RFC: Selection I/O

Neil Fortner
Jordan Henderson
John Mainzer

The VFD (Virtual File Driver) abstraction layer supported by the HDF5 library provides read and write operations that are similar to the POSIX I/O operations of the same name. While this has been adequate for the serial version of HDF5, it has presented problems for parallel HDF5 as it prevents the VFD from seeing the entire I/O request generated by an API call all at once, and thus prevents the use of MPI I/O optimizations on the entire I/O request. To date, this has been finessed by generating the required MPI derived types at higher levels in the HDF5 library, and then passing them down to the VFD layer via un-documented channels.

As long as MPI I/O was the only place that such optimizations were needed, the above situation, while not desirable, was workable. However, there are now at least three more cases where allowing the VFD to view and optimize the entire I/O request generated by an HDF5 API call is essential for high performance.

This RFC proposes extensions to the VFD interface necessary to expose entire I/O requests to the VFD where desired, and describes the associated architectural changes required in the HDF5 library. As shall be seen, these changes are not trivial. However, in addition to allowing efficient I/O in cases where that is not currently possible, they should also allow significant simplification of raw data I/O pathways in the upper levels of the HDF5 library.

1 Introduction

Conceptually, the VFD (Virtual File Driver) layer presents the underlying storage system as an extensible array of bytes, and hides most of the implementation details from the upper levels of the HDF5 library. Historically, this abstraction layer has been used to allow the HDF5 library to run on different operating systems with different file I/O APIs, to simulate large files on file systems with a 2 GB max file size, or to segregate metadata and raw data into separate files. More recent applications include supporting object stores, remote mirroring of HDF5 files as they are written, and an alternate implementation of SWMR (Single Writer Multiple Readers).

While the value of this abstraction layer has been well demonstrated, the supported read and write calls (signatures shown below) are essentially the UNIX system calls of the same the name with the addition of parameters specifying memory type, and an arbitrary property list.

```
herr_t H5FDread(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                haddr_t addr, size_t size, void *buf/*out*/);

herr_t H5FDwrite(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                 haddr_t addr, size_t size, const void buf /*in*/);
```

As can be seen from the above signatures, the I/O calls supported by the VFD interface require the upper levels of the HDF5 library to break the I/O requests generated by any given HDF5 API call into a sequence of type, offset, length, buffer quadruples, and pass them to the VFD layer in individual calls. This is a good match for most local file systems, but it is inadequate in the following contexts:

1. In MPI I/O, significant performance gains can be realized by bundling the entire set of I/O requests generated by an HDF5 API call into a MPI derived type, and passing it to MPI I/O in a single function call.
2. In the S3 VFD, network latency is a major concern. When circumstances permit, running multiple I/O requests concurrently instead of sequentially would allow the S3 VFD to provide significantly higher effective bandwidth.
3. Implementing a flexible version of sub-filing that is largely hidden from the upper levels of the HDF5 library requires access to the entire I/O request so that it can be broken up by sub-file and relayed to the appropriate I/O concentrator. See the VFD Sub-Filing RFC for details.
4. In recent work on topology aware I/O at ANL, the VFD requires access to the entire I/O request so that it can aggregate and re-locate I/O operations as needed to optimize the I/O across the members of the computation

The MPI I/O case has already been addressed via the expedient of constructing the necessary MPI derived types at higher levels of the HDF5 library and passing them down to the MPI I/O VFD via undocumented channels. However, this approach has the dual disadvantages of complicating raw data I/O pathways at higher levels of the library, and requiring code modifications that are not applicable to the similar cases described above.

An obvious way of addressing this limitation is to augment the existing read and write VFD calls with versions that accept vectors of types, addresses, sizes, and buffers. This is shown below in the signatures of the proposed `H5FDread_vector()` and `H5FDwrite_vector()` calls. Here the `count` parameter contains the length of the `types`, `addrs`, `sizes`, and `bufs` parameters.

```
herr_t H5FDread_vector(H5FD_t *file, hid_t dxpl_id, size_t count,
                       H5FD_mem_t types[], haddr_t addrs[], size_t sizes[],
                       void * bufs[] /*out*/);

herr_t H5FDwrite_vector(H5FD_t *file, hid_t dxpl_id, size_t count,
                        H5FD_mem_t types[], haddr_t addrs[], size_t sizes[],
                        const void *bufs[] /* in */);
```

To allow more efficient description of I/O requests in which all elements are of the same size and/or type, we allow the `types` and `sizes` arrays to be of length less than `count`. If they are, the arrays must be of length two, and the second element must be invalid – zero for `sizes`, and `H5FD_MEM_NOLIST` for `types`. If the invalid value exists at index 1 in one of these arrays, that entry and all subsequent entries are presumed to equal `sizes[0]` or `types[0]` as appropriate.[1]

Vector I/O is a conceptually simple solution, and is well suited both to the I/O requirements of the HDF5 metadata cache, the page buffer, the chunk cache, and also to the needs of VFDs targeting object stores.

However, it has a couple of major problems for some I/O requests on datasets – in particular when hyper-slab selections are used.

The most obvious of these is memory requirements – for example consider the footprint of a vector I/O representation of a read of every tenth element of a million-member vector vs. a regular hyper-slab selection describing the same I/O request.

The second issue is more subtle, yet quite important in the parallel case. Unlike vector representation, regular or near regular hyper-slab selections retain much of the structure of the I/O request. This in turn allows MPI derived types constructed for one part of a selection to be reused when the same pattern appears elsewhere – with obvious memory footprint and potential performance advantages.

To address these issues at minimal cost, it is expedient to introduce the selection I/O VFD calls which use the existing selection mechanism to describe the complete set of I/O requests generated by a given HDF5 API call. Proposed signatures are shown below:

```
herr_t H5FDread_selection(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                          size_t count, hid_t mem_spaces[],
                          hid_t file_spaces[], haddr_t offsets[],
                          size_t element_sizes[],void * bufs[] /*out*/);

herr_t H5FDwrite_selection(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                           size_t count, hid_t mem_spaces[],
                           hid_t file_spaces[], haddr_t offsets[],
                           size_t element_sizes[],
                           const void * bufs[] /*in*/);
```

Note that these calls take vectors of selections (or more correctly, vectors of `hid_t`'s that map to selections[2]) of length `count`. While multiple selections can be combined into single selections, doing so would degrade performance and increase memory footprint in some cases.

[1] This optimization exists to allow efficient representation of point selections in vector I/O format. Note change from earlier version where the invalid value could appear at any index greater than zero.
[2] Quincey suggests extending this to allow the `H5S_ALL` and `H5S_BLOCK` special values for the memory dataspace ID values. He also suggests that it may be useful to add optimizations for efficient representation of cases in which all memory space IDs or the buffer pointers are all identical.

The `type` parameter is a scalar, as for now at least, we don't have a use case in which a selection I/O call could refer to multiple memory types[3].

Each element of the `offsets` array contains an offset to be applied to each element in the associated file space to compute its actual base address in the file. While not strictly necessary, this field should allow us to reduce overhead for some internal calls. If it is `NULL`, all offsets are presumed to be zero.

One can argue that the `element_sizes` should all be one, allowing us to omit this parameter. While we may revisit this question, for now at least, the feeling is that this would increase processing overhead to no purpose.

Since cases in which all the elements of the `bufs[]` array are identical are reasonably common, we allow the same optimization as for the `types[]` and `sizes[]` arrays in the vector I/O calls. Specifically, we allow the `bufs[]` array to be of length less than `count`. If it is, it must be of length two, and the second element must be invalid – `NULL` in this case.

Unlike the vector read/write calls, the selection read/write calls introduce a great deal of complexity into the matter of writing a HDF5 VFD. For example, in the MPI I/O case, the VFD will require an intimate knowledge of the HDF5 selection mechanism to function efficiently. As this RFC is intended in part to serve as a manual for writing such VFDs, much of it must be devoted to documenting the selection facility to the extent necessary. Further, where the necessary operations are not possible with the current public selection interface, this RFC must propose the necessary extensions.

In cases where there is no great benefit in presenting the VFD with the entire set of I/O requests generated by an HDF5 API call in a single VFD call, we can (and must) hide this added complexity from such VFDs by augmenting the top level VFD code to translate the vector and / or selection I/O calls into sequences of simple read or write calls when the target VFD doesn't support them.[4]

Considering the proposed selection I/O API more generally, we should note that at least for our immediate needs, we only really need to support hyper slab selections, as both point and all selections can be translated into vector I/O format at minimal cost.

The primary advantage of this approach is reduced complexity for VFDs that support selection I/O, and an increased fraction of I/O requests that can be handled without translation for VFDs that support vector I/O only.

On the down side, in addition to the aesthetic argument that we should implement selection I/O proper for all types of selections, there is also the point that doing so will be convenient for a selection I/O based implementation of multi-dataset I/O[5]. Note, however, that a full implementation

[3] Indeed, we don't have a use case in which the memory type is anything other than `H5FD_MEM_DRAW` – which is an argument for omitting this parameter entirely.

[4] There is also the possibility that a VFD may support vector I/O, but not selection I/O – in this case, the top level VFD code must translate selection I/O calls into a sequence of vector I/O calls. This case will likely be common, as to date we are only aware of three cases that require the finely detailed view of the I/O operation that the selection I/O calls provide, and thus can be more easily managed with the vector I/O calls.

[5] Recall that multi-dataset I/O aims to improve bandwidth by combining read or write requests for multiple data sets, and executing them in a single I/O request.

of selection I/O doesn't solve this problem completely, as we would also have to require the chunk cache and parallel compression use selection I/O as well.

While we lean towards implementing only hyper-slab selections for at least the initial implementation, the matter is not settled. Given time and resource constraints, the decision will likely be driven by ease of initial implementation. If we do choose to implement selection I/O for hyper-slab selections only, perhaps better names for these functions would be `H5FDread_hs_selection()` and `H5FDwrite_hs_selection()`.

1.1 Data Type Conversions

Data type conversions raise some interesting issues with regards to selection I/O proper, as to obtain the desired efficiencies, the conversions must be done on the entire write before it is passed to the VFD layer, and to the entire read after it has been returned by the VFD layer. In cases where it is either impossible (or impolite) to perform the type conversions in place, the heap space requirements may be so large as to be un-acceptable[6].

In such cases, the vectors of selections must be divided into pieces of manageable size, with data type conversion and I/O performed on each piece sequentially.

This, of course, reduces the effectiveness which which VFDs can exploit knowledge of the entire I/O request for purposes of optimization. As the choice to perform data type conversions and to limit the size of the conversion buffer is the user's, it should be sufficient to handle this case correctly, and to document the performance implications.

To complicate matters further, in the parallel case, the number of pieces the vector of selections is divided into need not be the same for all ranks – which presents a problem for collective I/O. Aside from simply breaking collective, one way to handle this is to inform the VFD when a selection in a collective operation must be handled in pieces, and the local number of pieces required. With this information, the VFD could coordinate with its peers on other ranks to determine the maximum number of I/O requests the collective operation has been divided into, and generate NULL I/O requests as necessary to avoid a hang.

A final issue has been raised by the recent prototype NVIDIA GPU Direct Storage (GDS) VFD. Given the cost of moving data between GPU and CPU RAM, there is considerable incentive to allocate GPU memory for the conversion buffer for data that resides in GPU memory. Thus in such cases, it may be useful to allow the VFD to allocate the conversion buffer.

Observe that all of these issues can be addressed by moving data type conversion to the top of the VFD layer – that is the portion of the VFD code just above all calls into the VFD proper.

This approach has the following major advantages:

[6] Data transformations are conceptually different from data type conversions, and at first glance deserve a separate discussion. However, since data transformations occur just before type conversions on write, and just after on reads, for purposes of this discussion, they can be thought of as a variation on a theme.

1. Code managing large collective I/O calls that have to be broken up due to data type conversion buffer size limitations is centralized[7], and hidden from the upper regions of the library.
2. It allows us to permit a VFD (such as the GDS VFD mentioned above) to allocate the data type conversion buffer if desired.

Whether the proposed re-factoring is worth doing is another question. I expect that it will come down to the perceived value of the above listed advantages. In any case, we are unlikely to address this in the initial implementation.

2 Architectural Changes in HDF5

At a conceptual level, the architectural changes necessary to implement and use selection and vector I/O in the HDF5 library are simply a matter of refactoring and re-organizing existing code. While this is certainly true, it glosses over the magnitude to the effort required, as the current management of MPI I/O has been in place for more than a decade, and is thus heavily embedded in the current architecture of HDF5.

While in principle, it would be possible to construct a detailed list of all changes necessary, the author's experience with such modifications suggests that this would be costly, and error prone as significant experimentation is frequently necessary when modifying complex and largely un-documented code bases. Instead, the remainder of this section attempts to list the effected sections of the HDF5 library, and outline the required changes with as much specificity as can be attained with reasonable effort.

As currently understood, the effected areas are:

- Data set code
- Metadata Cache
- Page Buffer
- VFD layer – which can be divided into:
    - Upper level prior to VFD calls proper
    - API extensions needed for efficient traversal of selections by VFDs
    - Support for intersections of selections with sub-files
    - Support for efficient serialization / deserialization of selections (needed for sub-filing, and for a more efficient implementation of mirror VFD).
    - Modification of the MPIO VFD to support vector and selection I/O
- Miscellaneous changes -- needed to adjust to movement of most MPI I/O specific code to the MPIO VFD

[7] Filtered chunks are an exception to this, as any type conversions will have to be performed before a chunk is run through the filter pipeline on write, and after the pipeline on read.

Before addressing these areas individually, a few generic comments are in order.

First, at least for the initial implementation, we need to minimize the cost of the development effort. Thus, for example, we should only move the data type conversion code to the top of the VFD layer it it makes things easier in the first cut.

Second, we should only be using selection I/O where it has the potential to buy us something – specifically either ease of implementation, or a more space and/or time efficient representation of an I/O request.

Third, where practical, we should leave existing I/O pathways in place until we can demonstrate that selection I/O imposes no significant performance degradation.

2.1 Data Set Code

Selection I/O at least has the potential to greatly reduce the memory footprint of the data structure required to describe a regular selection – recall the example of a vector I/O representation of a read of every tenth element of a million-member vector vs. a regular selection describing the same I/O request.

Since point selections are quite similar to vector I/O and don't offer any of the heap space optimizations of regular hyper-slab selections, converting them to vector I/O for I/O purposes is arguably the easiest solution. If we ever extend point selections to support regular point selections, we will have to re-visit this.

Similarly, "All" selections are also a good fit for vector I/O.

Parallel compression, and the chunk cache (at least for filtered chunks) deal with a relatively small number of relatively large, contiguous regions of the HDF5 file. Thus vector I/O is a good match here as well.

Thus in a nut shell, for the data set code, the plan is to implement selection I/O proper where we support hyper-slab selections and don't use filters, and use vector I/O everywhere else. Note that this decision is driven by presumed ease of implementation, and will be revisited if this presumption turns out to be incorrect. Even it it is correct, we may have to re-visit the issue to simplify multi-dataset I/O implementation.

For now, the plan is to leave data type conversion more or less where it is – although we hope to retain the option of moving it at later date if we have strong enough reason to do so.

For PHDF5, construction of MPI derived types will be moved to the MPIO VFD, and thus most MPIO specific code will be moved out of the data set code. Note, however, the the MPIO VFD will still have to be told whether the I/O request is independent or collective, and where necessary, will still have to be told to join collective I/O operations with empty requests. As discussed earlier, selection I/O offers the possibility of moving this latter function to the MPIO VFD – but it will probably not prove practical to implement this in the first cut.

To-Do list:

- Add "readsel" and "writesel" selection I/O analogs for readvv/writevv
- Implement version of H5Dmpio.c for selection I/O

  - Link chunk
  - Multi chunk – may be able to move to VFD
  - Filtered link chunk
  - Filtered multi chunk
- Implement selection I/O version of chunk cache, only calls single_read/write analog with all chunk selections
  - Modify chunk cache to evict (and load?) multiple chunks at a time using vector I/O
  - Modify chunk cache to perform direct disk I/O on multiple chunks with a single VFD call
  - Modify H5D__select/scatgath_read/write to support new interface (receiving multiple selections)
  - Implement selection I/O versions of H5D__select_read/write
- Modify parallel compression to use vector I/O

Note that we don't have to do anything to compact datasets – they will be converted to vector I/O in passing when the metadata cache is converted.

2.2 Metadata Cache

On flush, the metadata cache constructs a list of buffers to flush, and then walks that list to flush them. Converting this to vector I/O is trivial. Note, however that for basic functionality, it isn't necessary as the existing I/O calls will still work, albeit more slowly than we would like.

The metadata cache doesn't need vector I/O for reads, as it reads one contiguous section of the file at a time (but note that this may change in the future).

In the parallel case, when collective metadata writes are enabled, the metadata cache distributes the metadata writes across the available ranks, constructs the necessary MPI derived type, and then writes them in a collective operation. This should be easy enough to implement with vector I/O once we have the MPI I/O VFD modified to support selection and vector I/O.

2.3 Page Buffer

When caching pages of metadata, the behavior of the page buffer is essentially the same as that of the metadata cache, and thus conversion to use vector I/O is trivial in this case.

However, the page buffer can also be configured to buffer cache raw data, which raises the following issues:

In addition to regular and vector I/O requests, raw data I/O requests can also be in selection I/O form. From the perspective of the page buffer, four strategies for handling selection I/O requests present themselves.

1. Allow the page buffer to temporarily increase in size as necessary to contain all pages touched by the selection I/O request, and use a vector I/O read request to load pages as necessary. Once all pages are in the buffer, the selection I/O request could be applied to the page buffer, and the page buffer allowed to shrink to its maximum allowed size after the selection I/O

request has been serviced. While this option is conceptually clean, the lack of any upper bound on the size of selection I/O requests probably makes this solution unacceptable from a memory footprint perspective.

2. Flush all dirty pages touched by the selection I/O request, and then pass the request down to the VFD layer. This solution also has the virtue of conceptual cleanliness, but it causes additional I/O, and essentially bypasses the page buffer for selection I/O requests.
3. Determine the set of pages touched by the selection I/O request, and intersect this set with the set of pages currently in the page buffer – call this the resident page set.

   Intersect the resident page set with the selection I/O request to find the resident selection I/O request. Subtract the resident selection I/O request from the original selection I/O request to obtain the non-resident selection I/O request. [8]

   Satisfy the resident selection I/O request from the page buffer, and pass the non-resident selection I/O request to the VFD layer. Note that if we construct the resident and non-resident selection I/O requests correctly, we can setup pointers into the buffer used by the original selection I/O request, and thus avoid the need for any touch up.

   In addition to being complicated, this solution forces us to flatten the selection earlier than we would otherwise do so.
4. Determine how many pages the selection I/O request touches. If this number is no larger than the maximum number of pages in the page buffer, apply solution 1. If it is greater, apply solution 2.

   In addition to being conceptually simple, this solution conforms to the original goal of the page buffer, which was to optimize small data I/O requests.

Fortunately, it is neither necessary nor prudent to pick a solution now.

It is not necessary, as the initial goal is to support sub-filing which implies parallel HDF5. Since the page buffer is disabled in this case, the point is moot.

It is not prudent for the following reasons:

- Implementation of VFD SWMR required a complete re-implementation of the page buffer. As this new implementation of the page buffer is likely to be moved into develop within a year, any work on the existing page buffer will only have to be repeated at that time.
- The problem of intersecting pages in the page buffer with selection I/O requests is quite similar to the problem of intersecting selection I/O requests with sub-files. Since we have to solve the latter problem more or less immediately, best to wait on the former until we can apply the lessons learned.
- If we can restrict the page buffer to only regular and vector I/O requests, the problem become much easier. Since cache coherency concerns make it likely that we will never enable the

[8] We can complicate this solution further by loading pages into the page buffer so as to maximize the size of the resident I/O request. While this would be truer to the concept of the page buffer, the current outline should be sufficient for now.

page buffer for raw data in parallel HDF5, and since no serial VFD under consideration will have need for selection I/O, we could achieve this restriction by moving the page buffer to the VFD layer either as part of the H5FD code proper, or as a pass through VFD. Whether this is a good idea or not, we will not be in a position to attempt it until the question of VFD SWMR is settled.

For these reasons, we will bypass the page buffer for now by simply disabling the page buffer whenever selection I/O is in use.

2.4 VFD Layer

The work needed in the VFD layer is a mix of items needed to support vector & selection I/O in general, and also to support sub-filing – the first major user of selection I/O.

2.4.1 Upper level prior to VFD calls proper

The top of the VFD layer is that portion of the VFD code that supports the public H5FD interface, defines the interface that must be supported by VFDs, and does any necessary translation for VFDs that do not support selection and/or vector I/O. This breaks into the following tasks:

- Specify the internal vector and selection I/O interfaces called by the upper levels of the HDF5 library
- Specify the necessary VFD interface extensions
- Implement the necessary translation facilities needed to support VFDs that do not support selection and/or vector I/O.

2.4.1.1 Internal vector and selection I/O interfaces

2.4.1.1.1 Vector I/O

The signatures of the private H5FD vector I/O calls (shown below) are essentially identical to those of the public versions.

```
herr_t H5FD_read_vector(H5FD_t *file, size_t count, H5FD_mem_t types[],
                        haddr_t addrs[], size_t sizes[],
                        void *bufs[] /*out*/);

herr_t H5FD_write_vector(H5FD_t *file, size_t count, H5FD_mem_t types[],
                         haddr_t addrs[], size_t sizes[],
                         const void *bufs[] /* in */);
```

2.4.1.1.2 Selection I/O Proper

Internally, the HDF5 library represents selections as instances of `H5S_t`, which on the surface suggests that the internal versions of the selection I/O routines should use vectors of pointers to `H5S_t` for the `mem_spaces` and `file_spaces` parameters.

However, this is complicated by the following considerations:

In cases where the underlying VFD supports selection I/O, the vectors of pointers to `H5S_t` must be converted to vectors of `hid_t`[9]. Normally, this is not a problem, as the instances of `H5S_t` will have been internally generated, and we can simply assign `hid_t`'s. However, for I/O on contiguous datasets, the selections provided by the user may be used directly, and thus already have associated `hid_t`'s, which will not be readily available.

In contrast, when the underlying VFD does not support selection I/O, the H5FD selection I/O calls must walk the supplied selection and translate them into vector and/or regular I/O calls. For efficiency in this case, receiving vectors of `H5S_t` will be more convenient.

While several solutions are possible, having two sets of internal H5FD selection I/O calls, one accepting vectors of `hid_t`, and the other vectors of pointers to `H5S_t` seems plausible. This will allow us to pass the hid_t associated with a selection down to be H5FD code when available, and delay assigning `hid_t`'s to instances of `H5S_t` until necessary.

That said, this approach may not be flexible enough for multi-dataset I/O.

With the proviso that we may revisit this issue, the proposed H5FD selection I/O internal API signatures are shown below:

`hid_t` based versions:

```
    herr_t H5FD_read_selection_id(H5FD_t *file, H5FD_mem_t type,
                                  size_t count, hid_t mem_spaces[],
                                  hid_t file_spaces[], haddr_t offsets[],
                                  size_t element_sizes[], void * bufs[] /*out*/);

    herr_t H5FD_write_selection_id(H5FD_t *file, H5FD_mem_t type,
                                   size_t count, hid_t mem_spaces[],
                                   hid_t file_spaces[],haddr_t offsets[],
                                   size_t element_sizes[],
                                   const void * bufs[] /*in*/);
```

`*H5S_t` based versions:

```
    herr_t H5FD_read_selection(H5FD_t *file, H5FD_mem_t type,
                               size_t count, H5S_t *mem_spaces[],
                               H5S_t *file_spaces[], haddr_t offsets[],
                               size_t element_sizes[], void * bufs[] /*out*/);

    herr_t H5FD_write_selection(H5FD_t *file, H5FD_mem_t type,
                                size_t count, h5S_t *mem_spaces[],
                                H5S_t *file_spaces[], haddr_t offsets[],
                                size_t element_sizes[],
                                const void * bufs[] /*in*/);
```

[9] At least for external VFD's. See discussion in "Extensions to the VFD Interface".

2.4.1.2 Extensions to the VFD interface

The proposed vector and selection I/O additions to H5FD_class_t are shown below. Parameters and semantics are identical to the public versions.

```
herr_t (*read_vector)(H5FD_t *file, hid_t dxpl_id, size_t count,
                      H5FD_mem_t types[], haddr_t addrs[], size_t sizes[],
                      void *bufs[] /*out*/);

herr_t (*write_vector)(H5FD_t *file, hid_t dxpl_id, size_t count,
                       H5FD_mem_t types[], haddr_t addrs[], size_t sizes[],
                       const void *bufs[] /* in */);

herr_t (*read_selection)(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                         size_t count, hid_t mem_spaces[],
                         hid_t file_spaces[], haddr_t offsets[],
                         size_t element_sizes[], void * bufs[] /*out*/);

herr_t (*write_selection)(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id,
                          size_t count, hid_t mem_spaces[],
                          hid_t file_spaces[],haddr_t offsets[],
                          size_t element_sizes[], const void * bufs[] /*in*/);
```

For efficiency, internal VFDs will likely examine selections directly, and thus assigning `hid_t`'s for the call, and then dereferencing them again to obtain pointers to the `H5S_t`'s seems an unnecessary epicycle. Unfortunately, the alternatives to this are either a private VFD interface that allows us to pass the `mem_spaces` and `file_spaces` parameters as vectors of pointer to `H5S_t`, or fun and games with C type coercion – neither of which is attractive. Thus we will do neither unless the cost of the epicycle proves significant.

2.4.1.3 Translation Facilities

Since VFDs will not support vector I/O and/or selection I/O unless there is a performance advantage in doing so, the internal versions of the vector and selection I/O calls first test to see if the underlying VFD supports vector or selection I/O as appropriate.

If it does, the vector or selection I/O call is simply passed down to the underlying VFD.

If it doesn't, the vector I/O call walks the vector of <type>, <addr>, <size>, and <buffer> quadruples, and generates the sequence read or write calls necessary to execute the vector I/O call. Observe that there is no plan to support conversion of vector I/O calls to selection I/O proper. This is driven by the observations that:

1. It is not in general possible[10], and
2. Given the relative ease of implementation, it seems unlikely that a VFD would support selection I/O but not vector I/O.

[10] Vector I/O can have a different value of `H5FD_mem_t` for each offset, address, buffer triplet in the vector, where as selection I/O requires the same `H5FD_mem_t` for the entire I/O request

Similarly, if the underlying VFD doesn't support selection I/O, the selection I/O calls must walk the selection, and convert it into one or more vector I/O calls if the underlying VFD supports vector I/O, or a sequence of regular read / write calls if it doesn't. In the event of conversion to one or more vector I/O calls, the decision between single and multiple vector I/O calls will be driven by a (user configurable?) limit on vector lengths in selection to vector I/O translations[11].

While the algorithm for walking vector I/O requests and turning them into sequences of regular I/O calls is obvious, that for walking selections and converting them into vector or regular I/O calls isn't.

Fortunately, this is largely a matter of re-locating and re-working code that already exists in the HDF5 library (see `H5S_mpio_space_type()`). For efficiency, the existing code walks the selection data structures directly. While we should do the same for efficiency, if we want to use this code in the pass through VOL that we have been using to develop the sub-filing VFD, we will have to use the API extensions for efficient traversal of selections discussed below instead.

2.4.2 API extensions needed for efficient traversal of selections by VFDs

While HDF5 currently provides routines to examine hyperslab selections, none of them run efficiently for non-regular hyperslabs. Here we propose extensions to the HDF5 API to allow traversal of hyperslab selections with performance similar to that of HDF5's internal "span tree" implementation.

Since this API is intended for use by VFD/VOL developers we want to translate the multi-dimensional dataspace selection information into linear offset/length information, while retaining the ability to efficiently describe patterns. Instead of describing runs in a single dimension with a link to runs in faster changing dimensions (as in span trees), we do this by describing patterns in a linear space of regular "sub patterns", which are regularly repeated patterns of selected bytes. The pattern is defined using the familiar start/stride/count/block parameters, though again flattened to 1-dimension. This start/stride/count/block set defines a pattern of sub-patterns, instead of dataset elements. The size (in bytes) of the sub-pattern is given, as is an iterator that may be used to recursively traverse the sub-pattern using the same method. If no iterator is given, then the entire sub-pattern is selected. Offsets calculated using this information should be added to the corresponding element of the `offsets` array passed through the `read/write_selection` callback to obtain file addresses.

The proposed API extensions are:

```
typedef struct {
    hsize_t start;
    hsize_t stride;
    hsize_t count;
    hsize_t block;
    hsize_t sub_pattern_size;
    hid_t sub_pattern_iter;
```

[11] If a selection I/O request must be broken into multiple vector I/O calls, and the request is marked as collective, the underlying VFD must be informed of this so it can coordinate with its counterparts, and issue empty collective I/O requests as necessary to avoid deadlocks. This will be a moot point to begin with, as all VFDs that support MPIO (the modified MPIO VFD, and the Argonne topology aware VFD) will have to support selection I/O for performance reasons. However, we don't want to lose track of this point as it may become relevant in the future.

```
    void *udata;
} H5S_sel_pattern_segment_t;

herr_t
H5Ssel_iter_get_pattern_segment(hid_t sel_iter_id, hbool_t *complete,
    H5S_sel_pattern_segment_t *pattern_segment);

herr_t
H5Ssel_iter_pattern_cache_udata(hid_t sel_iter_id, void *udata);
```

`H5Ssel_iter_get_pattern_segment` is the primary routine used to traverse the selection. It returns a pattern segment defined by the `H5S_sel_pattern_segment_t struct pattern_segment`.

`sel_iter_id` is a selection iterator that was either created with `H5Ssel_iter_create` or returned as `sub_pattern_iter` by this routine. When using `H5Ssel_iter_create` to create an iterator for this, you should pass the corresponding element of the `element_sizes` array passed through the `read/write_selection` callback as the `elmt_size` parameter for `H5Ssel_iter_create`. It is illegal to mix calls to `H5Ssel_iter_get_pattern_segment` and `H5Ssel_iter_pattern_cache_udata` with other calls using the same iterator. If the output parameter `complete` is set to `FALSE`, then this indicates there are more pattern segments to process with this iterator.

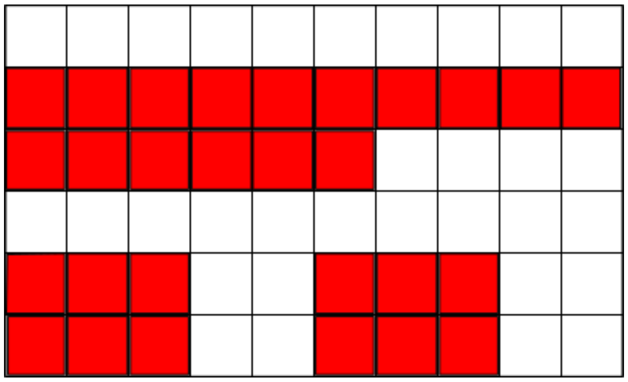
`sub_pattern_iter` is a selection iterator created by `H5Ssel_iter_get_pattern_segment` which may be used to recursively examine the sub pattern of selected bytes, by passing it as the `sel_iter_id` parameter to a recursive call to `H5Ssel_iter_get_pattern_segment`. All iterators returned as `sub_pattern_iter` must eventually be closed using `H5Ssel_iter_close`.

`H5Ssel_iter_pattern_cache_udata` can be used by the caller to store any data with the most recently returned pattern segment. If the same pattern segment is returned by a later call to `H5Ssel_iter_get_pattern_segment`, the `udata` field will be the same as that passed to `H5Ssel_iter_pattern_cache_udata`. By default the `udata` field will be set to `NULL`. Passing the dataspace ID to intervening HDF5 calls during traversal may interrupt this and cause `udata` to be reset to NULL.

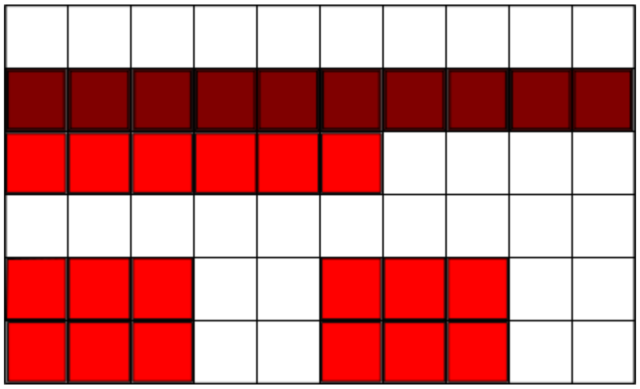
Initially, in order to closely match the internal span tree data structure, when operating on non-regular hyperslabs, `H5Ssel_iter_get_pattern_segment` will only return single blocks of patterns, that is, stride and count will be set to 1. For regular hyperslabs it will return the full start/stride/count/block, with one pattern block at each level of recursion. `H5Ssel_iter_get_pattern_segment` will be implemented for all selection types, though it will not offer any performance advantage for selection types other than hyperslab.

2.4.2.1 Example

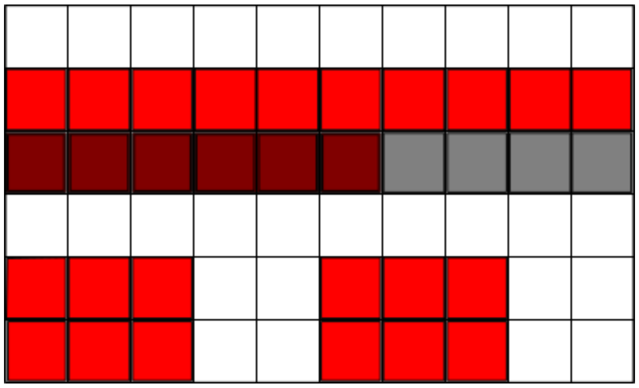
This example walks through the use of the above facilities to traverse an irregular hyperslab selection on a 2 dimensional dataspace. The illustrations will show the portions of the selection described by calls to `H5Ssel_iter_get_pattern_segment`, but keep in mind that data from `H5Ssel_iter_get_pattern_segment` contains no dimensionality information, only repeating patterns in a 1-dimensional byte array. Consider the following selection within a 6x10 dataspace, with 4 byte elements:



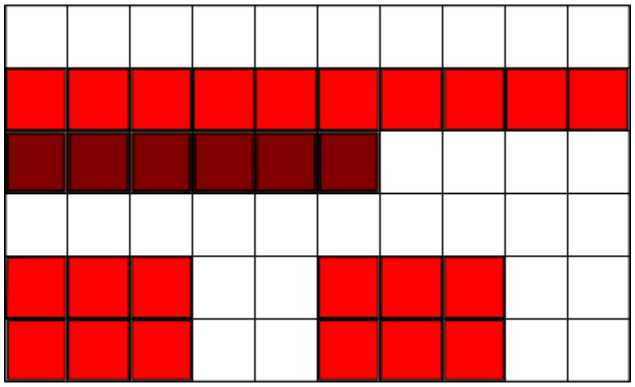
The first call to `H5Ssel_iter_get_pattern_segment` will return the following for `pattern_segment`, corresponding to the first selected row: `{1, 1, 1, 1, 40, H5I_INVALID_HID, NULL}`. This denotes a contiguous block of selected bytes at offset 1 (start) x 40 (sub pattern size) and with length 1 (block) x 40 (sub pattern size):



There is no sub-pattern, so the app will simply call `H5Ssel_iter_get_pattern_segment` again with the same iterator, and the second call will return `{2, 1, 1, 1, 40, <valid ID>, NULL}`. This indicates a non-contiguous sub pattern starting at offset 80 and occurring only once:

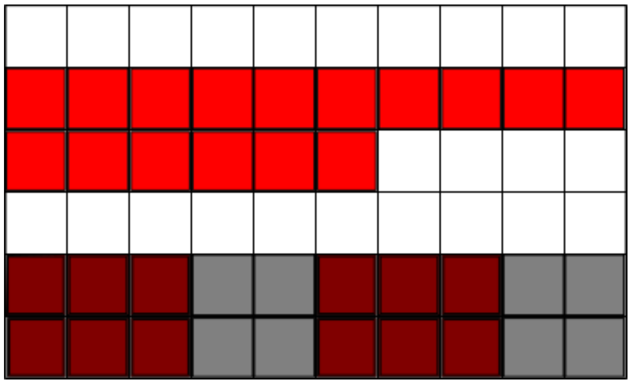


The app will then call `H5Ssel_iter_get_pattern_segment` with the returned sub pattern iterator ID, which will return the following sub pattern: `{0, 1, 1, 6, 4, H5I_INVALID_HID, NULL}`. This indicates that the selected sub pattern is a contiguous block of 6 4-byte elements beginning at the start (0 offset) of the sub pattern. This information can be combined with that returned at the higher level of recursion to indicate a block that begins at offset 80 with a length 24:

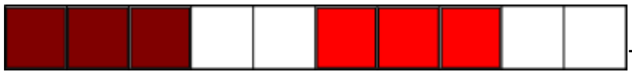


The function will return `TRUE` for the `complete` parameter, so the app will then close the sub pattern iterator.

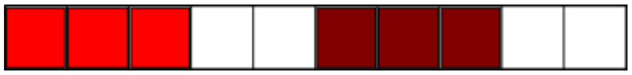
Next, the app will again call `H5Ssel_iter_get_pattern_segment` with the top level iterator, which will return `{4, 1, 1, 2, 40, <valid ID>, NULL}`. This indicates a non-contiguous sub pattern starting at offset 160 and occurring twice (encompassing two rows in the 2-dimensional representation):



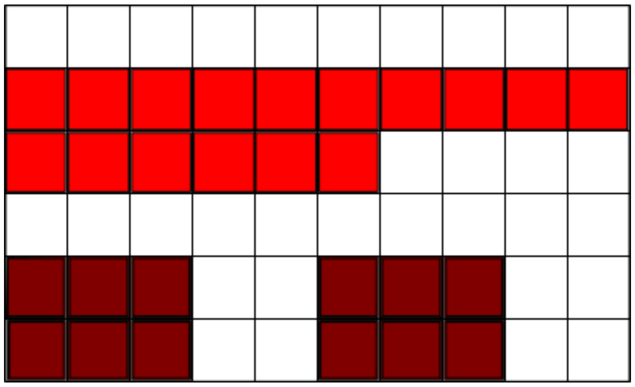
A call with the sub pattern iterator ID will then return `{0, 1, 1, 3, 4, H5I_INVALID_HID, NULL}`. This indicates a contiguous block of 3 4-byte elements beginning at offset 0 of the sub pattern. Here we show only the sub-pattern:



The `complete` parameter will be `FALSE`, so the next call with the sub pattern iterator will return `{5, 1, 1, 3, 4, H5I_INVALID_HID, <possibly a valid pointer>}`. This indicates a contiguous block of 3 4-byte elements beginning at offset 20 of the sub pattern:



If a call to `H5Ssel_iter_pattern_cache_udata` was made after the previous call to `H5Ssel_iter_get_pattern_segment` the `udata` field will be the same as that passed to `H5Ssel_iter_pattern_cache_udata`. The app then has all information about the sub pattern, and from the higher level of recursion it knows that it repeats twice starting at offset 160:



It would be possible for the second recursive call to `H5Ssel_iter_get_pattern_segment` to instead return `{0, 5, 2, 3, 4, H5I_INVALID_HID, NULL}` and then be complete. Initially, however, this will not happen as the implementation will closely follow the internal span tree data structure, which does not have this information.

2.4.3 Support for intersections of selections with sub-files

While it is intended to facilitate a variety of improvements to the HDF5 library, the initial reason for implementing selection I/O is to permit the implementation of sub-filing. Thus some discussion of features needed specifically for sub-filing is appropriate.

When the sub-filing VFD receives either a vector or a selection I/O request, it must parcel that request out across the applicable sub-files. This may be thought of as involving the following operations:

- Determining which sub-files are touched by the I/O request.
- For each sub-file that is touched, construct a vector or selection (as appropriate) describing the portion of the I/O request that effects that sub-file (and for selection I/O, construct a selection describing the matching portion of the memory buffer).

As each sub-file can be described as a selection on the HDF5 file, and as we have existing code to perform intersections on selections and project those intersections onto the matching memory selections, we already have code to perform the above functions – with the caveat the selections describing the sub-files will be selections on a vector of bytes, and thus selections received by the sub-filing VFD may have to be flattened before these intersections can be performed. The features described in the previous section should allow efficient construction of this flattened selection, though we may implement an additional HDF5 routine to accomplish this.

While it seems that we already have much of the necessary functionality, efficiency is a concern, as there may be hundreds of sub-files in large computations. Thus, we need to at least consider the following items:

1. Efficient method of determining which sub-files are touched by a selection I/O request
2. Were a sub-file is touched by a selection I/O request, efficient construction of a selection I/O request describing I/O on the target sub-file

Of the above, only 1) is of immediate interest, as the selections used in sub-filing I/O requests are expected to be relatively simple in all use cases of immediate interest. This can be efficiently accomplished in most cases by calculating the bounds of each selection, converting those bounds to file offsets, and checking these blocks for intersection with the sub-files.

For 2) the proposed improvements to the existing `H5Sselect_project_intersection()` function should improve sub-file selection construction efficiency. Alternatively, we should observe that it is not strictly necessary to describe the potion of a selection I/O request directed at a given sub-file in selection I/O format – vector I/O will work, and may be preferable in some cases.

A further point is whether we will assume that the sub-filing VFD is built into the HDF5 library, and is thus eligible to use HDF5 internal API's for the above purposes, or if we will need to extend the public API for this purpose. Current thinking is the former, but we should keep in mind the costs of changing our minds on this point.

2.4.4 Support for efficient serialization / deserialization of selections (needed for sub-filing, and for a more efficient implementation of mirror VFD).

Once the sub-filing VFD divides a selection I/O request into sub-requests for each sub-file touched by the overall selection I/O request, it must somehow transmit these requests to the I/O concentrators responsible for the sub-files involved.

Again, we have existing mechanisms to serialize and de-serialize selections that should provide the necessary functionality, and again there are performance concerns about this code. As before, the expected relative simplicity of selection I/O in the sub-filing use cases of immediate interest suggests that this may not be an issue for now. That said, it is something we should think on.

One possible alternate solution is to use vector I/O for communications with the I/O concentrators. This has the twin advantages of simplicity, and relieving the I/O concentrator of the overhead of walking the selection. On the down side, it has the potential of increasing message size by more than an order of magnitude in the worst possible case.[12]

2.4.5 Modification of the MPIO VFD to support vector and selection I/O

While sub-filing is the initial use case for selection I/O, we need a version of the MPIO VFD that supports vector and selection I/O before we can demonstrate equivalent performance and remove the old code that constructs MPI derived types in the data set code.

As with modifying the VFD layer to perform translations of selection I/O requests to regular or vector I/O as needed, this is largely a matter of re-factoring existing code and relocating it to the new MPIO VFD. Since this VFD will be built in to the HDF5 library, there is no reason for it not to walk the selection data structures directly.

That said, as time permits, it would be useful to have a reference version of this VFD that uses the API extensions to support efficient traversal of selections discussed above. In particular, the topology aware VFD folks at Argonne will need something like this as an example when they upgrade their VFD to support selection I/O.

[12] The worst case is single byte read or write requests, yielding an extra 16 bytes of offset and length per request, and perhaps an additional byte of memory type. While possible, this seems an improbable workload on HPC machines.

2.5 Miscellaneous Changes

This section is a catchall for changes required to adjust to the movement of most MPIO specific code to the VFD layer in general, and the MPIO VFD in particular. Ideally, we will note all such issues before we encounter them – however, it is more likely that this section will be a list of issues encountered along with a description of how we addressed them.

2.5.1 Exposure of MPI Communicator, Rank, and Size to the Upper Levels of the HDF5 Library

While this section addresses changes in the VFD layer, these changes are orthogonal to selection I/O, and thus we classify them as miscellaneous changes.

At present, any VFD that supports MPI must support three additional private callbacks (defined as fields of `H5FD_mpi_class_t` in H5FDmpi.h):

`int (*get_rank)(const H5FD_t *file),`

`int (*get_size)(const H5FD_t *file),` and

`MPI_Comm (*get_comm)(const H5FD_t *file).`

VFDs that support MPIO are required to obtain the communicator from the FAPL, obtain the MPI rank and size, and make all this information available to the upper levels of the HDF5 library via the

`H5FD_mpi_get_rank(),`

`H5FD_mpi_get_size(),` and

`H5FD_mpi_get_comm()`

routines (also declared in H5FDmpi.c), which simply call the associate callbacks supported by the VFD.

While we could retain this architecture, it has the following deficits:

1) VFDs that support MPI must support an extended set of callbacks.
2) The `get_rank()`, `get_size()`, and `get_comm()` callbacks must be added to the public interface for VFDs.

   Further, since `get_comm` returns `MPI_Comm`, it can't appear in the serial build – which implies different VFD interfaces for serial vs. parallel.
3) It requires duplicate code in each VFD that supports MPI to obtain the communicator from the FAPL, obtain the rank and size, and report this information on request.

While it would be nice to delete the MPI specific VFD callbacks, and move management of exposure of the MPI communicator, rank, and size to the H5FD code, doing so would create problems down the road.

To see this, consider that we would have to store the communicator, rank, and size somewhere. The H5FD_t structure is the obvious location, but this will cause problems with dynamically loaded VFDs, as it makes the size of the `H5FD_t` structure variable between serial and parallel builds.[13] While we could code around the problem with unions and reserved space, that is asking for portability problems.

Alternatively, we could store the communicator, rank, and size in another data structure. While that works if all VFDs in the stack either don't use this data, or all use the same communicator, rank, and size, it is easy to construct sub-filing VFD use cases where this is not the case.

Thus we are pushed back to querying the VFD whenever the MPI communicator, rank, or size is needed. However, we still need to get rid of the MPI specific VFD callbacks so that the `H5FD_class_t` does not change depending on serial vs. parallel build.

We propose to square this circle by the addition of a generic callback that allow VFDs to support VFD specific operations. This notion was proposed in the onion VFD RFC. The version here has been modified slightly to offer more graceful stacking of VFDs. It will probably be refined further as part of the ongoing efforts to support pluggable VFDs, and to redesign the `H5FD_class_t` structure. The proposed signatures for the external, internal, and VFD callback versions are as follows:

```
herr_t H5FDctl(H5FD_t *file, uint64_t op_code, uint64_t flags,
               const void * input, void ** result);

herr_t H5FD_ctl(H5FD_t *file, uint64_t op_code, uint64_t flags,
                const void * input, void ** result);

herr_t (*ctl)(H5FD_t *file, uint64_t op_code, uint64_t flags,
              const void * input, void ** result);
```

where:

- `file` is a pointer to the `H5FD_t` for the target VFD instance.
- `op_code` is an integer code specifying the desired operation. For our immediate purposes we need 3:
  - `H5FD_CTL__GET_MPI_COMMUNICATOR`
  - `H5FD_CTL__GET_MPI_RANK`
  - `H5FD_CTL__GET_MPI_SIZE`
- `flags` specify handling of the `H5FDctl` call if it is not handled by the target VFD. The list of defined flags will likely expand, but the obvious ones are:
  - `H5FD_CTL__FAIL_IF_UNKNOWN_FLAG` – Absent modifying flag, fail if the opcode is unknown / unsupported.
  - `H5FD_CTL__IGNORE_IF_UNKNOWN_FLAG` – Absent modifying flag, ignore the call and return success if the opcode is unknown / unsupported.
  - `H5FD_CTL__ROUTE_TO_TERMINAL_VFD_FLAG` – pass through VFDs for which the op_code is unknown / unsupported should relay the H5FD_ctl() call towards the terminal VFD.

[13] Hat tip to Quincey for pointing this out.

For our immediate purposes, we only need the first and third flags. The second one will be needed for the limited asynchronous I/O support discussed below.

- `input` points to any data supplied by the caller (typically, an `op_code` specific structure).
- `result` points to a buffer supplied by in caller in which any required data is returned. This buffer must be both allocated and freed by the caller, and must be the expected size.

The general semantics of the H5FDctl() call is as follows:

- If the `op_code` is known, process it as appropriate.
- If the `op_code` is unknown, proceed as indicated by the flags – specifically:
    - If the VFD is terminal[14], and `H5FD_CTL__FAIL_IF_UNKNOWN_FLAG` is set, return `FAIL`.
    - If the VFD is terminal, and `H5FD_CTL__IGNORE_IF_UNKNOWN_FLAG` is set, return SUCCEED.
    - If the VFD is pass through, and `H5FD_CTL__ROUTE_TO_TERMINAL_VFD_FLAG` is set, pass the call to either the terminal VFD, or to the next pass through VFD on the path to the terminal VFD.
    - If the VFD is pass through, and no routing flags are set, proceed as for terminal VFDs.

The semantics for the three op_codes we have defined are what one would expect – return the MPI communicator, rank, or size in `**result`. The caller must allocate the appropriate buffer. Note that the communicator is not duplicated – the caller must do so if desired.

We will likely want to reserve ranges of op_codes and flags both for THG use, and for experimental use. A registry external op_codes and flags may also be useful. However, these are topics best addressed in the context of pluggable VFDs, and are left until after we have made any necessary adjustments both for it, and the other VFDs currently in progress.

Observe that the above solution makes MPI support transparent to pass through VFDs, and resolves the current problem of an MPI specific version of `H5FD_class_t`. It does require duplicate code in all VFDs that support MPI to make the MPI communicator, rank, and size available, but on balance, this seems an acceptable trade off relative to the cost of moving this functionality to the H5FD code in the parallel build.

[14] In essence, a terminal VFD is a VFD that manages I/O for an entire HDF5 file. Thus, the sec2 VFD is terminal, unless it is being used by the family or multi file driver to manage I/O for some portion of the logical HDF5 file. Similarly, the multi file driver, the family file driver, the VFD SWMR reader VFD, and the sub-filing VFD are all terminal as they are all the lowest point in the VFD stack through which all I/O requests pass. In contrast, VFDs that simply pass I/O requests down the VFD stack (i.e. logging and splitter VFDs) are pass through VFDs and not terminal since while while all I/O requests may pass through them, at least one underlying VFD receives the same stream of I/O requests.
While this definition works at present, it is easy to come up with scenarios where it might have to be twisted a bit – for example, consider the implications of a parallel version of a true network VFD.

2.5.2 Sub-Filing VFD Needs Minimal AIO Support

Review of Richard's sub-filing VFD code has made it clear that if we want a performant sub-filing VFD that uses stacked VFDs to handle sub-file I/O, we must add some minimal AIO support to the VFD interface.

To see this, consider a case in which the I/O generated by a single API call touches ten sub-files, and is delivered to the sub-filing VFD in a single selection I/O call. The sub-filing VFD must break this selection I/O call into ten sub calls (one per affected sub-file), and then relay the sub-calls down to the VFDs responsible for the target sub-files.

In the absence of AIO support, these ten sub-requests must be serialized. While this is probably acceptable on a local file system, it defeats the purpose of sub-filing on large HPC machines.

The current sub-filing VFD gets around this problem by merging the sub-filing VFD and the I/O concentrator VFDs. While this works, and we will use this solution for the initial merge of sub-filing into the selection I/O branch, we must break the two apart into stackable VFDs to gain the desired flexibility.

For purposes of sub-filing proper, the AIO extension needed are very rudimentary – the equivalent of the Concurrent Pascal cobegin / coend construct would do the job nicely, and avoid separate code paths for underlying VFDs that support AIO vs. those that don't.

Observe that the H5FD_ctl() call discussed above is a good fit for this – if we define the following op_codes:

- `H5FD_CTL__COBEGIN` – All I/O calls issued on the current thread after the COBEGIN and before the COEND may run asynchronously and concurrently.
- `H5FD_CTL__`COEND – All I/O calls issued on the current thread before the COEND must complete before the COEND call returns. The call returns `SUCCEED` if no errors are detected, and `FAIL` otherwise.

If we bracket per sub-file I/O requests with the COBEGIN and COEND H5FD_ctl() calls to each of the relevant VFDs, and combine these calls with:

`H5FD_CTL__IGNORE_IF_UNKNOWN_FLAG` and

`H5FD_CTL__ROUTE_TO_TERMINAL_VFD_FLAG`

we can get the desired asynchrony when supported, and still use VFDs that don't support asynchrony without code changes.

Whether we implement something this rudimentary, or opt for more general AIO support depends on what other plausible use cases we can think of, and available resources. However, since the selection I/O extension to the sub-filing VFD will be easier to debug in serial, it would be useful to have this extension in place when we get to that point.

2.5.3 Miscellaneous Notes

Issues to be investigated and dealt with as appropriate:

From Jordan: Here are my findings on HDF5's usage of MPI derived types, based on a quick search for MPI_Type_free. Note that in some of these cases, the derived types are composed of other derived types. I don't know if this would affect the selection I/O work or not, but some of the types can get fairly complicated.

- H5C__collective_write
  - Derived types created for mem and file type
  - Single I/O seems to be sent down via H5F_block_write
- H5D__chunk_collective_fill
  - Derived types created for mem and file type for all chunks at once
  - Single I/O sent down via H5F_shared_block_write
- H5D__link_chunk_collective_io
  - Derived types created for mem and file type for each chunk, then coalesced into single struct derived type
  - Single I/O sent down via H5D__final_collective_io
- H5D__link_chunk_filtered_collective_io
  - Derived types created for mem and file type for all chunks at once
  - Single I/O sent down via H5D__final_collective_io
- H5D__multi_chunk_filtered_collective_io
  - Derived types created for mem and file type for a "round" of chunks (at most 1 chunk per rank)
  - Multiple I/Os sent down via H5D__final_collective_io, 1 per "round" of chunks
- H5D__inter_collective_io
  - Might use a derived type for mem and file types, depending if file_space/mem_space are passed in
  - May be called multiple times during H5D__multi_chunk_collective_io to write out several "rounds" of chunks, as in the filter case above
- H5FD__mpio_write
  - Might use a derived type if a large (> 2GB) I/O is needed for MPI I/O call
  - Single I/O sent down via MPI_File_write_at/MPI_File_write_at_all
- H5_mpio_create_large_type
  - Used in multiple places to create a derived type for >2GB I/Os
  - Most or all of these places appear to be covered by the previous bullets

3 Recommendation

Review this RFC to verify that no significant issues have been missed, and that there are no obvious show stoppers.

Assuming no red flags, proceed to implementation, with updates to this RFC as technical details are worked out and tested.

Acknowledgements

TBD

Revision History

February 19, 2021:	Version 1 circulated for comment.
March 4, 2021:	Version 2 circulated for comment. Major edits to introduction and section 2.3.2 to address comments, lesser edits elsewhere.
March 23, 2021:	Version 3 circulated for comment. Moved discussion of page buffer modifications into its own section, and rewrote to address incoming selection I/O calls.
May 11, 2021:	Version 4 circulated for comment. Modified `types[]` and `sizes[]` optimization in vector I/O calls to require invalid value at index 1 only. Added similar optimization for the `bufs[]` parameter in selection I/O calls. Added discussion of management of VFDs that support MPI, and exposure of communicator, rank, and size to upper levels of the library (section 2.5.1).
May 22, 2021:	Version 5 circulated for comment. Added discussion of need for some sort of AIO support in the VFD interface for sub-filing (section 2.5.2 – move this discussion to the sub-filing RFC?). Removed the `dxpl_id` parameter from the internal versions of the vector and selection I/O calls.
June 10, 2021:	Version 6 circulated for comment. Re-worked discussion of management of VFDs that support MPI, and exposure of communicator, rank, and size to upper levels of the library (section 2.5.1) to address interaction with pluggable VFDs. Reworked discussion of limited AIO support required for sub-filing (section 2.5.2) to use the proposed `H5FDctl()` call.