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

TBD

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).

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. 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.

Revision History

Version Number	Date	Comments
v1	Jan. 24, 2025	Version 1 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 3.7.1.

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 3.7.1.

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 3.7.1.

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 3.7.1. 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.

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.

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.

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.

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 3.7.1.

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.

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.

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 3.7.1.

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 3.7.1.

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 3.7.1.

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 3.7.1. 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 3.7.1.

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.

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. As the structure simply contains two other structures which are the focus of concurrency issues in H5S and which are on the larger side (80 bytes and 288 bytes, respectively), initial effort will explore pushing thread-safety down into those structures rather than here, unless another approach proves better.

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 H5S_set_version() function, 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.

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.

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 `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.