RFC: New HDF5 API Routines for HPC Applications Read/Write Multiple Datasets in an HDF5 file

Peter Cao
Quincey Koziol
Jonathan Kim
Neil Fortner

The HDF5 library allows a data access operation to access one dataset at a time, whether access is collective or independent. However, accessing multiple datasets will require the user to issue an I/O call for each dataset. This RFC proposes new routines to allow users to access multiple datasets with a single I/O call.

This RFC describes the new API routines, *H5Dread_multi()* and *H5Dwrite_multi()*, which perform a single access operation to multiple datasets in the file. The new routines can improve performance, especially when data accessed across several datasets from all processes can be aggregated in the HDF5 or MPI-I/O library.

This RFC was initially released in 2012, but has now been updated in 2022 to reflect a renewed effort to finish implementing this feature.

1 Introduction

Parallel HDF5 (PHDF5) supports both independent and collective dataset access. When collective I/O is used, all processes that have opened the dataset may do collective data access by calling *H5Dread()* or *H5Dwrite()* on the dataset with the transfer property set for collective access. Accessing datasets collectively using the MPIO VFD can improve I/O performance [1] since MPI can aggregate data into large contiguous accesses to disk instead of small non-contiguous ones.

However, the current HDF5 library does not support a single I/O call for accessing multiple datasets. For example, if one accesses five datasets in a file, one will need at least five I/O calls for each dataset.

We propose to add two new functions to the HDF5 library: *H5Dread_multi()* and *H5Dwrite_multi()*. With the proposed new read/write functions, users will make a single function call to read or write data to multiple datasets in an HDF5 file. Note that the multiple datasets are located in the same HDF5 file for the scope of this task. The new functions can be used for both independent and collective I/O access, but this task's primary purpose is to utilize the collective I/O case.

1.1 Purpose

The purpose of the work is to implement two new functions to the HDF5 library: *H5Dread_multi()* and *H5Dwrite_multi()*, which should give better I/O performance when collective I/O is used.

1.2 Scope

H5Dread_multi() will take information from multiple datasets and read data from a file for the datasets requested. If collective I/O is used, a single I/O call is used for better performance. Overlapping data selections are supported.

H5Dwrite_multi() will take information from multiple datasets and write data to a file for the datasets requested. If collective I/O is used, a single I/O call is used for better performance. If data selections from multiple ranks overlap, the behavior of the H5Dwrite_multi() is not defined. Therefore, overlapping data selections should not be used.

Datasets requested in H5Dread_multi() and H5Dwrite_multi() must reside in the same file. The new functions do not support datasets cross files.

Once the feature is fully productized, the benchmarks mentioned in this report will be rerun, along with other standard I/O kernels, to fully determine the performance of the feature. The final deliverables will also include a reference manual entry for each function and regression tests.

2 Use Case

This section presents two use cases: the FLASH I/O and E3SM. The primary purpose of the use cases is to show how the new library functions can be used to improve I/O.

2.1 Improving FLASH I/O for an ANL project

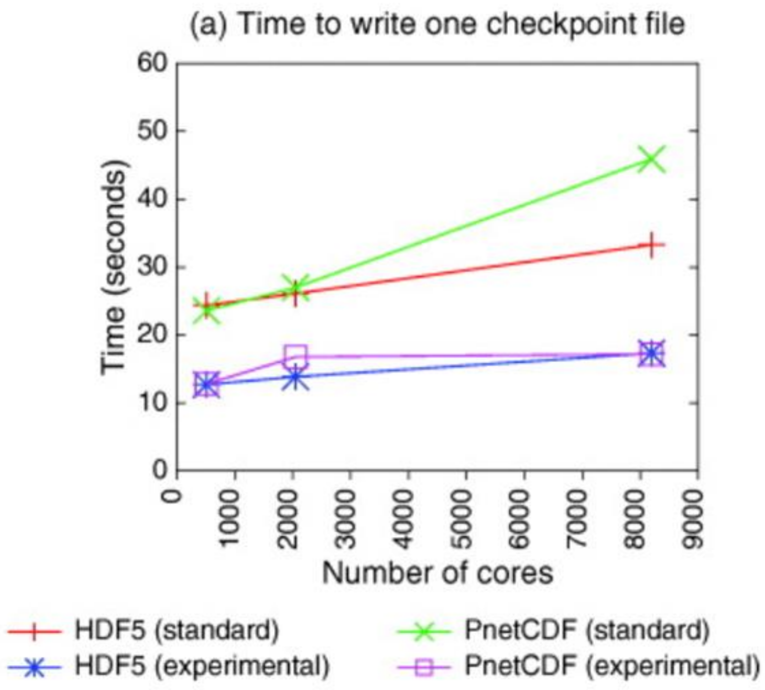
FLASH code was designed to simulate thermonuclear flashes on a Cartesian, structured mesh. The mesh consists of cells that contain physical quantities such as density, pressure and temperature (also known as mesh variables). Each cell is assigned to a self-contained block. In the FLASH file layout, a block is stored in an HDF5 file, and mesh variables are stored as 4D datasets in the file.

The time spent on file I/O in a FLASH simulation is a common bottleneck. Using collective I/O[1] improves I/O performance for HPC applications like FLASH. Current parallel HDF5 performs collective I/O on a single dataset and requires many I/O calls in FLASH simulations since many variables are frequently accessed during each time step. Using the proposed collective I/O on multiple datasets will reduce the number of I/O calls. In an experimental study, Rob Latham, Chris Daley, etc.[2] have shown that the average time for writing a file is reduced by half when collective I/O on multiple variables is used:

> "*The standard file layout approach (storing application data in multiple library objects), however, offers a slight performance trade-off. Each function call represents a relatively expensive I/O operation. All other factors aside, if the goal is to achieve the highest I/O performance a better approach would describe the entire application I/O pattern and then execute a single call. If the application places all mesh variables into a single I/O library object, as in the experimental file layout approach, then a single I/O library call could be issued to service all application variables*

instead of N separate calls. Experiments confirm that this approach does improve performance."[2]

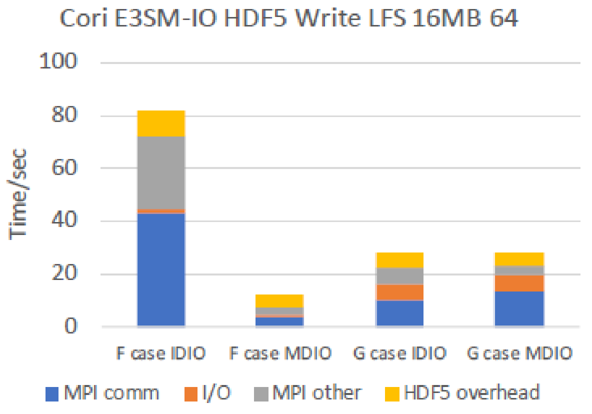
The following figure shows the I/O results for standard file layout (storing mesh variables in multiple datasets) and the experimental file layout approach (placing all mesh variables into a single I/O library object). The results generally show that the single I/O approach (the experimental file layout) reduces the time to write checkpoint files by half.

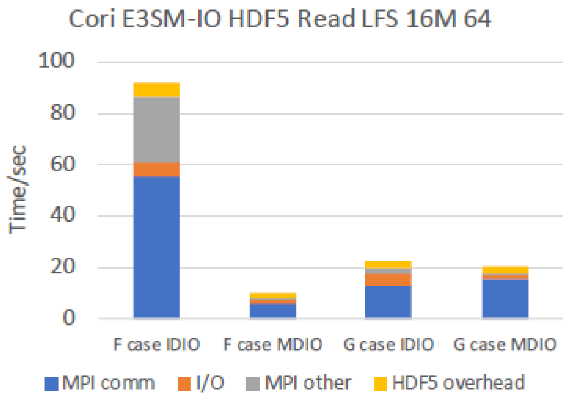


2.2 Investigating E3SM performance improvements for the Exascale Computing Project

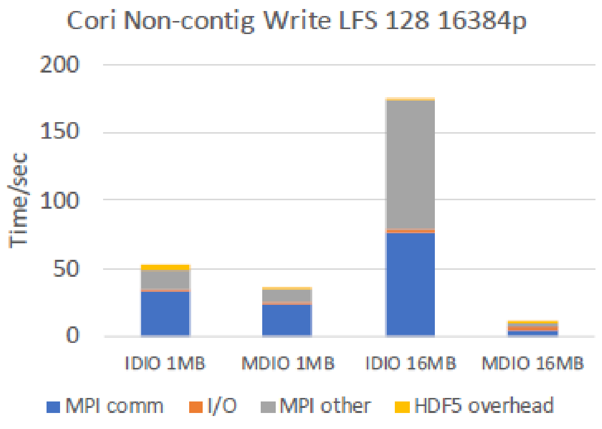
The Energy Exascale Earth System Model (E3SM) is a large-scale Earth modeling code that couples ocean, atmosphere, and ice models. As part of the Exascale Computing Project, the performance of the existing multi dataset I/O prototype in HDF5 was evaluated for two different cases in E3SM, as well as in a synthetic I/O benchmark[3]. We briefly summarize these cases here; for full results, see the reference.

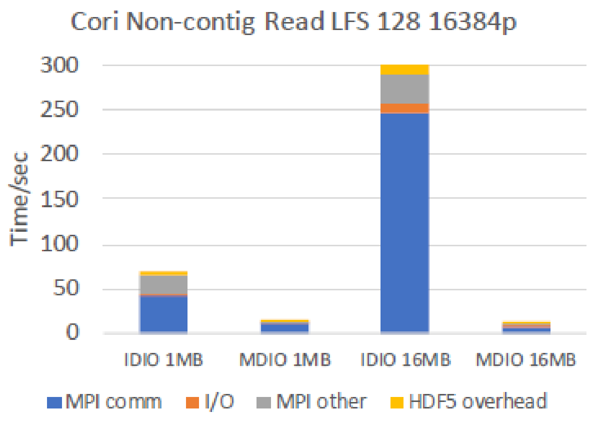
Two cases were evaluated for the E3SM benchmark: the "F" and "G" cases. The F case involved large numbers of datasets, contiguous smaller amounts of data per dataset. The G case involved smaller numbers of datasets with large amounts of data per dataset. This means the F case is more likely to benefit from multi dataset I/O since more I/O operations can be combined, and it is not as limited by raw bandwidth. We indeed see a significant performance improvement of approximately 10x in the F case with only a minor improvement in the G case:



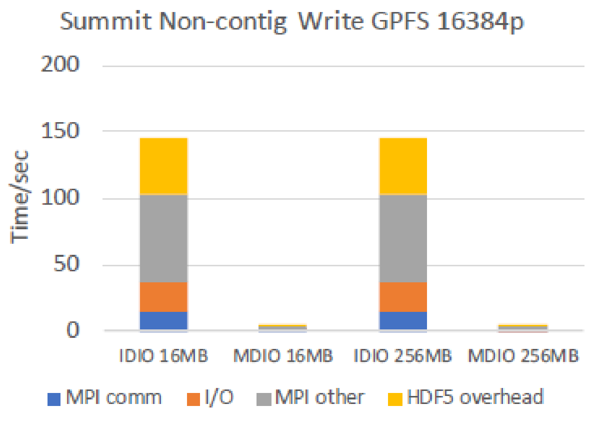


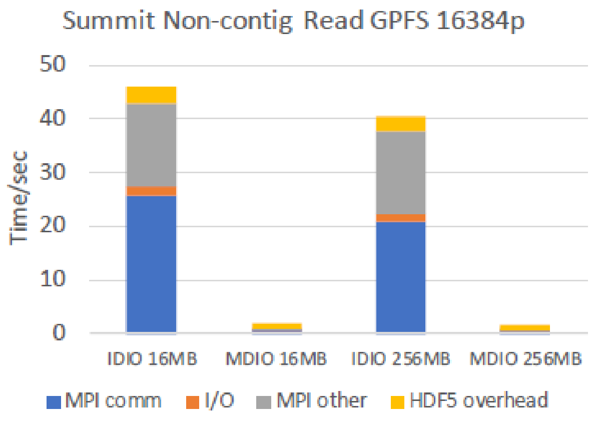
The synthetic benchmark pushes this further by increasing the number of non-contiguous blocks in the datasets, while having as many datasets as the E3SM F case and a total amount of data in between the E3SM cases. This, similarly to the E3SM F case, shows substantial performance improvement from multi dataset I/O:





While a different system using GPFS showed even greater improvements:





3 Functional requirements

The two main purposes of the H5Dread/write_multi() functions are: better I/O performance and convenient function calls for multiple datasets. In addition, the two new functions should meet the following specific requirements other than the standard HDF5 API function requirements.

3.1 H5Dread_multi()

This function will read data from a file to memory buffers for multiple datasets. This function should attain no less I/O performance than reading data from individual datasets.

3.2 H5Dwrite_multi()

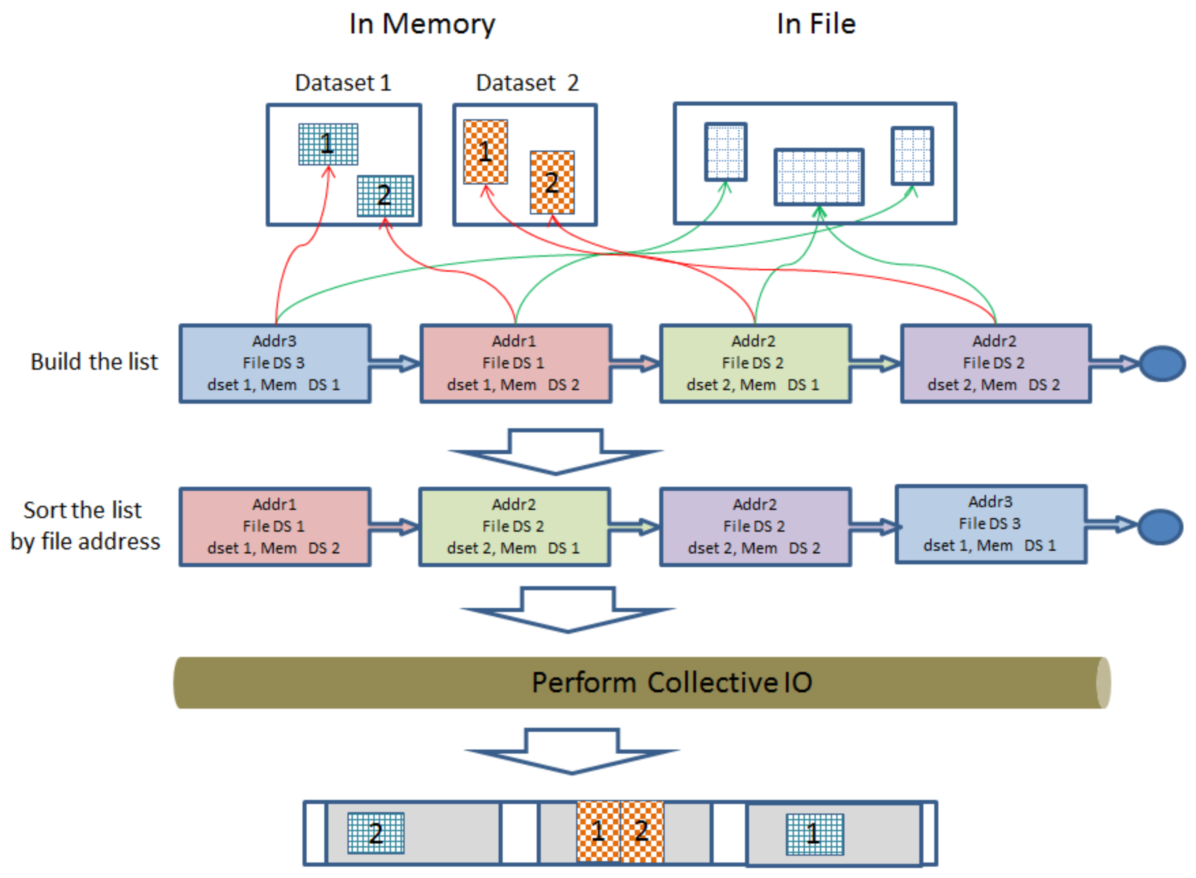
This function will write data in memory to a file for multiple datasets. The selection of data regions to be written cannot overlap. This function should attain no less I/O performance than writing data for individual datasets.

4 Implementation design

The basic approach for multi-dataset collective I/O is similar to the POSIX lio_listio() call, which takes a list of buffers, offsets and lengths to perform a series of read and write operations on a file in a single call. The primary difference from the typical HDF5 API call is that the new routines add information from multiple datasets to the I/O mapping list and construct larger MPI-derived datatypes for collective I/O operations for read and write operations in a separate manner. Internally, the multi-dataset implementation will be similar to the current implementation of collective chunk I/O on a single dataset.

4.1 Top-level design

The following example chart shows the conceptual implementation approach for the new API functions.


In Memory
In File
Dataset 1
Dataset 2
1
2
1
2
Build the list
Addr3 File DS 3 dset 1, Mem DS 1
Addr1 File DS 1 dset 1, Mem DS 2
Addr2 File DS 2 dset 2, Mem DS 1
Addr2 File DS 2 dset 2, Mem DS 2
Sort the list by file address
Addr1 File DS 1 dset 1, Mem DS 2
Addr2 File DS 2 dset 2, Mem DS 1
Addr2 File DS 2 dset 2, Mem DS 2
Addr3 File DS 3 dset 1, Mem DS 1
Perform Collective IO
2
1 2
1

Note that sorting the list by file addresses is necessary because MPI requires the file type to consist of derived datatypes whose displacements are monotonically non-decreasing.

4.2 Code-level design

4.2.1 Existing code

The current implementation achieves multi dataset I/O by creating a new structure, *H5D_dset_info_t*, which contains all the information necessary for I/O on a single dataset. This information was previously contained in structs specific to each layout type. *H5D__read()* and *H5D__write()* then allocate an array of these structs, iterate over all the datasets in the operation, passing the corresponding element of the *H5D_dset_info_t* struct to each dataset's layout's *io_init* operation.

The io_init operations operate mostly as before, except they place info in the *H5D_dset_info_t* struct, and place info on I/O within a single chunk or contiguous dataset in an *H5D_piece_info_t* struct, then insert that struct into a skip list sorted by address. This is analogous to how the chunk map worked before, except it has been extended to work with a mixture of contiguous datasets and chunks. A single chunk or contiguous dataset is called a "piece". This way, the chunks and datasets are sorted by address for MPI, and the only sorting that needs to be done is within individual selections (in the case of point selections), which is accomplished in *H5S_mpio_space_type()* as called by *H5D__all_piece_collective_io()*, again analogous to how it worked before.

We may remove the skip list and instead store pieces in an unsorted array, and shift responsibility for sorting to lower levels of code. The need for the pieces to be sorted by address is specific to MPI I/O, and we are trying to minimize the amount of MPI-specific code in the higher levels of the library, to make it easier to develop high performance back ends for HDF5 that do not use MPI.

4.2.2 Future work

The multi dataset branch has been brought in up to date with the mainline develop branch of HDF5. However there is still work remaining to complete integration with the latest HDF5 features.

Three new features that will need special consideration are parallel compression, and selection I/O[1], and the VOL layer. We will initially fall back to performing I/O on one dataset at a time for parallel compression since it will take some effort to implement parallel compression in the multi dataset case. In addition, the "multi chunk" pathway in H5Dmpio.c, which performs I/O on each chunk individually instead of together in a single operation, does not support multi dataset. We may wish to implement multi dataset support for this pathway in the future.

We intend to support selection I/O with multi dataset fully. This feature introduces a new Virtual File Driver (VFD) interface to pass HDF5 selections to the file driver instead of byte offsets and lengths. We will pass one selection for each piece (chunk or contiguous dataset). The chunk I/O functions that were extended to handle selection I/O were also extended to support multi dataset I/O in the multi dataset branch, so extending this new selection I/O code to handle the multi dataset case will be a natural part of the merge process. The code that currently builds a list of chunk offsets, buffers and chunk file/memory dataspaces will be extended to build a list of piece offsets, buffers, and file/memory dataspaces spanning multiple datasets, with the lists stored in the *io_info* struct which is global to the I/O operation.

The VOL layer allows developers to re-implement HDF5 features by implementing custom callbacks for HDF5 API calls that access the HDF5 file. Since the multi dataset API functions do this, we will need to add VOL callbacks and route the calls through the VOL layer. This is not currently done since the feature was originally implemented before the VOL layer was introduced.

4.3 New API functions

Two new functions, *H5Dread_multi()* and *H5Dwrite_multi()* are proposed here.

4.3.1 H5Dread_multi()

The API function description is as shown below.

```
herr_t H5Dread_multi(size_t count,
                     hid_t dset_id[],
                     hid_t mem_type_id[],
                     hid_t mem_space_id[],
                     hid_t file_space_id[],
                     hid_t dxpl_id,
```

[1] https://docs.hdfgroup.org/hdf5/rfc/selection_io_RFC_210610.pdf

```
                            void *buf[] /*out*/);
```

Parameters:

- count: the number of datasets being accessed (the length of the arrays).
- mem_type_id: array of memory type IDs.
- mem_space_id: array of memory dataspace IDs.
- file_space_id: array of file dataspace IDs.
- dxpl_id: dataset transfer property.
- buf: array of read buffers.

Return:

- A non-negative value if successful; otherwise returns a negative value.

This routine performs collective or independent I/O reads from multiple datasets. In collective mode, all process members of the communicator associated with the HDF5 file must participate in the call.

Each process creates the information required to perform each read in the arrays of parameters and passes the array through to *H5Dread_multi()*.

Brief description for internals after being called:

- Each process constructs an MPI-derived datatype describing the sections from multiple datasets in an HDF5 file to be read.
- All processes end up calling *MPI_File_read_at_all()* once each for collective I/O or *MPI_File_read_at()* once each for independent I/O.
- Each process tidies up and then returns the desired data into the buffer of the info[] array structure.

When an application issues the multi-read call, *file_space_id* array elements are expected to differ among processes participating in the collective operation due to different selections. This means that not only the actual data in the buffers can be distinct (like most collective I/O operations), but the dataset (dataspaces, datatypes, etc…) values for every process can be distinct.

All processes are required to pass the same property values for the *dxpl_id*.

All datasets are required to be in the same file, and this file must be the same across all processes.

Refer to the example section for a better understanding of usage.

The same rule applies to *H5Dwrite_multi(),* detailed in the following section.

4.3.2 H5Dwrite_multi()

The API function description is as shown below.

```
herr_t H5Dwrite_multi(size_t count,
                      hid_t dset_id[],
                      hid_t mem_type_id[],
                      hid_t mem_space_id[],
                      hid_t file_space_id[],
                      hid_t dxpl_id,
                      const void *buf[] /*out*/);
```

Parameters:

- count: the number of datasets being accessed (the length of the arrays).
- mem_type_id: array of memory type IDs.
- mem_space_id: array of memory dataspace IDs.
- file_space_id: array of file dataspace IDs.
- dxpl_id: dataset transfer property.
- buf: array of write buffers.

Returns:

- A non-negative value if successful; otherwise returns a negative value.

This routine performs collective or independent I/O writes to multiple datasets. In collective mode, all process members of the communicator associated with the HDF5 file must participate in the call.

Each process creates the information required to perform each write in the arrays of parameters, and passes the array through to *H5Dwrite_multi()*.

Note that when overlapping selections are used, the data stored in the file for the overlapping regions is undefined (as is the case for H5Dwrite).

Brief description for internals after being called:

- Each process constructs an MPI derived type describing the sections from multiple datasets in an HDF5 file to be written.
- All processes ends up calling *MPI_File_write_at_all()* once each for collective I/O or *MPI_File_write_at()* once each for independent I/O.

4.4 Example cases

These examples are based on the assumption that using multi read API on an HDF5 file with four datasets, ‘d1’, ‘d2’, ‘d3’ and ‘d4’. Using multi write API would be practically identical.

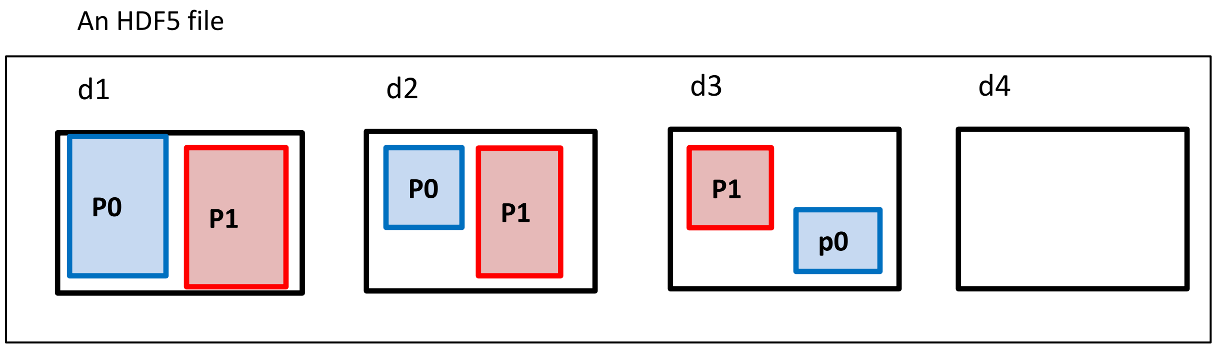
Pseudocode is used to show how the API can be used in a simplified manner focusing on this task’s scope.

4.4.1 **Example1:** all processes read from the same datasets ‘d1’, ’d2’ and ‘d3’

Consider the following as an example running with two processes:

- Rank 0 process (P0) reads data portions from datasets ‘d1’, ‘d2’, and ‘d3’.
- Rank 1 process (P1) reads data portions from datasets ‘d1’, ‘d2’ and ‘d3’.

Chart view:



Pseudocode below:

```
Open datasets ‘d1’, ‘d2’ and ’d3’
Make selections from each dataset.
Set ‘dxpl’ for collective operation.
Set ‘mem_type_ids’, ‘mem_space_ids’, and ‘bufs’ arrays as appropriate.

size_t count = 3        /* three datasets */
If (mpi_rank == 0)      /* P0 */
   hid_t file_space_ids[3] = { {d1’s P0 select}, {d2’s P0 select}, {d3’s P0 select} }

If (mpi_rank == 1)      /* P1 */
   hid_t file_space_ids[3] = { {d1’s P1 select}, {d2’s P1 select}, {d3’s P1 select} }

H5Dread_multi (count, mem_type_ids, mem_space_ids, file_space_ids, dxpl, bufs)
```

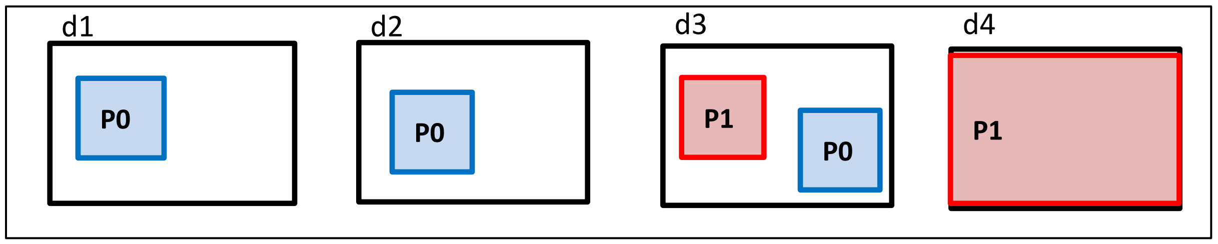
4.4.2 **Example2:** each process read from different datasets or none

Consider the following as an example running with three processes:

- Rank 0 process (P0) reads data portions from datasets ‘d1’, ‘d2’, and ‘d3’.
- Rank 1 process (P1) reads data portions from datasets ‘d3’ and ‘d4’.
- Rank 2 process (P2) does not read anything.

Chart view:

An HDF5 file



Pseudo code below:

```
Open datasets 'd1', 'd2','d3' and 'd4'

Make selections from each dataset.

Set 'dxpl' for collective operation.

Set 'mem_type_ids', 'mem_space_ids', and 'bufs' arrays as appropriate.


If (mpi_rank == 0)      /* P0 */
   count = 3;           /* three datasets */
   hid_t  file_space_ids [3] = { {d1's P0 select},  {d2's P0 select},   {d3's P0 select} }

If (mpi_rank == 1)      /* P1 */
    count = 2;          /* two datasets */
    hid_t  file_space_ids [2] = { {d3's P1 select},   {d4's P1 select} }

If (mpi_rank >= 2)      /* P2 */
    count = 0           /* no dataset access */
    hid_t  *file_space_ids = NULL

H5Dread_multi (count, mem_type_ids, mem_space_ids, file_space_ids, dxpl, bufs)
```

5 Limitations

While the API will work for any datasets and any I/O, the initial implementation will fall back to simply performing I/O for one dataset at a time in some cases. It will only perform simultaneous multi dataset I/O using MPI I/O and only in collective mode. It will only support contiguous and chunked datasets and may not support compression depending on available time for implementation. Note that any other conditions that break collective I/O (datatype conversion, etc.) will also break simultaneous multi dataset I/O.

6 Future Consideration

In addition to addressing the limitations outlined above, according to some discussions, we may be able to consider developing H5Dcreate_multi(), H5Dopen_multi() and H5Dclose_multi() APIs in the future as separate tasks if necessary or requested by the user.

7 Code Repository

The latest version of the multi dataset branch can be found at https://github.com/HDFGroup/hdf5/tree/feature/multi_dataset

[1] Yang M and Koziol Q, 2006. Using collective IO inside a high performance IO software package—HDF5 Technical Report National Center of Supercomputing Applications

[2] Rob Latham, Chris Daley, etc., March 2012. A case study for scientific I/O: improving the FLASH astrophysics code, http://iopscience.iop.org/1749-4699/5/1/015001/article

[3] Qiao Kang, Scot Breitenfeld, Kaiyuan Hou, Wei-keng Liao, Robert Ross, and Suren Byna, December 2021. Optimizing Performance of Parallel I/O Accesses to Non-contiguous Blocks in Multiple Array Variables, 2021 IEEE International Conference on Big Data (Big Data), https://ieeexplore.ieee.org/document/9671638

Revision History

August 28, 2012:	Version 1 by Peter Cao. Circulated internally.
Sep 27, 2012:	Version 2: updated based on internal reviews.
Feb 15, 2013:	Version 3: Updated based on internal reviews. Revised APIs and related contents. The task entry is HDFFV-8313 in JIRA.
March 04, 2013:	Version 3.1: Updates based on internal reviews. More updates and add an example section.
March 07, 2013:	Version 3.2: Some minor updates. Add chart view in the example section.
March 12, 2013	Version 3.3: Some updates from an internal presentation on 03-08-2013.
March 21, 2013	Version 4: revised based on the comments from the internal presentation on 03-08-2013.
January 24, 2022	Version 5: Updated to reflect the status in early 2022.
February 15, 2022	Version 6: Updated based on internal reviews.
May 23, 2022	Version 7: Updated to reflect the latest state of development.