch board relaying HDF5 API calls to the appropriate VOL connector, or in some cases, relaying the API calls through the specified stack of VOL connectors. While it is similar to the VFD layer (H5FD) in certain ways, it is vastly more complex due to much richer HDF5 API. Further, for objects that are implemented different ways between different VOL connectors, it must perform the necessary wrapping and unwrapping so that the different variations may be stored using the same utilities (e.g., H5I).

At present, the HDF5 global mutex is acquired on entry to any HDF5 API call. As our objective is to relax this requirement for VOL connectors (or portions of VOL connectors) that are multi-thread safe, H5VL will have to be modified to support this. The details of a clean, maintainable solution for this problem are unknown pending multi-thread revisions for the minimal set of packages required for initial functionality.

While the management of the HDF5 global mutex must be put to one side for now, a brief discussion of the multi-thread issues surrounding the VOL layer will help direct the multi-thread modification of the packages that H5VL depends on.

Multi-Thread Issues in H5VL

From our initial review, it seems that three types of multi-thread issues in H5VL: use of other, potentially non-multi-thread safe packages, issues internal to H5VL proper, and the need to acquire the global mutex whenever directing control flow into a VOL connector that is not multi-thread safe.

Use of other HDF5 packages by H5VL

H5VL makes use of the many packages, which are best discussed in the following groups:

- H5MM
- H5FL
- H5E
- H5I
- H5P
- H5CX

As usual, issues with H5MM and H5FL may be avoided by using C memory allocation functions directly, and by either not maintaining free lists, or doing so internally.

Once they are made multi-thread safe, calls to H5E, H5I, and H5P are non-issues, as long as H5VL can refrain from holding locks during these calls. However, care must be taken that, when multiple calls are made, they are made in such a way that is multi-thread safe. There are also calls to H5I_iterate() in H5VLint.c, which should be replaced with more thread friendly calls once these are added to H5I.

Some of the not obviously multi-thread safe H5CX API calls (e.g., H5CX_retrieve_state / restore_state()) are used. More study of H5CX is needed to see if these functions can be made multi-thread safe.

In addition, there are several utility packages that not on our list for immediate conversion to multi-thread safety.

- H5PL
- H5ES
- H5S
- H5SL
- H5T

The first three of these (H5PL, H5ES, and H5S) are all on our list for conversion as time and resources permit, but are not seen as critical for development of initial multi-thread VOL connector support. Calls into these packages will have to be protected by the global mutex to exclude the possibility of multi-thread processing in these packages pending conversion.

In contrast, H5SL (skip lists) may never be converted to multi-thread safety due to the performance issues that have dogged it. Instead, it will almost certainly be replaced with some other thread safe data structure. In H5VL, it is used in the registration / deregistration of dynamic operations. Unless this proves to be a common and costly operation, this will probably be protected by the HDF5 global mutex and left for another day.

The H5T calls are related to committed datatypes, and are therefore unlikely to be called often, and simply taking the global mutex for these calls should not be a major hindrance. Some of these calls could be easily made multi-thread safe, namely H5T_get_named_type() and H5T_already_vol_managed(), but H5T_construct_datatype() would be more difficult and will likely remain protected by the global mutex for the foreseeable future.

Finally, there are the many HDF5 native VOL calls that appear in the H5VLnative*.c files.

- H5A
- H5AC
- H5C
- H5D
- H5F
- H5G
- H5HG
- H5M
- H5O
- H5PB
- H5T

These will have to be protected by the HDF5 library global mutex to preclude multi-thread execution. While this is conceptually straight forward, the number and complexity of these calls present a challenge – whose solution is still very much undecided.

H5T is listed in both of the last two categories because it contains both calls related to VOL objects and calls related to purely in-memory objects. This is due to the mixed nature of committed datatypes.

Multi-thread thread issues in H5VL proper

A review of the H5VL public, private, and developer APIs reveals the following multi-thread safety issues in the current H5I implementation.

Global Variables:

H5VL uses global variables for convenience to store the ID assigned to the native and passthrough VOLs (H5VL_NATIVE_ID_g and H5VL_PASSTHRU_g). These variables are used to note whether the target VOL connectors have been registered, and if so, to store their IDs.

This isn't a major issue, but to avoid the possibility of duplicate registrations, some sort of mutual exclusion is needed. Since these are managed by the VOL connectors themselves (in H5VL_native_register() and H5VL_pass_through_register()), and only visible inside those connectors' source files, making these routines take the global mutex is to be expected.

There are also the H5VL_native_cls_g and H5VL_pass_through_g global tables, but as they are constant, they are not a multi-thread issue.

H5I_inc_ref() Ordering

Some routines in H5VL obtain a H5VL_class_t connector object using H5I_object_verify() and then increment the reference count on that ID using H5I_inc_ref(). This introduces a potential race condition if the ID's reference count is decremented by another thread between these operations. While this may only be possible due to an application or library bug, we should still try to handle such an error more gracefully. We can simply switch the orders of these operations and add a memory fence in between them. In this way, if the connector is closed by another thread, it will simply throw an error when H5I_inc_ref() cannot find the ID, instead of causing memory errors and potential security issues.

VOL Connector and Object Reference Counting

Both VOL connectors, using the H5VL_t struct, and VOL objects, using the H5VL_object_t struct, are reference counted to allow being accessed by multiple owners. Since these objects can be shared by multiple threads, these reference counting schemes must be made multi-thread safe. This can be done by converting the reference count to an atomic variable and by manipulating this reference count using atomic operations.

VOL Connector Registration and Deregistration:

We believe that, since the H5VL_class_t VOL struct is wrapped by a reference counted ID, we will not need to protect VOL connector registration and deregistration with a mutex, except as needed for dependencies like H5PL, provided we are careful to keep the reference count correct at all times and order accesses appropriately. The underlying, non reference counted H5VL_class_t struct is cached in the H5PL package, but that should not be an issue for the reasons discussed below.

We will want to move the H5I_inc_ref() from H5VL__register_connector_by_class/name/value() to H5VL__get_connector_cb(), so that we take a reference to the ID as soon as we store it in the op_data struct, so it is not freed before the calling function has a chance to use it. We will rely on H5I_iterate() being made thread safe (or taking the global mutex) in a way that protects the ID from deletion while the callback is made. This will require adding a matching H5I_dec_ref() call to places that use H5VL__get_connector_cb() but don't currently call H5O_inc_ref(), like H5VL__peek_connector_id_by_name/value(), and modifying H5VL__get_connector_id_by_name/value() to avoid calling the peek functions and instead call H5I_iterate() directly.

VOL connectors are registered and associated with an ID in H5VL__register_connector() in H5VLint.c, and the primary place where it is closed is in H5VL_conn_dec_rc(). Several other places manipulate the reference count of this ID, including H5VLclose() and H5VLunregister_connector(), registering an already registered connector, VOL object registration and close, and querying the connector of a VOL object. Since these simply make use of H5I functions, and these H5I functions are to be made thread safe, there should be no changes necessary in these places in the H5VL package to support thread safe connector registration and deregistration, except for the aforementioned move of H5I_inc_ref().

File Open

The file open routines in H5VLcallback.c, primarily H5VL_file_open(), are handled differently from other routines, since they need to set up the connector in addition to making the connector callback and creating the VOL object. There is a potential thread safety issue here where, if the library needs to search the plugin list for a usable VOL connector, the plugin search callback function H5VL__file_open_find_connector_cb() properly registers a new ID for the connector using H5VL__register_connector_by_class(), which copies the class struct, but also returns a point to the original H5VL_class_t class struct via which is not reference counted by the ID nor owned by the calling function in any way. This struct is returned via udata→cls. While this class struct points into the plugin cache which is not currently emptied until the HDF5 library is closed, we should change this code to only return the new ID, and use the VOL class associated with that ID to open the file in order to properly track ownership and prevent potential future issues if we allow plugin cache evictions in the future.

VOL Wrap Contexts

It is the responsibility of the VOL connector author to ensure that everything under the VOL layer is thread safe, but one potential non-obvious issue involves VOL wrap contexts which are used by passthrough VOL connectors. While these are stored in the API context, which is thread specific, the connector must make sure it either returns a separate wrap context on each thread, or the wrap context it returns is properly reference counted and capable of concurrent access (within the connector's wrap_object callback).

Dynamic Operations

The VOL interface contains, in H5VLdyn_ops.c, a facility for dynamically registering arbitrary optional operations, and this must be made thread safe. The implementation consists of a global array (H5VL_opt_ops_g) of pointers to skip lists, one for each object type ("subclass"). These skip lists are indexed by the user-supplied operation names and contain integer operation values. The skip lists are created and added to the array dynamically in H5VL__register_opt_operation(), so we must make sure to do so in a thread safe manner, possibly with a compare and swap operation: check if the slot is NULL, if it is then create the skip list, compare and swap with the slot (compare to NULL), if it fails free the skip list and load the skip list from the slot. These optional operations are currently never freed until the library is shut down, so we should not need to worry about the thread safety of the free operation, but this is a good candidate for adding a virtual lock assertion (see below).

VOL Connector Interface

In order to support older VOL connectors which were not designed with thread concurrency in mind, we will need to provide a way for VOL connectors to report if they are multi-thread safe. To do this, we can simply use the existing VOL capability flag H5VL_CAP_FLAG_THREADSAFE which is to be set to 1 if thread concurrency is supported. The H5VL layer will acquire the global mutex before making a connector callback for a connector that does not have this flag set. Connectors that are only partially multi-thread safe, either on a per-callback basis, or at lower levels of code, can set this flag and take the HDF5 global mutex using H5TSmutex_acquire() as needed.

Appendix

Virtual Lock Assertion

In some cases, such as when operating on data found in the API context, when operating on the native VOL connector, or when operating on the global optional operations registry, we can assume that certain operations will never happen concurrently, and therefore do not need to implement thread safety for these operations. We would like to document these assumptions in a manner analogous to how assumptions in serial code are documented with assertions. One way to do this would be to implement a virtual read-write lock on the object, which is only compiled in when assertions are enabled, which is to be taken in read mode when accessed by thread safe code and write mode when accessed by non thread safe code. However, instead of blocking in case of a conflict, it will cause an assertion failure. This could be implemented using custom code, or with a real read-write lock and using try_lock() instead of lock(). We may also want to add an option to assert that a block of memory was is modified while a thread holds a virtual read lock.

There are also cases where proper improper usage of the API will cause non thread safe behavior that would be impossible otherwise. One example is if one thread closes the last reference to an ID while another thread is in the middle of an HDF5 operation that uses that ID. The proper way to handle this situation is for each thread to take a reference to the ID and close their reference when no it is no longer needed by that thread. We may want to give an option to enable the use of the virtual lock described above to detect these situations, though it would be enabled via a property list instead of the status of NDEBUG, and would throw an HDF5 error instead of an assertion. This option should be useful to help application developers find threading bugs in their own code.

No Modifications Needed

These are locations in the H5VL code that initially appear to be thread safety issues but under closer inspection we believe they are not going to need modification.

Wrap Context Reference Counting

Since wrap contexts are stored in the API context, and each thread maintains its own API context, there should not be any changes necessary to support concurrent thread safe access to wrap contexts. This includes wrap context reference counting, which can continue to use non-atomic operations.

H5PL (Plugin) Cached Connectors

The H5PL layer is used to load external connectors located in dynamically loaded libraries. These loaded connectors are then cached in the H5PL layer as non-reference counted H5VL_class_t structs. Since they are not reference counted, this may appear to be a thread safety issue, but because these connectors are never freed until HDF5 library shutdown, we do

not expect this to be an issue, but may wish to add an assertion to alert future developers to this potential issue if they attempt to implement H5PL cache evictions.

Connector Property Struct

The connector property struct, H5VL_connector_prop_t, is not reference counted. However, this struct is only used to store connector info in a property list, and as such is always owned by the property list it resides in, and therefore as long as access to the property list is made threadsafe (and the property list ID is properly reference counted), there should be no thread safety issues involving this struct.

H5VL_object_is_native()

H5VL_object_is_native() initially appears to have a non thread safe code pattern, since it retrieves one (non reference counted) VOL class then another, then compares them. However, since it is passed a reference counted H5VL_object_t struct for the first object, we can assume the caller has a reference to that object and its connector has not been freed, and we can also assume the native connector, which is the second VOL class, has not been freed. In addition, as outlined above, H5VL_class_t structs are currently never freed until the library is shut down.

H5VL_file_is_same()

H5VL_file_is_same() follows a similar path of reasoning as H5VL_object_is_native(), except it compares the VOL connectors of two arbitrary objects instead of comparing one to the native connector. Similarly, this is not an issue for thread safety for the same reasons as above.