Adapting HDF5's H5S interface to multi-threaded environments

Jordan Henderson

This document gives an overview of the HDF5 library's H5S interface for dataspace objects, outlines the concurrency issues with this interface and proposes some solutions for allowing concurrent operations involving HDF5 dataspaces.

Contents

1. Introduction **6**

2. The H5S interface **6**
2.1. Dataspace object extents . . . 6
2.1.1. 'Scalar' extent . . . 6
2.1.2. 'Simple' extent . . . 6
2.1.3. 'Null' extent . . . 6
2.2. Dataspace object selections . . . 6
2.2.1. 'All' selection . . . 7
2.2.2. 'None' selection . . . 7
2.2.3. 'Hyperslab' selection . . . 7
2.2.4. 'Point' selection . . . 7
2.3. Dataspace selection iterators . . . 7

3. Issues with concurrency in the H5S interface **8**
3.1. Concurrent manipulation of dataspace objects . . . 8
3.1.1. Concurrent read-write operations on shared dataspace object . . . 8
3.1.2. Concurrent write modifications to shared dataspace object . . . 9
3.2. Sharing of internal selection information between dataspace objects . . . 9
3.3. Dataspace selection iterators . . . 10
3.4. Usage of free lists in the H5S interface . . . 11
3.5. Concurrency issues specific to 'All' selections . . . 11
3.6. Concurrency issues specific to 'None' selections . . . 11
3.7. Concurrency issues specific to hyperslab selections . . . 11
3.7.1. Rebuilds of hyperslab dimension information . . . 11
3.7.2. Concurrent updates of hyperslab selection span 'op generations count' . . . 11
3.8. Concurrency issues specific to point selections . . . 12

4. Current state of concurrency in the H5S interface **12**
4.1. Current state . . . 12
4.2. Concurrent closes on shared dataspace IDs . . . 15
4.3. Concurrent modification of selection or extent in shared dataspace . . . 16
4.4. Miscellaneous bugs in HDF5 . . . 17
4.5. RCU-like approach proposal . . . 17
4.5.1. Overview . . . 18
4.5.2. Implementation . . . 18
4.5.3. Note on RCU approach with multiple writers to shared structure . . . 20

A. Appendix: H5S interface public API functions **22**
A.1. `H5Sclose()` . . . 22
A.2. `H5Scombine_hyperslab()` . . . 23
A.3. `H5Scombine_select()` . . . 23
A.4. `H5Scopy()` . . . 24
A.5. `H5Screate()` . . . 24

A.6. H5Screate_simple() . . . 24
A.7. H5Sdecode() . . . 25
A.8. H5Sencode2() . . . 25
A.9. H5Sextent_copy() . . . 26
A.10. H5Sextent_equal() . . . 26
A.11. H5Sget_regular_hyperslab() . . . 27
A.12. H5Sget_select_bounds() . . . 27
A.13. H5Sget_select_elem_npoints() . . . 27
A.14. H5Sget_select_elem_pointlist() . . . 28
A.15. H5Sget_select_hyper_blocklist() . . . 28
A.16. H5Sget_select_hyper_nblocks() . . . 28
A.17. H5Sget_select_npoints() . . . 29
A.18. H5Sget_select_type() . . . 29
A.19. H5Sget_simple_extent_dims() . . . 29
A.20. H5Sget_simple_extent_ndims() . . . 29
A.21. H5Sget_simple_extent_npoints() . . . 30
A.22. H5Sget_simple_extent_type() . . . 30
A.23. H5Sis_regular_hyperslab() . . . 30
A.24. H5Sis_simple() . . . 30
A.25. H5Smodify_select() . . . 30
A.26. H5Soffset_simple() . . . 31
A.27. H5Ssel_iter_close() . . . 31
A.28. H5Ssel_iter_create() . . . 31
A.29. H5Ssel_iter_get_seq_list() . . . 32
A.30. H5Ssel_iter_reset() . . . 32
A.31. H5Sselect_adjust() . . . 32
A.32. H5Sselect_all() . . . 33
A.33. H5Sselect_copy() . . . 33
A.34. H5Sselect_elements() . . . 33
A.35. H5Sselect_hyperslab() . . . 34
A.36. H5Sselect_intersect_block() . . . 34
A.37. H5Sselect_none() . . . 35
A.38. H5Sselect_project_intersection() . . . 35
A.39. H5Sselect_shape_same() . . . 35
A.40. H5Sselect_valid() . . . 35
A.41. H5Sset_extent_none() . . . 36
A.42. H5Sset_extent_simple() . . . 36

B. Appendix: H5S private functions **37**

C. Appendix: H5S interface data structures **38**
C.1. H5S_t . . . 38
C.2. H5S_extent_t . . . 38
C.3. H5S_select_t . . . 39
C.4. H5S_hyper_sel_t . . . 40
C.5. H5S_hyper_span_info_t . . . 40

C.6. H5S_pnt_list_t . . . 41
C.7. H5S_pnt_node_t . . . 42
C.8. H5S_sel_iter_t . . . 42

D. Appendix: Fields modified by API functions 44
D.1. 'Get'-style functions . . . 44
D.1.1. H5Screate() . . . 44
D.1.2. H5Screate_simple() . . . 44
D.1.3. H5Scombine_hyperslab() . . . 44
D.1.4. H5Scombine_select() . . . 44
D.1.5. H5Sdecode() . . . 44
D.1.6. H5Sencode2() . . . 45
D.1.7. H5Sextent_equal() . . . 45
D.1.8. H5Sselect_intersect_block() . . . 45
D.1.9. H5Sselect_project_intersection() . . . 45
D.1.10. H5Scopy() . . . 45
D.1.11. H5Sget_regular_hyperslab() . . . 46
D.1.12. H5Sget_select_bounds() . . . 46
D.1.13. H5Sget_select_elem_npoints() . . . 46
D.1.14. H5Sget_select_elem_pointlist() . . . 46
D.1.15. H5Sget_select_hyper_nblocks() . . . 46
D.1.16. H5Sget_select_hyper_blocklist() . . . 46
D.1.17. H5Sget_select_npoints() . . . 46
D.1.18. H5Sget_select_type() . . . 47
D.1.19. H5Sget_simple_extent_ndims() . . . 47
D.1.20. H5Sget_simple_extent_dims() . . . 47
D.1.21. H5Sget_simple_extent_npoints() . . . 47
D.1.22. H5Sget_simple_extent_type() . . . 47
D.1.23. H5Sis_regular_hyperslab() . . . 47
D.1.24. H5Sis_simple() . . . 47
D.1.25. H5Sselect_shape_same() . . . 48
D.1.26. H5Sselect_valid() . . . 48
D.2. 'Set'-style functions . . . 48
D.2.1. H5Sclose() . . . 48
D.2.2. H5Sextent_copy() . . . 48
D.2.3. H5Smodify_select() . . . 48
D.2.4. H5Soffset_simple() . . . 49
D.2.5. H5Sselect_adjust() . . . 49
D.2.6. H5Sselect_all() . . . 49
D.2.7. H5Sselect_copy() . . . 49
D.2.8. H5Sselect_elements() . . . 49
D.2.9. H5Sselect_hyperslab() . . . 50
D.2.10. H5Sselect_none() . . . 50
D.2.11. H5Sset_extent_none() . . . 50
D.2.12. H5Sset_extent_simple() . . . 50

D.3. Other functions 51
D.3.1. H5Ssel_iter_create() 51
D.3.2. H5Ssel_iter_get_seq_list() 51
D.3.3. H5Ssel_iter_reset() 51
D.3.4. H5Ssel_iter_close() 51

1. Introduction

To begin adapting HDF5's H5S interface for concurrency, an initial investigation was made into the internals of the H5S code to enumerate all of the issues to be addressed. This document lists the issues that were found and proposes an RCU-like (read-copy-update) approach for dealing with the majority of the issues. The initial implementation of this approach will focus more on correctness than performance and will likely need to be optimized later as issues arise.

2. The H5S interface

HDF5's H5S interface contains functions for manipulating HDF5 dataspace objects. These objects consist of two main components, the extent, or dimensionality, of the dataspace object and a selection on the data elements of the dataspace within that extent.

2.1. Dataspace object extents

There are currently three different types of extents that a dataspace object may have: 'scalar', 'simple' and 'null'.

2.1.1. 'Scalar' extent

A dataspace with a scalar extent is comprised of a single data element and is considered to be dimensionless.

2.1.2. 'Simple' extent

A dataspace with a simple extent is comprised of up to 32 orthogonal, evenly spaced dimensions of data elements, with each dimension currently being able to contain up to a `hsize_t` (currently defined as `uint64_t`) number of data elements. Each dimension must be specified with an initial number of data elements and a maximum number of data elements, which could be greater than or equal to the initial number of elements, or `H5S_UNLIMITED` to allow for the possibility of that dimension growing larger than its initial size.

2.1.3. 'Null' extent

A dataspace with a null extent contains no data elements and is considered to be dimensionless.

2.2. Dataspace object selections

A selection can be made on the elements in the extent of a dataspace object to specify which of those elements should participate in operations that the dataspace object is used for. There are currently four different types of selections that can be made on a dataspace object: 'all', 'none', 'point' and 'hyperslab'. Each type of

selection is implemented in a separate file in HDF5's source code by using a system of callbacks that code in the H5S interface utilizes.

2.2.1. 'All' selection

This selection type is set on a dataspace object with the `H5Sselect_all()` function and signifies that all elements in the dataspace object's extent are selected for operations involving dataspaces. This is the default selection within a newly-created dataspace object.

2.2.2. 'None' selection

This selection type is set on a dataspace object with the `H5Sselect_none()` function and signifies that none of the elements in the dataspace object's extent are selected for operations involving dataspaces.

2.2.3. 'Hyperslab' selection

This selection type is set on a dataspace object with the `H5Sselect_hyperslab()` function and allows selecting of a region of elements in the dataspace according to four array parameters: `start`, `stride`, `count` and `block`. Each of these arrays must have the same size (in terms of array elements) as the dimensionality of the dataspace object.

The `start` array contains the offset, in each dimension, of the region of elements to select.

The `stride` array contains the number of elements to move, in each dimension, from the offset specified in `start` when selecting elements. For example, a `stride` value of 1 moves to the next element in that dimension, while a `stride` value of 2 moves to every other element in that dimension.

The `count` array contains the number of blocks (specified in the `block` array) to select in the dataspace, in each dimension.

The `block` array specifies the size of the blocks of elements being selected, in each dimension. For example, a block size of 1 selects blocks of single elements in that dimension, while a block size of 2 selects blocks of two elements in that dimension.

2.2.4. 'Point' selection

This selection type is set on a dataspace object with the `H5Sselect_elements()` function and is used to select individual data elements in the dataspace object's extent. Point selections are currently implemented using a simple linked list structure (see C.6).

2.3. Dataspace selection iterators

The H5S interface also includes functions for creating and using an iterator object which iterates over a selection within a dataspace and returns sequences of offset/length pairs. Dataspace iterators can be created and manipulated with the functions `H5Ssel_iter_create()`, `H5Ssel_iter_get_seq_list()`, `H5Ssel_iter_reset()` and `H5Ssel_iter_close()`.

3. Issues with concurrency in the H5S interface

The following sections detail the general concurrency issues observed in HDF5's H5S interface. Concurrency issues specific to a particular public API function are listed in Appendix A. Concurrency issues specific to a particular internal H5S function are listed in Appendix B. Concurrency issues specific to a particular data structure in the H5S interface are listed in Appendix C.

Note that requiring HDF5 dataspace objects to be thread-local provides a solution to nearly all of the concurrency issues covered in the following sections. However, that approach has trade-offs, such as: minimal benefit beyond thread-safety, increased memory usage and the potential for bad performance if dataspaces need to be copied to each thread. Therefore, this document is written with the assumption that some level of concurrent access to dataspace objects is desired.

3.1. Concurrent manipulation of dataspace objects

If the chosen approach to concurrency in the H5S interface allows multiple threads to act on the same dataspace object, then there are two primary concerns around concurrent manipulation of dataspace objects. One concern is determining how to handle modifications to a dataspace object while that dataspace is being used by another thread for a read-like operation. Another concern is determining how to handle the ordering of operations when multiple threads wish to perform modifications to the same dataspace object concurrently.

3.1.1. Concurrent read-write operations on shared dataspace object

When a dataspace object is being accessed concurrently, there is currently the potential for the dataspace object's extent or selection to change while the dataspace is being used by another thread. For a first estimation, this use case doesn't appear as though it would be that interesting, so a more heavy-handed solution, such as a reader-writer lock, may be acceptable for controlling concurrent access to these components of a dataspace. However, other solutions should be explored here.

One issue with allowing this behavior is that certain combinations of manipulations may be fundamentally incompatible with each other when concurrency is involved and these cases may be too numerous to enumerate. For example, consider this sequence of operations:

1. Thread 1 calls `H5Sget_select_type()` to determine what type of selection is set on a dataspace object and is returned the value `H5S_SEL_POINTS`, signifying that a point selection is set on the dataspace.
2. Thread 1 is preempted and Thread 2 calls `H5Sselect_hyperslab()`, changing the selection in the dataspace object from a point selection to a hyperslab selection.
3. Thread 2 is preempted and Thread 1 calls `H5Sget_select_elem_npoints()` to retrieve the number of points selected in the dataspace object, but receives an error due to the selection in the dataspace object not being a point selection

Without a re-design of some H5S API routines to perform various combinations of H5S operations atomically, it appears that either these situations will have to be documented as invalid to perform concurrently or some user-level coordination ability may be needed.

3.1.2. Concurrent write modifications to shared dataspace object

When a dataspace object is being modified concurrently, there is currently the potential for conflicting accesses to result in an indeterminate extent or selection for the dataspace, depending on the manipulation being performed.

For the selection portion of a dataspace object, there are essentially two cases to be dealt with. The first case is multiple threads attempting to set different types of selections on a dataspace object. Initial thoughts suggest that this is an unreasonable use case and should be prevented or documented against. The second case is multiple threads attempting to set or refine the same type of selection on a dataspace object with their own selection parameters. In this case, it seems that some order of operations may need to be imposed to retain a coherent view of the selection. Further, some coordination between threads will likely be needed to ensure that all modifications on the dataspace selection have been performed before that dataspace is used for a following data operation. Similar to what was considered for HDF5's H5P interface, it may make sense to have a "finalize" type of operation for this case. Note that this second case results in every thread having the same selection in the dataspace, so the data operations performed by threads will be similar. Therefore, this may primarily be useful as a way of parallelizing the formation of the overall selection in the dataspace.

Though it is estimated that this will not be a very common use case, concurrent modifications to a dataspace's extent presents a dangerous situation to be protected against. If a dataspace's extent changes between threads, this presents the opportunity for buffer overflows if the dataspace cannot be both changed **and** used in an atomic manner together before another thread "takes control" of the dataspace. Note that this is assuming that the dataspace object is to be used in separate data operations on each thread. If the dataspace is only to be eventually used for a single data operation on one or multiple threads simultaneously, then initial thoughts suggest that modifications to the dataspace's extent by any thread other than the "last" are effectively useless.

3.2. Sharing of internal selection information between dataspace objects

When making an internal copy of a dataspace object with `H5S_copy()`, the caller must pass `true` or `false` for its boolean parameter `share_selection`. If specified as `true`, this signifies to the function that the selection in the source dataspace being copied can be directly shared with the destination dataspace. This flag is utilized by hyperslab and point selections and, to the best of knowledge at the time of writing, this is simply an optimization that allows skipping of a potentially very expensive copying operation. The `H5S_copy()` function mentions that the only time `true` should be passed for this parameter is in the case where the selection in the new dataspace will immediately be changed to a new selection after copying the source dataspace. However, investigation shows that there are some places in the library's code where this may not always happen as intended.

For point selections, the information that is shared is the entire structure containing information about the selected points in the dataspace (see C.6). For hyperslab selections, the information that is shared is the span tree information for the hyperslab, which is used by the library when the selection in the hyperslab is irregular. However, the span tree information may also be used by the library in certain cases for regular hyperslabs. The structure for the span tree information also contains a reference count specifying how many shared references there are to the structure, which would need to be protected from concurrent modification if the span tree information is allowed to be shared among multiple threads. See C.5 for the structure that is shared for hyperslab selections.

While the effects of selection information being shared between multiple threads is not yet fully understood, attention is brought to this situation as it results in shared state between multiple threads. This could cause operations on distinct dataspace IDs in different threads to update both of the underlying dataspace objects and result in unintended behavior. From the perspective of the HDF5 public API, most of the methods for obtaining a new dataspace ID from copying an existing dataspace all use `H5S_copy()` while providing `false` for the parameter. However, there are a few exceptions to this case listed below among the functions which return an ID from a copied dataspace object.

- `H5Aget_space()`
- `H5Dget_space()`
- `H5Dget_space_async()`
- `H5Pget_virtual_vspace()`
- `H5Pget_virtual_srcspace()`
- `H5Scopy()`
- **`H5Scombine_select()`**
- **`H5Scombine_hyperslab()`**
- **`H5Ssel_iter_create()`**

For the `H5Scombine_select()` and `H5Scombine_hyperslab()` functions, there appear to be some cases where the new dataspace object returned could end up sharing hyperslab span lists with the original dataspace objects passed in as parameters. This generally appears to be the case when adding to an existing dataspace object using selection operations other than the `H5S_SELECT_SET` operation. However, the `H5Scombine_select()` and `H5Scombine_hyperslab()` functions are generally less used, as the same effect can usually be achieved with repeated calls to `H5Sselect_hyperslab()` to refine an existing hyperslab selection. Therefore, consideration should be given toward how much effort should go into addressing this issue. An easy solution may be to simply force selections in the dataspaces to not be shared by making an extra copy of the dataspace to be returned with `H5S_copy(..., false, ...)`.

`H5Ssel_iter_create()` is a more interesting case, as dataspace selection iterator objects are useful for VOL connectors to process dataspace selections during I/O. See section 3.3 for information on dataspace selection iterator objects.

From the perspective of internal HDF5 code, there are several other opportunities for selection information to be shared among dataspaces. This information is covered in the descriptions for internal functions in Appendix B.

3.3. Dataspace selection iterators

In order to get a list of the offset / length pairs that make up a selection within a dataspace object, an HDF5 application may first create a dataspace selection iterator object with `H5Ssel_iter_create()`. Then, the application can repeatedly call `H5Ssel_iter_get_seq_list()` on that iterator object to retrieve offset / length pairs until all pairs have been retrieved, at which point the application should release the iterator object with `H5Ssel_iter_close()`. This mechanism is often used by VOL connectors to process a dataspace selection during I/O. The `flags` parameter to this function can be specified as

H5S_SEL_ITER_SHARE_WITH_DATASPACE, in which case the selection information from the source dataspace is shared with the selection iterator object, as described in section 3.2. Sharing of this internal state could make concurrent use of iterator objects (even distinct iterator objects) by multiple threads problematic. While the solution to this could be to simply require that each thread create its own dataspace selection iterator while not specifying the H5S_SEL_ITER_SHARE_WITH_DATASPACE flag, this could be very expensive for point and hyperslab selections due to the information that has to be copied for each thread.

Refer to C.8 for more information on the internal state that is problematic for concurrent use of selection iterator objects.

3.4. Usage of free lists in the H5S interface

As the free list (H5FL) interface is not yet safe for multi-threaded environments, several memory management calls in the H5S interface will need to be converted to regular standard C memory management calls until the H5FL interface can be made thread-safe. At the time of writing, H5FL operations are used in the following files: **H5S.c**, **H5Shyper.c**, **H5Smpio.c**, **H5Spoint.c** and **H5Sselect.c**.

3.5. Concurrency issues specific to 'All' selections

No particular concurrency issues specific to 'All' selections were noted.

3.6. Concurrency issues specific to 'None' selections

No particular concurrency issues specific to 'None' selections were noted.

3.7. Concurrency issues specific to hyperslab selections

3.7.1. Rebuilds of hyperslab dimension information

In various places in the hyperslab code, the library may try and "rebuild" a hyperslab selection by converting a span tree representation into a regular hyperslab selection representation with the H5S__hyper_rebuild() function. If the library is able to convert the span tree representation into a regular selection, several fields in the hyperslab information structure will be modified. If the hyperslab selection is not a regular selection (i.e., it cannot be represented with a single hyperslab selection "set" call with H5S_SELECT_SET), the library will simply set a field in the hyperslab information structure to indicate this and move on. In either case, these rebuild operations can be triggered from operations that normally would appear to be "read-only", which poses an issue for concurrency.

3.7.2. Concurrent updates of hyperslab selection span 'op generations count'

The hyperslab selection implementation file, H5Shyper.c, contains a global variable H5S_hyper_op_gen_g which begins at 1 and is incremented whenever a hyperslab operation needs to retrieve an 'operation generation value'. While not documented, the apparent intent of this value is to serve as an optimization flag that allows code to skip repeated operations on a hyperslab selection's span tree. The relevant code generally does this by

performing an operation once, setting the 'operation generation value' in a field in the span tree's information, then comparing the span tree's 'operation generation value' against a value passed as a parameter in the next hyperslab operation that occurs. If the value passed in is the same value as was set in the field in the span tree's information, the code for the operation can assume that operation was already performed before and can skip some unnecessary work. This might involve reusing a cached value for the number of blocks calculated in a span tree, avoiding making an adjustment to the offset for a span tree when it was already done previously, etc.

Since multiple threads could be performing operations in H5Shyper.c code concurrently, this value must be protected either through locking or by making it atomic. However, if multiple threads **are** performing operations in H5Shyper.c code concurrently, there's still the possibility that concurrent updates to this value could eliminate any optimizations since the value may be changed by another thread during a thread's execution. Further investigation is needed, but this optimization appears to rely on serial execution of hyperslab operations.

3.8. Concurrency issues specific to point selections

The main data structure for point selections, `H5S_pnt_list_t` (C.6), contains two fields, `last_idx` (`hsize_t`) and `last_idx_pnt` (`struct H5S_pnt_node_t *`). These fields keep track of the index value of, as well as a pointer to the structure for, the next point in the point list that would have been visited if iteration over the list had continued. These fields are used for optimizing calls to the `H5Sget_select_elem_pointlist()` function so that if one were to, for example, retrieve subsets of the selected points in a dataspace in a loop, iteration over the list could continue at the saved location rather than having to iterate through list elements until arriving at the starting location each time. This optimization was presumably added due to the implementation of point selections currently using a linked list. If multiple threads were to call `H5Sget_select_elem_pointlist()` on the same dataspace object, or on dataspace objects that have shared selection information, these cached values would likely not be coherent.

4. Current state of concurrency in the H5S interface

The following sections detail the current state of support for concurrency within the H5S interface of HDF5 and discusses design decisions and/or proposals for resolving the concurrency issues encountered and listed in this document.

4.1. Current state

Somewhat exhaustive testing for the H5S interface was written first before beginning to adapt the interface for concurrency. This testing is in the test program `test/mt_h5s_test.c` (which may be moved under `test/threads` in the future). Currently, this test program has only a few sub-tests. Two sub-tests cover the use case of reading from a dataset using multiple threads, both with a dataspace shared among the threads and with separate, per-thread dataspaces. The remaining sub-tests essentially just perform several random H5S operations on dataspaces, both shared and per-thread, and check for expected error cases where concurrency can result in interleaving of H5S operations in ways which are invalid from HDF5's perspective, such as calling a point selection-specific function on a dataspace with a hyperslab selection. Two sub-tests cover

the 'read-only' case where threads only call functions from the first list below. Another four sub-tests cover the 'read-write' case where threads can call functions from either list below. For the 'read-write' tests where dataspaces are shared between threads, the test program tests both the case where only 1 thread is allowed to modify a dataspace concurrently with reader threads and the case where multiple threads are allowed to modify a dataspace concurrently. A sub-test still needs to be written for the 'write-write' case where threads call only functions from the second list below. This test will likely be very similar to the 'read-write' case with multiple writer threads, but may help to uncover deeper concurrency issues by allowing for more concurrent modifications to be made to a shared dataspace. The idea in testing H5S like this is simply to test as exhaustively as possible while trying to bring up concurrency issues in H5S. This interface is well-tested for certain functions, such as `H5Sselect_hyperslab()`, but is very poorly tested for many other H5S functions. As concurrency within H5S stabilizes, more tests that are catered toward specific use cases will be added.

Here, 'read-only' testing means only calling API functions which either retrieve information from a dataspace or which return a new dataspace without modifying one that is passed in to the function. This list of functions is as follows:

- `H5Screate()`
- `H5Screate_simple()`
- `H5Scombine_hyperslab()`
- `H5Scombine_select()`
- `H5Sdecode()`
- `H5Sencode2()`
- `H5Sextent_equal()`
- `H5Sselect_intersect_block()`
- `H5Sselect_project_intersection()`
- `H5Scopy()`
- `H5Sget_regular_hyperslab()`
- `H5Sget_select_bounds()`
- `H5Sget_select_elem_npoints()`
- `H5Sget_select_elem_pointlist()`
- `H5Sget_select_hyper_nblocks()`
- `H5Sget_select_hyper_blocklist()`
- `H5Sget_select_npoints()`
- `H5Sget_select_type()`
- `H5Sget_simple_extent_ndims()`
- `H5Sget_simple_extent_dims()`
- `H5Sget_simple_extent_npoints()`
- `H5Sget_simple_extent_type()`

- H5Sis_regular_hyperslab()
- H5Sis_simple()
- H5Sselect_shape_same()
- H5Sselect_valid()

Testing in a 'read-write' capacity means concurrent, mixed usage of the functions mentioned above with functions that modify some portion, the extent or selection or both, of a dataspace. This list of 'write' functions is as follows:

- H5Sclose()
- H5Sextent_copy()
- H5Smodify_select()
- H5Soffset_simple()
- H5Sselect_adjust()
- H5Sselect_all()
- H5Sselect_copy()
- H5Sselect_elements()
- H5Sselect_hyperslab()
- H5Sselect_none()
- H5Sset_extent_none()
- H5Sset_extent_simple()

Dataspace selection iterator objects have their own concurrency issues which are separate from the H5S interface, so they are considered in a separate group here. For reference, the dataspace selection iterator functions are:

- H5Ssel_iter_create()
- H5Ssel_iter_get_seq_list()
- H5Ssel_iter_reset()
- H5Ssel_iter_close()

Initial testing appears to show that the acquisition of the library's global mutex can be dropped for nearly all H5S functions when testing on per-thread dataspaces without further modifications being made. The only portion of the H5S interface that hasn't yet been tested for concurrency issues in this case is the dataspace selection iterator portion. In the per-thread dataspaces case, there appear to be only three relevant issues with shared state, which are listed in sections 3.4, 3.7.2 and 3.8. For the first, multi-thread builds of the library currently disable free lists entirely. While this isn't an ideal solution, note that **H5S concurrency is currently relying on this fact**. For simplicity, the H5FL-related code in H5S files has not been modified. The second issue was resolved by simply making the counter value an atomic variable for now, though this effectively eliminates the optimizations that the counter was enabling, so a different solution (such as moving the value to be thread-local) should be considered for the future. The main initial goal was just to prevent different threads from ever seeing the same value for this variable, as it could result in threads returning stale cached data. The

third issue remains, but isn't dangerous, just problematic from a correctness perspective, and will be resolved as the design for H5S concurrency evolves.

Support for H5S concurrency is still being designed for the shared dataspace case, with notes in the following sections. Acquisition of the library's global mutex has been pushed below the initial `FUNC_ENTER` macro for all H5S functions in favor of acquiring it with `H5_API_LOCK` / `H5_API_UNLOCK` where currently necessary. This simply allows slightly more concurrency inside the H5S functions (which also implicitly involves testing concurrent H5I operations) by reducing the duration in which the lock is held and also makes it easier to spot which parts of the H5S code need to be focused on. In order to drop acquisition of the lock everywhere, either the library's `H5S_t` structure needs to be made atomic, or its two fields, `H5S_extent_t` and `H5S_select_t`, need to be made atomic. Without this (and without locking), a dataspace can be left in a bad state when a thread changes either the extent of a dataspace or the selection within it. Currently, it is being investigated whether it is less complex to mark the entire `H5S_t` structure / its fields as Atomic (in which case operations on the fields will likely be implicitly locked around due to the size of the structures) or whether the fields of the sub-structures can be grouped into smaller atomic kernel structures. If the structures can be split into smaller kernels, operations are more likely to be lock-free and have a smaller impact on performance for the per-thread dataspaces case, where there are effectively no concurrency issues that have to be dealt with.

Despite this, note that testing in a 'read-only' capacity works well in the shared dataspace case with the global mutex dropped for all 'read-only' functions. Ideally, a solution can be found for cases that involve modifications to the dataspaces as well, as copying of dataspaces has historically proven to be a very significant source of application overhead, meaning that only supporting per-thread dataspaces could be problematic.

4.2. Concurrent closes on shared dataspace IDs

Directly associated with a dataspace object ID is a pointer to the underlying `H5S_t` structure. With the global mutex pushed down past H5I operations inside H5S functions, concurrently closing a dataspace ID could result in segfaults due to a thread freeing the dataspace object pointer out from under other threads while they may be trying to access the dataspace object[1] While this issue is currently being ignored due to this essentially being an error from the perspective of the application, it would still seem important to find a solution to protect against this case. A naive first solution might be to take an internal reference on the ID before retrieving the underlying pointer and then decrementing the reference count on the way out of the H5S function. However, this has a few issues:

- The constant incrementing and decrementing of reference counts may bring significant overhead.
- By holding internal references on the dataspace ID, the dataspace object remains around even after the application reference count initially hits 0. At that point, multiple threads are able to call the function `H5I_dec_app_ref()` (in `H5Sclose()`) on the ID, decrementing the application reference count for the ID far more than intended. This has the effect of causing the application reference count to wrap around to `UINT_MAX` after reaching 0 and triggers the assertion failure:

```
H5Iint.c:5318: H5I__dec_ref: Assertion `mod_info_k.count >=
                             mod_info_k.app_count' failed.
```

[1] Note that this will essentially be the case for all concurrent close operations on HDF5 object IDs.

While this is easily fixed by checking the application reference count before decrementing it, it is not clear that this is a desired solution. Though, it does seem reasonable to protect against this case regardless.

- Decrementing of the reference count on the dataspace ID still needs to be coordinated among threads in `H5Sclose()`. Otherwise, multiple threads can still end up decrementing the regular reference count on the ID to the point that the pointer is freed out from under threads in other H5S functions. Note that locking around decrementing the reference count alone will not suffice here. In between the time that a thread in an H5S function acquires an internal reference to an ID and then uses the underlying dataspace pointer, multiple threads in `H5Sclose()` could acquire the lock, decrement the reference count (eventually down to 0) and drop the lock, resulting in the pointer still being freed out from under other threads. Something such as a thread barrier preceding the locking and decrementing calls would be needed, but this presents a challenge when determining the number of threads that will wait on the barrier.

For now, this issue has been somewhat hacked around by assuming that an ID has already been closed externally if the application reference count is 0. Therefore, the code in `H5Sclose()` acquires the global mutex, checks the application reference count on the ID (which could potentially be done outside the lock, but seemed reasonable to do under the lock for now) and then skips the decrement operation if the count is 0. Note that **this assumes that once the application reference count hits 0 it will stay at 0**. In theory, this should not be an issue, at least for dataspace IDs, but this is not an ideal or general solution. It is being investigated whether the approach in section 4.3 may be useful here.

4.3. Concurrent modification of selection or extent in shared dataspace

One of the first concurrency issues encountered during testing was the case where one thread changes the type of selection or extent within a dataspace while another thread is operating on it. For example, consider a dataspace with a hyperslab selection in it that is shared among threads. One thread calls the function `H5Sget_regular_hyperslab()` on the dataspace, while another thread calls `H5Sselect_all()` on the dataspace. This presents issues in both serial and multi-threaded builds of HDF5. In the serial case (or serialized multi-thread case), depending on the order the two calls happen in, this can succeed or fail if the selection type is changed to 'all' before `H5Sget_regular_hyperslab()` is called. In that case, the call to `H5Sget_regular_hyperslab()` will fail because it will determine that the selection within the dataspace object is not a hyperslab selection and will immediately return a failure. Note that there are several other combinations of mixing 'get' operations with operations that change the selection type or extent within a dataspace which can result in errors.

In the multi-threaded case, this often results in segfaults when acquisition of the library's global mutex is removed from H5S functions to open them up for concurrency. This is because operations which modify the selection or extent of a dataspace invalidate specific pointer fields within the dataspace object `H5S_t` structure. While locking can prevent this, the locking would have to be performed around nearly the entirety of every H5S function (at least right before the dataspace pointer is retrieved and around all following logic that makes use of the pointer), which generally defeats the purpose of initially pushing down acquisition of the global mutex by H5S functions. Another solution might be to simply detect concurrent access to shared dataspaces and return an error. But one goal of the investigation into H5S concurrency is to determine if a reasonable solution for allowing concurrent access to shared dataspaces can be implemented. To that end, it is proposed that the aforementioned RCU approach should be considered. This specific approach is recommended because

making updates to `H5S_t` structures atomically is only one part of the equation. An update to one of these structures could occur atomically at any point after a thread has atomically loaded the part of the structure they're interested in, still resulting in the same issue. Therefore, a way to signify that a specific data structure should not be de-allocated while there are readers is needed. Properly implemented, this should at least allow the H5S code to return either new pointers after modifications to a dataspace or the old, not yet de-allocated pointers. This would prevent threads from segfaulting due to trying to access already freed or otherwise invalid memory.

Note, however, that this still means that stale data could be returned. Depending on exactly what the application is attempting to do, it is still likely that the application will encounter the same error situations as in the serial case. For the time being, the H5S testing code has chosen to ignore errors like this, with the thought being that this should be considered user error. This situation could, in theory, be resolved with a versioning approach similar to what was implemented for the H5P interface. However, this would involve keeping around copies of old selections within the dataspace, which could be problematic for larger dataspace selections. This use case also seems to be less reasonable than the use case that resulted in the solution for the H5P interface. It is proposed that this limitation should be documented as a programming model issue. The application would be responsible for certain aspects of correct concurrent usage of read-write operations on dataspace objects.

4.4. Miscellaneous bugs in HDF5

During testing of the H5S interface, multiple bugs were found in HDF5, including:

- Several bugs in the H5S code that would result in assertion failures and were converted into normal error checks.
- A bug in the internal function for retrieving the list of blocks in a hyperslab selection which would result in a segfault or bus error due to over-reading from an array variable on the stack.
- A bug where `H5Sselect_copy()` or the selection copying portion of `H5Scopy()` would release the selection in the destination dataspace, even when the function fails. This would result in segfaults later on when the application attempts to close the dataspace. This was fixed by using a temporary dataspace object and only releasing the selection in the destination dataspace once everything had succeeded.

Many of these bugs were found due to unexpected circumstances in HDF5 after calling the function `H5Sset_extent_none()` on a dataspace. While the issues were generally fixed, it is recommended that multi-threaded code completely avoid calling this function as it can result in situations that application code may not have predicted, leading to issues such as 0-sized allocations and buffer overflows. Due to the issues found, recommendation sections were added to the descriptions of some H5S functions in Appendix A.

4.5. RCU-like approach proposal

To solve the issues in section 4.3, it's been proposed that the multi-thread library should adopt an approach similar to RCU for keeping coherent, concurrent access to certain data structures. The following sections describe the general idea and how it can be applied in the context of HDF5.

4.5.1. Overview

RCU (read-copy-update) is a technique that allows threads to modify a data structure without causing active readers of that data structure to encounter segfaults or other issues due to parts of the data structure being invalidated/de-allocated. At a basic level, this is accomplished by having writers make a copy of the data structure, modify the copy and then update the canonical pointer to the data structure with the modified copy, while retaining the old data structure until all active readers no longer need the old copy of the data structure. At that point, the old data structure can safely be de-allocated. In this manner, threads always have some valid data structure to refer to, whether it's the old or new version. Note that this means readers of the data structure may end up reading data that is stale or no longer valid, but, in the context of HDF5, this is fundamentally a matter of managing concurrency correctly at the application level.

4.5.2. Implementation

For a first cut, this implementation will focus more on ease of implementation than performance. RCU approaches often focus on making readers of data structures fast at the expense of writers of those data structures by avoiding the need for writes to any shared variables from the reader side. However, this implementation will use reference counting with atomics to keep track of how many readers of a data structure exist in order to determine when those structures can be de-allocated. While no official investigation into the performance of atomics has been made, a cursory search appears to show that this could be problematic from the perspective of reader performance due to the semantics of atomic fetch and add/subtract instructions. There are several ways this can be optimized later as needed.

To begin with, a way of denoting a critical section for reader threads is needed. This critical section will mark (and unmark) a data structure as protected in some way, signifying to the library that the data structure can't be de-allocated as long as there are active reader threads for the data structure. Following with traditional naming, the following new functions are initially proposed, pending further design investigation:

```
/*
 * Enter a reader side critical section. Finds an object pointer
 * for the specified ID, marks the object as protected for RCU
 * purposes, verifies that it's in a particular type and returns
 * the object pointer.
 */
void *
H5I_object_rcu_lock_and_verify(hid_t id, H5I_type_t type);

/*
 * Exits a reader side critical section. Marks the object pointed
 * to by the ID as no longer needed by the calling thread. Once
 * all threads that had RCU protected the object pointed to by the
 * ID have exited their critical sections by calling this function,
 * the object will be marked as available for reclamation.
 */
herr_t
H5I_object_rcu_unlock(hid_t id);
```

Note that the function `H5I_object_rcu_lock_and_verify()` is a combination function to perform both the RCU object protection and ID lookup similar to `H5I_object_verify()`. This is for two reasons:

- Since the RCU approach mandates that no references to an RCU-protected data structure can be held outside of a reader critical section, this enforces the ordering by combining the two operations.
- Due to the way HDF5 hides objects behind IDs, the library will need to look up the underlying ID info structure in order to RCU protect a data structure in the first place. By returning the object pointer from this function, this effectively prevents a caller from having the library look up an ID twice, once to RCU protect a structure and a second time to get the pointer to the structure.

While subject to change, these two functions represent the basic functionality needed by readers of a data structure: the ability to mark an object so that it can't be de-allocated, the ability to get a valid pointer to that object and the ability to release the hold on the object. When a reader thread wishes to access a data structure behind an H5I ID which could be modified by a writer thread, it will:

- Enter a reader critical section on the ID with `H5I_object_rcu_lock_and_verify()`
- Perform any read operations on the data structure pointed to **only through the pointer returned by `H5I_object_rcu_lock_and_verify()`**
- Exit the reader critical section on the ID with `H5I_object_rcu_unlock()`

In this case, entering a reader critical section will simply involve incrementing a count of the number of readers for the data structure, which will inform subsequent code that there are readers of the structure and it can't be de-allocated. Thus, exiting a critical section would involve decrementing that count and the trigger for a data structure becoming available for de-allocation would be that count reaching 0. The internal details of this count mechanism still largely need to be worked out.

The writer side doesn't necessarily need new functions for implementing the semantics of RCU, but will likely need some additional internal pointer maintenance logic. The main mechanism needed by writers is the ability to atomically swap a pointer to a new data structure in place of the old one, while retaining the old pointer. This can already be accomplished with the version of `H5I_subst()` that has been re-written for multi-threaded HDF5. Though, an additional `H5I_try_subst()` function might be needed to cover the issue discussed in section 4.5.3. When a writer thread wishes to modify a data structure behind an H5I ID, it will:

- Increment the internal reference count on the ID
- Get the pointer to the object underlying the ID
- Make a copy of the pointed to structure
- Decrement the internal reference count on the ID
- Modify its copy of the structure
- Atomically swap its copy of the structure in place of the old structure with `H5I_subst()`, removing the ability for new readers to get a reference to the old structure while still retaining its own reference to the structure
- Set up for later garbage collection of the old structure when there are no more readers of that structure

The last step in this process is vague, as additional details will need to be worked out regarding the best approach for dealing with the eventual reclamation of the memory for the old structure. While a typical

RCU approach would be to have the writer side block until all readers are finished, at which point it could de-allocate the memory itself, this poses some problems:

- Blocking the writer side is not ideal, especially if there are many concurrent, mixed reads and writes. While there are solutions for performing RCU without the writer side needing to block, the next point could be problematic even in that case.
- Depending on how tracking of the number of readers for a data structure is implemented, there could still exist the possibility of a writer thread de-allocating a structure before a reader has a chance to increment the readers count. This would look something like the following: a reader thread enters `H5I_object_rcu_lock_and_verify()` and gets to the point that the ID type info kernel (`H5I_mt_id_info_kernel_t`) structure is retrieved, at which point the thread is preempted **before incrementing the readers count**. A writer thread swaps in a new structure that is now pointed to by the ID, sees that there are **currently** no writers, de-allocates the structure and moves on. The reader thread resumes execution and is returned the now invalid pointer, resulting in badness.

The initial thoughts around this design are to have the reader/reference count be a part of the pointed to data structure, presenting the possibility for the thread racing issue in the second point above. A first solution discussed for this problem would be to have writer threads always place old references to a structure on a free list and use a separate mechanism for garbage collecting the old references once their reader count falls to 0. The internal details of this mechanism still largely need to be worked out.

Note that writer threads will make a copy of the entire `H5S_t` structure during this process, even though H5S operations generally only modify either the extent or the selection of a dataspace, but not both. This is due to the `extent` and `select` fields in `H5S_t` being structs, rather than pointers-to-structs, inside the overall dataspace structure. To keep the old values consistent for a reader thread, writer threads will swap in an entirely new structure. While expensive, this is a straightforward way of achieving the RCU goal and can be optimized later if need be.

4.5.3. Note on RCU approach with multiple writers to shared structure

When multiple writer threads are trying to update a shared `H5S_t` structure concurrently, the goal is to make these updates happen in some order, though it may be an unpredictable order, without any of the updates being dropped. To handle this, it is proposed that a version number should be added to the `H5S_t` structure to keep track of the current version pointed to by a dataspace ID. When a writer attempts to update the structure, it will increment the version number in its local copy of the structure and use an atomic compare and swap to try to update the structure with its modified copy **as long as the structure matches what is expected, e.g. the fields match the old structure and the current version is one less than the version in the structure replacing the current one**. If the compare and swap fails, the writer will discard its copy of the structure, make another copy, make any modifications it needs, including incrementing the version count from what the most recent value was and then try to replace the structure with its copy again. In this manner, the concurrent writes will be serialized in some fashion as the atomic compare and swaps from various threads succeed in some order and keep incrementing the version number. This will likely be very expensive if there are multiple writers to a shared dataspace, but this is again something that can be optimized later once a better view of use cases is developed.

Revision History

Version Number	Date	Comments
v1	Jan. 24, 2025	Version 1 drafted.
v2	Mar. 14, 2025	Version 2 drafted.

A. Appendix: H5S interface public API functions

The following sections give an overview of the public API functions in the H5S interface and outline any concurrency issues that will have to be dealt with for each function. Note that the following three general assumptions are made before getting into concurrency issues specific to individual functions:

- The assumption is made that the H5I interface in HDF5 is thread-safe. While this is, for the most part, currently the case in Lifeboat's HDF5 fork, there may be some additional concurrency bugs encountered when adapting the H5S interface for multi-thread environments. Most or all of these functions use the H5I interface in some form or another, so concurrent use of H5S relies on an H5I interface capable of concurrency.
- The assumption is made that all of the usages of HDF5's H5FL free list interface will have been converted to thread-safe equivalents before beginning the work of adapting the H5S interface for multi-thread environments. The H5FL interface is not currently thread-safe and would pose several concurrency issues related to memory management of objects.
- Unless specifically discussed, the assumption is made that parameters passed into these functions are either thread-local or are being concurrently accessed in a read-only capacity.

Some of the functions in the following sections discuss issues when a dataspace object is concurrently modified while being used inside the function. For these discussions, it should be kept in mind that concurrent modification to a dataspace object can primarily happen in two ways. The first way is via direct manipulation, e.g. multiple threads pass the same dataspace ID as a parameter to functions where one or more of the function calls will update the dataspace object in some manner. If the chosen approach to H5S concurrency does not allow for the same dataspace object to be shared by multiple threads, this issue is not relevant. Otherwise, a system will need to be devised for ensuring that the dataspace remains coherent during operations.

The second way is via indirect manipulation, e.g. multiple threads pass distinct dataspace IDs to functions where one or more of the function calls will update the dataspace objects in some manner and the underlying dataspace objects have shared selection information between them. Currently, this is a concern for dataspaces that have either a hyperslab or point selection set on them. In certain edge cases, these dataspace objects could have shared selection information between them and updating one dataspace object may update another. After determining the full impact of this state sharing, a system may need to be devised for ensuring that the dataspace remains coherent during operations. Refer to 3.2 for more information on this case.

A.1. `H5Sclose()`

```
herr_t
H5Sclose(hid_t space_id);
```

Releases and terminates access to a dataspace.

Concurrency notes: With the general assumptions made at the beginning of this appendix, this function should be thread-safe. However, note that locking may need to temporarily be performed around the call to `H5I_dec_app_ref()` to deal with issues with an ID being closed by one thread while already having been removed by another thread. Further investigation is needed to determine if this will actually be an issue.

A.2. `H5Scombine_hyperslab()`

```
hid_t
H5Scombine_hyperslab(hid_t space_id, H5S_seloper_t op,
                     const hsize_t start[],
                     const hsize_t stride[],
                     const hsize_t count[],
                     const hsize_t block[]);
```

Performs an operation on a hyperslab with an existing selection and returns the ID of a new dataspace object with the resulting selection.

Concurrency notes: This function appears to have several issues with concurrency.

First, unexpected results could occur if the dataspace is concurrently modified while this function is executing.

Next, when multiple threads call `H5Scombine_hyperslab()` on the same dataspace ID, then, depending on the selection operator used, the span tree information could be shared among all the dataspaces that are newly created for each thread. If the chosen approach to H5S concurrency does not allow for the same dataspace object to be shared by multiple threads, this issue is not relevant. A similar situation exists when multiple threads call `H5Scombine_hyperslab()` on distinct dataspace IDs where the span tree information is shared between the dataspaces. The potential danger for both of these situations is described more in section 3.2.

Finally, this function can encounter the issue described in section 3.7.2.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.3. `H5Scombine_select()`

```
hid_t
H5Scombine_select(hid_t space1_id, H5S_seloper_t op, hid_t space2_id);
```

Combines two hyperslab selections according to the given selection operation and returns the ID of a new dataspace object with the resulting selection. This function is very similar to `H5Scombine_hyperslab()`.

Concurrency notes: This function appears to have several issues with concurrency.

First, unexpected results could occur if either of the dataspaces are concurrently modified while this function is executing.

Next, when multiple threads call `H5Scombine_select()` on the same dataspace ID (`space1_id`), then, depending on the selection operator used and the selection in `space2_id`, the span tree information could be shared among all the dataspaces that are newly created for each thread. If the chosen approach to H5S concurrency does not allow for the same dataspace object to be shared by multiple threads, this issue is not relevant. A similar situation exists when multiple threads call `H5Scombine_select()` on distinct

dataspace IDs where the span tree information is shared between the dataspaces. The potential danger for both of these situations is described more in section 3.2.

Finally, this function can encounter the issue described in section 3.7.2.

A.4. `H5Scopy()`

```
hid_t
H5Scopy(hid_t space_id);
```

Copies the extent and selection for a dataspace object and returns the ID of a new dataspace object with the copied extent and selection.

Concurrency notes: If multiple threads call `H5Scopy()` on the same dataspace ID (presumably to obtain their own copy of a shared dataspace to work with) and that dataspace object is modified concurrently, the extent and/or selection of the resulting dataspace copy could be indeterminate. If the chosen approach to H5S concurrency does not allow for the same dataspace object to be shared by multiple threads, this issue is not relevant. Otherwise, a system will need to be devised for ensuring that the dataspace being copied remains coherent during the copy operation.

The same situation could occur for the selection in the resulting dataspace if multiple threads call `H5Scopy()` on distinct dataspace IDs where the underlying dataspace objects have shared selection information between them. Note that `H5Scopy()` never shares the selection information between the source and resulting dataspaces, but if the dataspaces are modified during the copy operation, the results could be indeterminate. Refer to section 3.2 for information on shared selection information.

A.5. `H5Screate()`

```
hid_t
H5Screate(H5S_class_t type);
```

Creates a new dataspace object of the specified type and returns the ID of the newly-created dataspace object.

Concurrency notes: With the general assumptions made at the beginning of this appendix, this function should be thread-safe. It simply allocates a new `H5S_t` structure, performs some setup operations on the allocated structure and then calls `H5I_register()` to get an ID for the dataspace object that is then returned.

A.6. `H5Screate_simple()`

```
hid_t
H5Screate_simple(int rank, const hsize_t dims[],
                 const hsize_t maxdims[]);
```

Creates a new simple dataspace object and returns the ID of the newly-created dataspace object.

Concurrency notes: With the general assumptions made at the beginning of this appendix, this function should be thread-safe. It simply allocates a new `H5S_t` structure, performs some setup operations on the

allocated structure and then calls `H5I_register()` to get an ID for the dataspace object that is then returned.

A.7. `H5Sdecode()`

```
hid_t
H5Sdecode(const void *buf);
```

Decodes a binary representation of a dataspace object and then returns the ID of a dataspace object created from the decoded data.

Concurrency notes: With the general assumptions made at the beginning of this appendix, the remaining concurrency issues with `H5Sdecode()` lie in its helper function `H5S_decode()`. To correctly decode a binary representation for a dataspace object, `H5S_decode()` calls `H5F_fake_alloc()` to allocate a fake `H5F_t` file structure object which is primarily used for determining the 'size of sizes' in a file. This function uses free list memory management operations and must be converted to thread-safe equivalents. `H5S_decode()` also calls the `H5O_msg_decode()` and `H5O_msg_copy()` functions to decode the dataspace extent part of the binary blob and copy it into the newly-created dataspace object. Both of these functions use free list memory management operations as well.

Finally, `H5S_decode()` calls `H5S_select_deserialize()` to decode the dataspace selection part of the binary blob and set that selection in the newly-created dataspace object. Depending on the selection type, there are further concurrency issues to consider.

- For 'all' and 'none' selections, no operations performed by their 'deserialize' callbacks appear to be problematic for concurrency.
- For point selections, the callback function `H5S__point_deserialize()` allocates space for the point coordinates array using `H5MM_malloc()` which should currently be thread-safe as it's just a wrapper around `malloc()`, but it may be worth converting to an explicit call to `malloc()` for future-proofing reasons. `H5S__point_deserialize()` also calls the `H5S_select_elements()` function to select elements in the newly-created dataspace object which were selected in the binary representation of the dataspace object. The main concurrency issue in this function is the various calls to free list memory management operations, which we'll assume will have been updated to thread-safe equivalents.
- For hyperslab selections, the callback function `H5S__hyper_deserialize()` eventually calls `H5S_select_hyperslab()` to setup the selection in the newly-created dataspace object. This function will need to generate the span information for the new dataspace object, which is very likely to encounter the issue with hyperslab operation generation values described in section 3.7.2.

A.8. `H5Sencode2()`

```
herr_t
H5Sencode2(hid_t obj_id, void *buf, size_t *nalloc, hid_t fapl);
```

Encodes a dataspace object into a binary buffer.

Concurrency notes: With the general assumptions made at the beginning of this appendix, `H5Sencode2()` has a few remaining concurrency issues. This function calls `H5CX_set_apl()`, which will need to be dealt with until the H5CX interface can be made thread-safe. Other concurrency issues with `H5Sencode2()` lie in its helper function `H5S_encode()`. To correctly encode a binary representation for a dataspace object, `H5S_encode()` calls `H5F_fake_alloc()` to allocate a fake `H5F_t` file structure object which is primarily used for determining the 'size of sizes' in a file. This function uses free list memory management operations and must be converted to thread-safe equivalents.

Then, `H5S_encode()` calls `H5S_select_serial_size()` to determine the size of the buffer needed for encoding the selection part of the dataspace object. Depending on the selection type, there are further concurrency issues to consider.

- For 'all' and 'none' selections, no operations performed by their 'serial_size' callbacks appear to be problematic for concurrency.
- For point selections, the callback function `H5S__point_serial_size()` eventually calls the function `H5CX_get_libver_bounds()`, which will need to be dealt with until the H5CX interface can be made thread-safe.
- For hyperslab selections, the callback function `H5S__hyper_serial_size()` can encounter the issue described in section 3.7.2. The function also eventually calls `H5CX_get_libver_bounds()`, which will need to be dealt with until the H5CX interface can be made thread-safe.

`H5S_encode()` then calls `H5S_select_serialize()` to perform the actual encoding operation. Each selection type's 'serialize' callback has essentially the same concurrency issues as their 'serial_size' callbacks above, so fixing the issues will address both cases.

Finally, unexpected results could occur if the dataspace is concurrently modified while it is being encoded into the buffer.

A.9. `H5Sextent_copy()`

```
herr_t
H5Sextent_copy(hid_t dst_id, hid_t src_id);
```

Copies the extent of a dataspace object to another dataspace object.

Concurrency notes: Unexpected results could occur if the source dataspace's extent is concurrently modified while copying of its extent is taking place.

Unexpected results could also occur if multiple threads attempt to copy an extent into the same destination dataspace object. This case seems unlikely to be of much use or interest and should likely be documented as an invalid operation.

A.10. `H5Sextent_equal()`

```
htri_t
H5Sextent_equal(hid_t space1_id, hid_t space2_id);
```

Compares the extents of two dataspace objects and determines if they are equal.

Concurrency notes: Unexpected results could occur if the extent of either dataspace object is concurrently modified during the compare operation.

A.11. `H5Sget_regular_hyperslab()`

```
htri_t
H5Sget_regular_hyperslab(hid_t spaceid, hsize_t start[],
                         hsize_t stride[], hsize_t count[],
                         hsize_t block[]);
```

Retrieves the parameters for a regular hyperslab selection set on a dataspace object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the selection within it is taking place.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.12. `H5Sget_select_bounds()`

```
herr_t
H5Sget_select_bounds(hid_t spaceid, hsize_t start[], hsize_t end[]);
```

Retrieves the parameters of a bounding box containing the current selection within a dataspace object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the bounds of the selection within it is taking place.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.13. `H5Sget_select_elem_npoints()`

```
hssize_t
H5Sget_select_elem_npoints(hid_t spaceid);
```

Retrieves the number of element points selected in the dataspace object. The dataspace object must have a point selection within it.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the number of selected points is taking place.

A.14. H5Sget_select_elem_pointlist()

```
herr_t
H5Sget_select_elem_pointlist(hid_t spaceid, hsize_t startpoint,
                             hsize_t numpoints, hsize_t buf[]);
```

Retrieves a list of the element points selected in the dataspace object. The dataspace object must have a point selection within it.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the list of selected points is taking place.

This function has additional concurrency issues described in 3.8.

Miscellaneous recommendations: It is recommended that a `buf_size` parameter be added to this function to specify the number of `hsize_t` elements that the buffer has space for. Without this, the library can easily end up overflowing the `buf` array, especially when the buffer is accidentally allocated with a size of 0 bytes (which can happen when a previous call to `H5Sget_select_elem_npoints()` returns 0. This is also an easy mistake to make when concurrency is involved with H5S functions.

A.15. H5Sget_select_hyper_blocklist()

```
herr_t
H5Sget_select_hyper_blocklist(hid_t spaceid, hsize_t startblock,
                              hsize_t numblocks, hsize_t buf[]);
```

Retrieves a list of hyperslab blocks selected in the dataspace object. The dataspace object must have a hyperslab selection within it.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the list of hyperslab blocks is taking place.

Miscellaneous recommendations: It is recommended that a `buf_size` parameter be added to this function to specify the number of `hsize_t` elements that the buffer has space for. Without this, the library can easily end up overflowing the `buf` array, especially when the buffer is accidentally allocated with a size of 0 bytes (which can happen when a previous call to `H5Sget_select_hyper_nblocks()` returns 0. This is also an easy mistake to make when concurrency is involved with H5S functions.

A.16. H5Sget_select_hyper_nblocks()

```
hssize_t
H5Sget_select_hyper_nblocks(hid_t spaceid);
```

Retrieves the number of hyperslab blocks selected in the dataspace object. The dataspace object must have a hyperslab selection within it.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the number of selected hyperslab blocks is taking place. This function can also encounter the issue described in section 3.7.2.

A.17. H5Sget_select_npoints()

```
hssize_t
H5Sget_select_npoints(hid_t spaceid);
```

Retrieves the number of elements selected in a dataspace object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the number of selected elements is taking place.

A.18. H5Sget_select_type()

```
H5S_sel_type
H5Sget_select_type(hid_t spaceid);
```

Returns the type of selection within a dataspace object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the type of selection is taking place.

A.19. H5Sget_simple_extent_dims()

```
int
H5Sget_simple_extent_dims(hid_t space_id, hsize_t dims[],
                          hsize_t maxdims[]);
```

Retrieves the current and maximum sizes of each dimension in a dataspace object's extent and also returns the number of dimensions in the dataspace object's extent. An initial call with both `dims` and `maxdims` specified as `NULL` can be made to determine how large each array must be before making a second call to populate those arrays.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dimensionality information is taking place.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.20. H5Sget_simple_extent_ndims()

```
int
H5Sget_simple_extent_ndims(hid_t space_id);
```

Retrieves the number of dimensions in a dataspace object's extent.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dimensionality information is taking place.

A.21. H5Sget_simple_extent_npoints()

```
hssize_t
H5Sget_simple_extent_npoints(hid_t space_id);
```

Retrieves the number of elements in a dataspace object's extent.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dataspace information is taking place.

A.22. H5Sget_simple_extent_type()

```
H5S_class_t
H5Sget_simple_extent_type(hid_t space_id);
```

Retrieves the type of extent within a dataspace object (currently, scalar, simple, null).

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dataspace information is taking place.

A.23. H5Sis_regular_hyperslab()

```
htri_t
H5Sis_regular_hyperslab(hid_t spaceid);
```

Returns whether or not the selection in the specified dataspace object represents a regular hyperslab selection. The dataspace object must have a hyperslab selection within it.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dataspace information is taking place.

A.24. H5Sis_simple()

```
htri_t
H5Sis_simple(hid_t space_id);
```

Returns whether or not the extent of the specified dataspace object is "simple".

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of the dataspace information is taking place.

A.25. H5Smodify_select()

```
herr_t
H5Smodify_select(hid_t space1_id, H5S_seloper_t op, hid_t space2_id);
```

Refines an existing hyperslab selection in the dataspace object `space1_id` by operating on it with the hyperslab selection in the dataspace object `space2_id` according to the selection operation specified by `op`.

Concurrency notes: Unexpected results could occur if either dataspace is concurrently modified while the selection modification operation is taking place. Further, this function is affected by the issues described in section 3.1 if multiple threads call `H5Smodify_select()` on the same dataspace `space1_id`. Specifically, if a dataspace object is shared among multiple threads then concurrently updating the selection in the dataspace object will be problematic. Regardless of the chosen selection operator, the resulting selection could be indeterminate as each thread attempts to replace some or all of the existing selection with its own.

A.26. `H5Soffset_simple()`

```
herr_t
H5Soffset_simple(hid_t space_id, const hssize_t *offset);
```

Changes the offset which the current selection in a dataspace object starts at. This function essentially allows moving around of a selection in a dataspace object without having to redefine the selection.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection modification operation is taking place. Further, concurrent use of this function can be affected by both of the issues described in section 3.1.

Miscellaneous recommendations: This function assumes that the passed in array is at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added to help prevent this issue.

A.27. `H5Ssel_iter_close()`

```
herr_t
H5Ssel_iter_close(hid_t sel_iter_id);
```

Closes the ID for and terminates access to a dataspace selection iterator object.

Concurrency notes: With the general assumptions made at the beginning of this appendix, this function should be thread-safe. However, note that locking may need to temporarily be performed around the call to `H5I_dec_app_ref()` to deal with issues with an ID being closed by one thread while already having been removed by another thread. Further investigation is needed to determine if this will actually be an issue.

A.28. `H5Ssel_iter_create()`

```
hid_t
H5Ssel_iter_create(hid_t spaceid, size_t elmt_size, unsigned flags);
```

Creates a dataspace selection iterator object and returns an ID for it.

Concurrency notes: With the general assumptions made at the beginning of this appendix, this function should be thread-safe as long as the dataspace is not concurrently modified while the selection iterator is being created. However, if the value `H5S_SEL_ITER_SHARE_WITH_DATASPACE` is passed in for the `flags` parameter, the selection iterator object will have shared state with the source dataspace object, which could be

problematic for concurrent execution, as described in section 3.2. For point selections, the list of points will be uncopied and directly used. For hyperslab selections, the span information will be uncopied and directly used.

Dataspace selection iterator objects have additional concurrency issues noted in section 3.3.

A.29. `H5Ssel_iter_get_seq_list()`

```
herr_t
H5Ssel_iter_get_seq_list(hid_t sel_iter_id, size_t maxseq,
                         size_t maxelmts, size_t *nseq,
                         size_t *nelmts, hsize_t *off,
                         size_t *len);
```

Retrieves the next set of offset / length sequences for the elements in a dataspace selection iterator object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while retrieval of a sequence list operation is taking place, or even in between calls to this function. This function would also be affected by the concurrency issues noted in 3.3.

A.30. `H5Ssel_iter_reset()`

```
herr_t
H5Ssel_iter_reset(hid_t sel_iter_id, hid_t space_id);
```

Resets a dataspace selection iterator object back to its initial state so that it may be used once again without needing to create another selection iterator object.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection iterator object is being reset. This function would also be affected by the concurrency issues noted in 3.3. Finally, this function has a few additional concurrency issues specific to it:

- This function releases the current selection on the dataspace iterator object, posing further concurrency issues if the same iterator object is used by multiple threads.
- This function allows the caller to change the dataspace object that the iterator object will iterate over by providing the ID of a different dataspace object than the iterator object was created with in `H5Ssel_iter_create()`. If the iterator object is being used by multiple threads, this is another piece of shared state that has to be handled to prevent threads from having out of date information.

A.31. `H5Sselect_adjust()`

```
herr_t
H5Sselect_adjust(hid_t spaceid, const hssize_t *offset);
```

Adjusts the selection within a dataspace object by subtracting an offset. Essentially a counterpart to `H5Soffset_simple()`, this function allows moving around of a selection in a dataspace object without having to redefine the selection.

Concurrency notes: This function is affected by all the concurrency issues described in sections 3.1, 3.2 and 3.7.2.

Miscellaneous recommendations: This function assumes that the passed in array is at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added to help prevent this issue.

A.32. `H5Sselect_all()`

```
herr_t
H5Sselect_all(hid_t spaceid);
```

Changes the selection in a dataspace object so that all of the elements in its extent are selected.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection in the dataspace is being updated. Further, this function releases the previous selection in the dataspace before updating to an 'all' selection, which could cause the dataspace to be in a problematic intermediate state if it is shared by multiple threads.

A.33. `H5Sselect_copy()`

```
herr_t
H5Sselect_copy(hid_t dst_id, hid_t src_id);
```

Copies the selection in the dataspace object `src_id` into the dataspace object `dst_id`.

Concurrency notes: Unexpected results could occur if the source dataspace's selection is concurrently modified while copying of its selection is taking place.

Unexpected results could also occur if multiple threads attempt to copy a selection into the same destination dataspace object. This case seems unlikely to be of much use or interest and should likely be documented as an invalid operation.

A.34. `H5Sselect_elements()`

```
herr_t
H5Sselect_elements(hid_t space_id, H5S_seloper_t op, size_t num_elem,
                   const hsize_t *coord);
```

Selects individual elements in a dataspace object, resulting in a point selection.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection modification operation is taking place. Further, this function is affected by the issues described in section 3.1. Specifically, if a dataspace object is shared among multiple threads then concurrently selecting points in the dataspace object will be problematic; the linked list of points will need to be updated atomically.

A.35. `H5Sselect_hyperslab()`

```
herr_t
H5Sselect_hyperslab(hid_t space_id, H5S_seloper_t op,
                    const hsize_t start[],
                    const hsize_t stride[],
                    const hsize_t count[],
                    const hsize_t block[]);
```

Selects a hyperslab region within a dataspace object and either replaces the existing selection or refines it according to the selection operator specified in `op`.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection modification operation is taking place. Further, this function is affected by the issues described in section 3.1. Specifically, if a dataspace object is shared among multiple threads then concurrently selecting a hyperslab in the dataspace object will be problematic. Regardless of the chosen selection operator, the resulting selection could be indeterminate as each thread attempts to replace some or all of the existing selection with its own.

This function can also encounter the issue described in section 3.7.2.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.36. `H5Sselect_intersect_block()`

```
htri_t
H5Sselect_intersect_block(hid_t space_id, const hsize_t *start,
                          const hsize_t *end);
```

Returns whether or not the selection within a dataspace object intersects with the block described by the coordinates specified in `start` and `end`.

Concurrency notes: Unexpected results could occur if the source dataspace is concurrently modified while the intersection information is being determined.

This function can also encounter the issue described in section 3.7.2.

Miscellaneous recommendations: This function assumes that the passed in arrays are each at least big enough to hold a number of elements equal to the rank value (number of dimensions) of the passed in dataspace. This presents an obvious memory safety issue which could be made worse in the presence of concurrency. It is recommended that a size parameter be added, either a single parameter covering all arrays or individual parameters, to help prevent this issue.

A.37. H5Sselect_none()

```
herr_t
H5Sselect_none(hid_t spaceid);
```

Changes the selection in a dataspace object so that none of the elements in its extent are selected.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the selection in the dataspace is being updated. Further, this function releases the previous selection in the dataspace before updating to a 'none' selection, which could cause the dataspace to be in a problematic intermediate state if it is shared by multiple threads.

A.38. H5Sselect_project_intersection()

```
hid_t
H5Sselect_project_intersection(hid_t src_space_id, hid_t dst_space_id,
                               hid_t src_intersect_space_id);
```

Computes the intersection between the selections within two dataspace objects and then projects that intersection into a third dataspace object, then returns an ID of the newly-created dataspace object. This function is primarily a helper function for VOL connector authors implementing chunked or virtual datasets.

Concurrency notes: Unexpected results could occur if any of the provided dataspaces are concurrently modified while the selection projection operation is taking place. Further, this function has, at minimum, the issues described in 3.2 and 3.7.2. This function also uses the internal versions of several of the other functions listed in this appendix and likely inherits any concurrency issues specific to them as well.

A.39. H5Sselect_shape_same()

```
htri_t
H5Sselect_shape_same(hid_t space1_id, hid_t space2_id);
```

Returns whether or not the selections within two dataspace objects are the same shape as each other.

Concurrency notes: Unexpected results could occur if either of the source dataspaces are concurrently modified while the comparisons between the two selections are being made.

For hyperslab selections, this function can also encounter the issue described in section 3.7.2.

A.40. H5Sselect_valid()

```
htri_t
H5Sselect_valid(hid_t spaceid);
```

Returns whether or not the current selection in a dataspace object is valid, given its current extent.

Concurrency notes: Unexpected results could occur if the source dataspace is concurrently modified while the validity of its selection is being determined.

A.41. H5Sset_extent_none()

```
herr_t
H5Sset_extent_none(hid_t space_id);
```

Changes the extent of a dataspace object to a null extent type.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the extent in the dataspace is being updated. Further, this function releases the previous extent in the dataspace before updating to a null extent, which could cause the dataspace to be in a problematic intermediate state if it is shared by multiple threads.

A.42. H5Sset_extent_simple()

```
herr_t
H5Sset_extent_simple(hid_t space_id, int rank, const hsize_t dims[],
                     const hsize_t max[]);
```

Changes the extent of a dataspace object to a simple extent.

Concurrency notes: Unexpected results could occur if the dataspace is concurrently modified while the extent in the dataspace is being updated. Further, this function releases the previous extent in the dataspace before updating to a simple extent, which could cause the dataspace to be in a problematic intermediate state if it is shared by multiple threads.

B. Appendix: H5S private functions

The following sections give an overview of concurrency issues noted for functions which are internal to the H5S interface and may be called by other parts of the HDF5 library, but are otherwise not exposed publicly.

This section still under construction

C. Appendix: H5S interface data structures

The following sections give an overview of the data structures in the H5S interface and outline any concurrency issues that will have to be dealt with for each structure. Note that initial sketches of revisions proposed in the following sections are based off of the approach proposed in section 4.5, so readers should be familiar with those details first.

C.1. H5S_t

```
struct H5S_t {
    H5S_extent_t extent; /* Dataspace extent (must stay first) */
    H5S_select_t select; /* Dataspace selection */
};
```

H5S_t is the main structure representing a dataspace object. Concurrency issues with the structure lie within the extent and select fields; the structure itself doesn't have any inherent concurrency issues at the top level. Based on the initial sketch for adopting an RCU-like approach for solving H5S concurrency issues, a revised H5S_t structure may look like the following:

```
typedef struct H5S_kernel_t {
    uint32_t reader_count;
    bool     struct_replaced;
} H5S_kernel_t;

struct H5S_t {
    H5S_extent_t extent; /* Dataspace extent (must stay first) */
    H5S_select_t select; /* Dataspace selection */
    _Atomic H5S_kernel_t kernel;
};
```

The new H5S_kernel_t structure field would essentially be used to reference count an instance of the structure in order to determine when it can be garbage collected. The reader_count field serves as the reference count. When it drops to 0, the structure is allowed to be reclaimed/de-allocated. The struct_replaced field serves as a flag that specifies whether the structure has actually been replaced with a new version by a writer. If the structure hasn't been replaced (and therefore the original is still the current version), then the structure doesn't need to be de-allocated as it is still in use. The flag would initially be set to false and would be changed to true when a writer replaces a version of H5S_t with a new copy.

C.2. H5S_extent_t

```
struct H5S_extent_t {
    H5O_shared_t sh_loc; /* Shared message info (must be first) */

    H5S_class_t type;    /* Type of extent */
    unsigned    version; /* Version of object header message to
                            encode this object with */
```

```
    hsize_t     nelem;    /* Number of elements in extent */

    unsigned rank; /* Number of dimensions */
    hsize_t *size; /* Current size of the dimensions */
    hsize_t *max;  /* Maximum size of the dimensions */
};
```

`H5S_extent_t` is the data structure for the extent of a dataspace object.

The `sh_loc` field can be shared among dataspace object header messages, meaning that an investigation of HDF5's object header sharing mechanism is likely needed to understand the effect of concurrency here.

The `version` field is set at dataspace object creation time and is only ever modified by the function `H5S_set_version()`, which is only called on copies of existing dataspaces in `H5A__create()` and `H5D__init_space()`.

All remaining fields are vulnerable to concurrency issues if the extent of a dataspace is modified concurrently. See Appendix D for details on when these fields might be modified.

C.3. `H5S_select_t`

```
typedef struct {
    const H5S_select_class_t *type; /* Pointer to selection's class info */

    hbool_t  offset_changed;       /* Indicate that the offset for the
                                      selection has been changed */
    hssize_t offset[H5S_MAX_RANK]; /* Offset within the extent */

    hsize_t num_elem; /* Number of elements in selection */

    union {
        H5S_pnt_list_t  *pnt_lst; /* Info about list of selected points
                                     (order is important) */
        H5S_hyper_sel_t *hslab;   /* Info about hyperslab selection */
    } sel_info;
} H5S_select_t;
```

`H5S_select_t` is the data structure for the selection within a dataspace object.

The `type` field always points to one of the 'class' structures `H5S_sel_all`, `H5S_sel_none`, `H5S_sel_hyper` or `H5S_select_class_t`, which should be unchanging. However, this pointer can change at any time as the type of selection set within a dataspace object changes.

The `offset` and `offset_changed` fields are primarily updated by the `H5Soffset_simple()` function, though they are also reset to 0 and `false`, respectively, by the internal `H5S_set_extent_simple()` function.

The `num_elem` field is updated any time that the selection in a dataspace changes.

The `sel_info` field contains the selection information for a dataspace object which could end up being shared between dataspaces in the situations described in 3.2. This field is only valid when the selection in the dataspace is either a point or hyperslab selection and is updated if changing between the two, or if the selection in the dataspace is being released.

See Appendix D for details on when these fields might be modified.

C.4. H5S_hyper_sel_t

```
typedef struct {
    H5S_diminfo_valid_t diminfo_valid; /* Whether the dataset has valid
                                          diminfo */

    H5S_hyper_diminfo_t diminfo;            /* Dimension info form of
                                               hyperslab selection */
    int                 unlim_dim;          /* Dimension where selection
                                               is unlimited, or -1 if none */
    hsize_t             num_elem_non_unlim; /* # of elements in a "slice"
                                               excluding the unlimited
                                               dimension */
    H5S_hyper_span_info_t *span_lst;        /* List of hyperslab span
                                               information of all
                                               dimensions */
} H5S_hyper_sel_t;
```

`H5S_hyper_sel_t` is the data structure that maintains information about a hyperslab selection within a dataspace object. Each of the fields in this structure can be modified as a hyperslab selection is set on a dataspace or an existing hyperslab selection in the dataspace is refined.

C.5. H5S_hyper_span_info_t

```
struct H5S_hyper_span_info_t {
    unsigned count; /* Ref. count of number of spans which share this span */

    /* The following two fields define the bounding box of this set of spans
     *  and all lower dimensions, relative to the offset.
     */
    /* (NOTE: The bounds arrays are _relative_ to the depth of the span_info
     *          node in the span tree, so the top node in a 5-D span tree will
     *          use indices 0-4 in the arrays to store it's bounds information,
     *          but the next level down in the span tree will use indices 0-3.
     *          So, each level in the span tree will have the 0th index in the
     *          arrays correspond to the bounds in "this" dimension, even if
     *          it's not the highest level in the span tree.
     */
    hsize_t *low_bounds;  /* The smallest element selected in each dimension */
```

```
    hsize_t *high_bounds; /* The largest element selected in each dimension */

    /* "Operation info" fields */
    /* (Used during copies, 'adjust', 'nelem', and 'rebuild' operations) */
    /* Currently the maximum number of simultaneous operations is 2 */
    H5S_hyper_op_info_t op_info[2];

    struct H5S_hyper_span_t *head; /* Pointer to the first span of list of
                                      spans in the current dimension */
    struct H5S_hyper_span_t *tail; /* Pointer to the last span of list of
                                      spans in the current dimension */
    hsize_t                  bounds[]; /* Array for storing low & high bounds */
                                       /* (NOTE: This uses the C99 "flexible
                                          array member" feature) */
};
```

`H5S_hyper_span_info_t` is the data structure for representing the span tree information for a hyperslab selection.

C.6. H5S_pnt_list_t

```
struct H5S_pnt_list_t {
    /* The following two fields defines the bounding box of the whole
       set of points, relative to the offset */
    hsize_t low_bounds[H5S_MAX_RANK];  /* The smallest element selected
                                          in each dimension */
    hsize_t high_bounds[H5S_MAX_RANK]; /* The largest element selected
                                          in each dimension */

    H5S_pnt_node_t *head; /* Pointer to head of point list */
    H5S_pnt_node_t *tail; /* Pointer to tail of point list */

    hsize_t last_idx; /* Index of the point after the last returned from
                         H5S__get_select_elem_pointlist() */
    H5S_pnt_node_t *last_idx_pnt; /* Point after the last returned from
                                   * H5S__get_select_elem_pointlist().
                                   * If we ever add a way to remove points
                                   * or add points in the middle of
                                   * the pointlist we will need to invalidate
                                   * these fields. */
};
```

`H5S_pnt_list_t` is the data structure that maintains information about a point selection within a dataspace object.

The `head` and `tail` fields keep track of the head and tail of the linked list of currently selected points. If concurrent selection of points within a dataspace is allowed, this linked list structure should be able to

be converted into a lock-free linked list to keep close to the original design. However, keep in mind that performance problems have previously been reported for HDF5's current linked list implementation of point selections, so a different data structure may be warranted.

See 3.8 for a discussion on concurrency issues with the `last_idx` and `last_idx_pnt` fields.

The `low_bounds` and `high_bounds` fields are used to keep track of a bounding box around the currently selected points in a dataspace and present an issue for concurrent modification of point selections. If the list of points selected in a dataspace object is concurrently modified, there needs to be some form of coordination for updating the bounding box once all points have been selected. These fields are currently updated on a point-by-point basis as each individual point is selected, which could potentially be done atomically on a point-by-point basis, but may be prohibitively expensive to do so.

C.7. H5S_pnt_node_t

```
struct H5S_pnt_node_t {
    struct H5S_pnt_node_t *next;   /* Pointer to next point in list */
    hsize_t                pnt[];  /* Selected point */
                                   /* (NOTE: This uses the C99 "flexible
                                      array member" feature) */
};
```

`H5S_pnt_node_t` is the data structure for a single point selected within a dataspace object's extent. Provided that a lock-free linked list is used for the main `H5S_pnt_list_t` structure, adapting usage of this structure for concurrency should be relatively straightforward. Otherwise, further design discussion will be needed.

C.8. H5S_sel_iter_t

```
typedef struct H5S_sel_iter_t {
    /* Selection class */
    const struct H5S_sel_iter_class_t *type; /* Selection iteration
                                                class info */

    /* Information common to all iterators */
    unsigned rank;                   /* Rank of dataspace the selection
                                        iterator is operating on */
    hsize_t  dims[H5S_MAX_RANK];     /* Dimensions of dataspace the selection
                                        is operating on */
    hssize_t sel_off[H5S_MAX_RANK]; /* Selection offset in dataspace */
    hsize_t  elmt_left;              /* Number of elements left to iterate
                                        over */
    size_t   elmt_size;              /* Size of elements to iterate over */
    unsigned flags;                  /* Flags controlling iterator behavior */

    /* Information specific to each type of iterator */
    union {
```

```
        H5S_point_iter_t pnt; /* Point selection iteration information */
        H5S_hyper_iter_t hyp; /* New Hyperslab selection iteration
                                 information */
        H5S_all_iter_t   all; /* "All" selection iteration information */
    } u;
} H5S_sel_iter_t;
```

`H5S_sel_iter_t` is the data structure used for dataspace selection iterator objects.

The `type` field always points to one of the 'class' structures `H5S_sel_all`, `H5S_sel_none`, `H5S_sel_hyper` or `H5S_select_class_t`, which should be unchanging. However, this pointer can change at any time as the dataspace object set for a selection iterator changes via `H5Ssel_iter_reset()`.

The `rank`, `dims`, `sel_off`, `elmt_size` and `flags` fields are generally unchanging after selection iterator creation. However, a concurrent call to `H5Ssel_iter_reset()` could modify these if an iterator object is shared between threads.

The `elmt_left` field is updated during each call to `H5Ssel_iter_get_seq_list()` on an iterator object. If multiple threads are calling that function on the same iterator object, this field will be a point of contention. However, whether or not there is a reasonable use case for multiple threads to share the same iterator object remains to be seen.

Each of the fields in the `u` union contain information that is updated during each call to the function `H5Ssel_iter_get_seq_list()` on an iterator object. If multiple threads are calling that function on the same iterator object, this information will be a point of contention. However, whether or not there is a reasonable use case for multiple threads to share the same iterator object remains to be seen.

D. Appendix: Fields modified by API functions

The following sections give an overview of the fields modified in a `H5S_t` structure by API functions in the H5S interface. This information is somewhat similar to that covered in Appendix A, but is gathered here for now with more details and notes for quicker review.

D.1. 'Get'-style functions

The following functions are considered 'get'-style functions, in that their primary purpose is to either retrieve information from a dataspace or return a new dataspace without modifying one that may be passed in to the function. Therefore, any modifications made to a dataspace within one of these functions presents an obvious concurrency issue to be addressed.

D.1.1. `H5Screate()`

Fields modified: None

Notes: No dataspace ID passed in

D.1.2. `H5Screate_simple()`

Fields modified: None

Notes: No dataspace ID passed in

D.1.3. `H5Scombine_hyperslab()`

Fields modified: `select.sel_info.hslab->span_lst`

Notes: This field may or may not be modified, but a much more thorough investigation is needed. This function may internally copy the old dataspace while sharing the span tree with the new copy, meaning that further operations in the internal call to `H5S_select_hyperslab()` have the potential to modify the span tree of the original dataspace passed in. This may not actually occur, but the surface to be audited is fairly large.

D.1.4. `H5Scombine_select()`

Fields modified: `select.sel_info.hslab->span_lst`

Notes: This function may call the internal `H5S_combine_hyperslab()`, so can have similar issues as above.

D.1.5. `H5Sdecode()`

Fields modified: None

Notes: No dataspace ID passed in

D.1.6. `H5Sencode2()`

Fields modified: `select.sel_info.hslab->span_lst->op_info[0]`
`select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the ”operation info” field inside a structure while trying to cache the results of a hyperslab operation. See section 3.7.2. Can also result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.7. `H5Sextent_equal()`

Fields modified: None

Notes:

D.1.8. `H5Sselect_intersect_block()`

Fields modified: `select.sel_info.hslab->span_lst->op_info[0]`
`select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the ”operation info” field inside a structure while trying to cache the results of a hyperslab operation. See section 3.7.2. Can also result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.9. `H5Sselect_project_intersection()`

Fields modified: `select.sel_info.hslab->span_lst->op_info[0]`
`select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the ”operation info” field inside a structure while trying to cache the results of a hyperslab operation. See section 3.7.2. Can also result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.10. `H5Scopy()`

Fields modified: None

Notes:

D.1.11. H5Sget_regular_hyperslab()

Fields modified: `select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.12. H5Sget_select_bounds()

Fields modified: None

Notes:

D.1.13. H5Sget_select_elem_npoints()

Fields modified: None

Notes:

D.1.14. H5Sget_select_elem_pointlist()

Fields modified: `space->select.sel_info.pnt_lst->last_idx`
`space->select.sel_info.pnt_lst->last_idx_pnt`

Notes: Modifies values in point selection information structure to cache the last point that was visited and try to speed up subsequent iteration operations.

D.1.15. H5Sget_select_hyper_nblocks()

Fields modified: `select.sel_info.hslab->span_lst->op_info[0]`

Notes: Can result in modifying the ”operation info” field inside a structure while trying to cache the results of a hyperslab operation. See section 3.7.2.

D.1.16. H5Sget_select_hyper_blocklist()

Fields modified: `select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.17. H5Sget_select_npoints()

Fields modified: None

Notes:

D.1.18. `H5Sget_select_type()`

Fields modified: None

Notes:

D.1.19. `H5Sget_simple_extent_ndims()`

Fields modified: None

Notes:

D.1.20. `H5Sget_simple_extent_dims()`

Fields modified: None

Notes:

D.1.21. `H5Sget_simple_extent_npoints()`

Fields modified: None

Notes:

D.1.22. `H5Sget_simple_extent_type()`

Fields modified: None

Notes:

D.1.23. `H5Sis_regular_hyperslab()`

Fields modified: `select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in modifying the dimension info fields for a dataspace if the library needs to ”rebuild” a regular selection. See section 3.7.1.

D.1.24. `H5Sis_simple()`

Fields modified: None

Notes:

D.1.25. H5Sselect_shape_same()

Fields modified: `select.sel_info.hslab->span_lst`
`select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Can result in creating a span tree for the dataspace(s) if they don't already have one. Can also result in modifying the dimension info fields for a dataspace if the library needs to "rebuild" a regular selection. See section 3.7.1.

D.1.26. H5Sselect_valid()

Fields modified: None

Notes:

D.2. 'Set'-style functions

The following functions are considered 'set'-style functions, in that their primary purpose is to modify some portion, the extent or selection or both, of a dataspace.

D.2.1. H5Sclose()

Fields modified: `extent`, `select`

Notes: Frees `extent.size` and `extent.max` and sets `extent.rank` and `extent.nelem` to 0.

Performs selection-specific cleanup on `select` field. 'all', point and hyperslab selections set `select.num_elem` to 0 ('none' selections already have this). Point selections free `select.sel_info.pnt_lst` and set it to `NULL`. Hyperslab selections free the span tree information in `select.sel_info.hslab->span_lst` then free `select.sel_info.hslab`.

D.2.2. H5Sextent_copy()

Fields modified: `extent`, `select` (in destination dataspace)

Notes: In destination dataspace, frees `extent.size` and `extent.max`, then re-allocates them and copies all values over from source dataspace's `extent` field. If the selection in the destination dataspace is an 'all' selection, also updates `select.num_elem` in the destination dataspace.

D.2.3. H5Smodify_select()

Fields modified: `select` (in destination dataspace `space1_id`)

Notes: Depending on the path through the hyperslab code, each field in the `select` field may be modified, other than `select.offset` and `select.offset_changed`. Any modifications to the union field `sel_info` will be made to `sel_info.hslab`.

D.2.4. H5Soffset_simple()

Fields modified: `select`

Notes: Copies the passed in offset into `select.offset` and then sets `select.offset_changed` to `true`.

D.2.5. H5Sselect_adjust()

Fields modified: `select`

Notes: Calls selection type-specific callback 'adjust_s' for adjusting the selection within a dataspace. For 'all' and 'none' selections, does nothing. For point selections, iterates through each selected point (`select.sel_info.pnt_lst->head->pnt` / `...->head->next->pnt`) and adjusts the location of each point by subtracting the passed in offset from the point coordinates. Then, updates the bounding box of the selection in `select.sel_info.pnt_lst->low_bounds` and `select.sel_info.pnt_lst->high_bounds`. For hyperslab selections, updates the regular selection information in `select.sel_info.hslab->diminfo.opt`, `select.sel_info.hslab->diminfo.low_bounds` and `select.sel_info.hslab->diminfo.high_bounds`, if available. Also updates the span tree information in the nodes in `select.sel_info.hslab->span_lst`, if available.

D.2.6. H5Sselect_all()

Fields modified: `select`

Notes: Releases selection in dataspace by performing selection-specific cleanup on `select` field (see entry for `H5Sclose()` in this section) and then sets `select.type` to point to `H5S_sel_all` structure. Updates `select.num_elem` field to the number of points in the dataspace's extent.

D.2.7. H5Sselect_copy()

Fields modified: `select` (in destination dataspace)

Notes: Releases selection in destination dataspace by performing selection-specific cleanup on `select` field (see entry for `H5Sclose()` in this section). Then, all fields from source dataspace `select` field are copied into destination dataspace `select` field. Finally, selection type-specific adjustments are made to `select` field of destination dataspace. For 'all' selections, `select.num_elem` is updated to the number of points in the destination dataspace's extent. For 'none' selections, `select.num_elem` is set to 0. For point selections, the list of selected points in the source dataspace is copied into `select.sel_info.pnt_lst` in the destination dataspace. For hyperslab selections, a new instance of `H5S_hyper_sel_t` is allocated and eventually assigned to `select.sel_info.hslab`. If the source dataspace has span tree information, a copy is made into `select.sel_info.hslab->span_lst` in the destination dataspace.

D.2.8. H5Sselect_elements()

Fields modified: `select`

Notes: If current selection in dataspace isn't a point selection, or if a point selection is being set with the H5S_SELECT_SET selection operator, releases selection in dataspace by performing selection-specific cleanup on select field (see entry for H5Sclose() in this section). Next, allocates a new instance of H5S_pnt_list_t in select.sel_info.pnt_lst if the dataspace didn't already have a point selection within it. Then, points are added to nodes in select.sel_info.pnt_lst->head / ...->head->next, etc. The bounding box of the selection is updated in select.sel_info.pnt_lst->low_bounds and select.sel_info.pnt_lst->high_bounds as points are added. Once all points have been added, select.num_elem is updated according to the points selected. Finally, select.type is set to point to the H5S_sel_point structure.

D.2.9. H5Sselect_hyperslab()

Fields modified: select

Notes: Depending on the specific operation used, releases selection in dataspace by performing selection-specific cleanup on select field (see entry for H5Sclose() in this section) and then allocates a new instance of H5S_hyper_sel_t in select.sel_info.hslab. In this case, select.type is changed to point to the H5S_sel_hyper structure. Otherwise, essentially tries to combine a hyperslab selection into the existing selection. This could result in the selection being converted into a 'none' selection (see entry for H5Sselect_none() in this section). Depending on the path through the hyperslab code, each field in the select field may be modified, other than select.offset and select.offset_changed.

D.2.10. H5Sselect_none()

Fields modified: select

Notes: Releases selection in dataspace by performing selection-specific cleanup on select field (see entry for H5Sclose() in this section) and then sets select.type to point to H5S_sel_none structure.

D.2.11. H5Sset_extent_none()

Fields modified: extent, select

Notes: Frees extent.size and extent.max and sets extent.rank and extent.nelem to 0. Sets extent.type to H5S_NULL.

D.2.12. H5Sset_extent_simple()

Fields modified: extent, select

Notes: Frees extent.size and extent.max and sets extent.rank and extent.nelem to 0. Then, performs extent type-specific setup on the dataspace. For scalar dataspaces (when the passed in rank parameter is 0), sets extent.type to H5S_SCALAR and sets extent.nelem to 1. For simple dataspaces, sets extent.type to H5S_SIMPLE, re-allocates extent.size and extent.max and copies over passed in values. extent.nelem is set to the calculated number of elements.

For both scalar and simple dataspaces, `select.offset` is set to zeroes and `select.offset_changed` is set to `false`. If the selection in the dataspace was an 'all' selection, `select.num_elem` is updated to reflect that all elements in the new extent are selected.

D.3. Other functions

The following functions are in a separate category due to primarily operating on a different object than a dataspace. However, a dataspace has to be passed in to some of these functions initially, which could have secondary effects on those dataspaces.

D.3.1. `H5Ssel_iter_create()`

Fields modified: `select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Dataspace selection iterators call an 'init' callback when being created which may operate on the passed in dataspace. Can result in modifying the dimension info fields for a dataspace if the library needs to "rebuild" a regular selection. See section 3.7.1.

D.3.2. `H5Ssel_iter_get_seq_list()`

Fields modified: None

Notes:

D.3.3. `H5Ssel_iter_reset()`

Fields modified: `select.sel_info.hslab->diminfo_valid`
`select.sel_info.hslab->diminfo`

Notes: Dataspace selection iterators call an 'init' callback when being reset which may operate on the passed in dataspace. Can result in modifying the dimension info fields for a dataspace if the library needs to "rebuild" a regular selection. See section 3.7.1.

D.3.4. `H5Ssel_iter_close()`

Fields modified: None

Notes: