HDF5 Asynchronous I/O VOL Connector

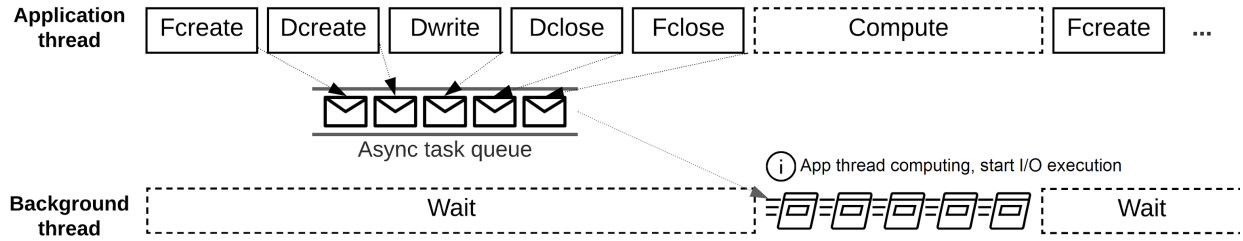
Release 0.1

Houjun Tang, Quincey Koziol, Suren Byna

Sep 14, 2023

GETTING STARTED

Asynchronous I/O enables an application executing the I/O operations at the same time as performing computation or communication tasks. By scheduling I/O operations early and overlapping them with computation or communication, asynchronous I/O can effectively hide the I/O cost and reduce the total execution time.



The HDF5 Asynchronous I/O VOL connector maintains a queue of asynchronous tasks and tracks their dependencies as a directed acyclic graph, where a task can only be executed when all its parent tasks have been completed successfully. Collective operations are executed in the same order as in the application, in an ordered but asynchronous manner. To reduce overhead and avoid contention for shared resources between an application's main thread and the background thread that performs the asynchronous I/O operations, we use a status detection mechanism to check when the main thread is performing non-I/O tasks. We also provide an EventSet interface in HDF5 to monitor asynchronous operation status and to check errors for a set of operations instead of individual ones.

This work is supported by the DOE ECP-ExaIO project.

CHAPTER

ONE

CITATION

- Houjun Tang, Quincey Koziol, Suren Byna, and John Ravi, "Transparent Asynchronous Parallel I/O using Background Threads", IEEE Transactions on Parallel and Distributed Systems 33, no. 4 (2021): 891-902, doi: 10.1109/TPDS.2021.3090322.
- Houjun Tang, Quincey Koziol, Suren Byna, John Mainzer, and Tonglin Li, "Enabling Transparent Asynchronous I/O using Background Threads", 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW), 2019, pp. 11-19, doi: 10.1109/PDSW49588.2019.00006.

1.1 Building with Spack

Spack is a flexible package manager that supports multiple versions, configurations, platforms, and compilers. Async VOL and its dependent libraries (MPI, HDF5, Argobots) can all be installed with the following spack command:

```
spack install hdf5-vol-async
```

If the application needs Async VOL for dataset write buffer management (double buffering), e.g. SW4, Flash-X, and AMReX applications, use the following spack command:

```
spack install hdf5-vol-async +memcpy
```

1.2 Building from source code

We have tested async VOL compiled with GNU (gcc 6.4+), Intel, and Cray compilers on Summit, Cori, Perlmutter, and Theta supercomputers.

1.2.1 Preparation

Define the following configuration parameters, which may be used in the instructions.

```
VOL_DIR : directory of HDF5 Asynchronous I/O VOL connector repository
ABT_DIR : directory of Argobots source code
H5_DIR  : directory of HDF5 source code
```

1. Download the async VOL with Argobots git submodule. Latest Argobots can also be downloaded separately from https://github.com/pmodels/argobots

```
git clone --recursive https://github.com/hpc-io/vol-async.git
```

2. Download the HDF5 source code

```
git clone https://github.com/HDFGroup/hdf5.git
(Optional) git checkout hdf5-1_14_0 # use latest stable version
```

3. (Optional) Set the environment variables for the paths of the codes if the full path of `VOL_DIR`, `ABT_DIR`, and `H5_DIR` are used in later setup.

```
export H5_DIR=/path/to/hdf5
export VOL_DIR=/path/to/vol_async
export ABT_DIR=/path/to/vol_async/argobots
```

1.2.2 Build Async VOL

1. **Compile HDF5**

   HDF5 must be compiled with threadsafety support, and optionally parallel I/O support. CC=cc/CC=mpicc may needed in the following commands.

1.1 Using CMake

```
export HDF5_DIR=$H5_DIR/install
cd hdf5
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$HDF5_DIR -DHDF5_ENABLE_PARALLEL=ON -DHDF5_ENABLE_
↪THREADSAFE=ON \
  -DALLOW_UNSUPPORTED=ON -DCMAKE_C_COMPILER=mpicc ..
make -j && make install
```

1.2 Using Makefile

```
cd $H5_DIR
./autogen.sh
./configure --prefix=$H5_DIR/install --enable-parallel --enable-threadsafe --enable-
↪unsupported
make && make install
```

2. Compile Argobots

```
cd $ABT_DIR
./autogen.sh
./configure --prefix=$ABT_DIR/install
make && make install
```

Note: Using mpixlC on Summit may result in an Argobots runtime error, use xlC or gcc instead.

3. Compile Asynchronous VOL connector

3.1 Using CMake

```
cd $VOL_DIR
mkdir build
```

(continues on next page)

(continued from previous page)

```
cd build
export HDF5_DIR=$H5_DIR/install
cmake -DCMAKE_INSTALL_PREFIX=$VOL_DIR/install -DCMAKE_C_COMPILER=mpicc ..
make && make install
```

3.2 Using Makefile

```
cd $VOL_DIR/src
Prepare Makefile
    Copy a sample Makefile (Makefile.cori, Makefile.summit, Makefile.macos), e.g. "cp␣
↪Makefile.summit Makefile", which should work for most linux systems
    Change the path of HDF5_DIR and ABT_DIR to $H5_DIR/install and $ABT_DIR/install␣
↪(replace $H5_DIR and $ABT_DIR with their full path)
    (Optional) update the compiler flag macros: DEBUG, CFLAGS, LIBS, ARFLAGS
    (Optional) comment/uncomment the correct DYNLDFLAGS & DYNLIB macros
make
```

1.2.3 Set Environmental Variables

Async VOL requires the setting of the following environmental variable to enable it with HDF5. Depending on which way Async VOL is compiled, the libh5async.so library file may be installed to $VOL_DIR/install/lib (CMake) or $VOL_DIR/src (Makefile), and should be set accordingly to the LD_LIBRARY_PATH and HDF5_PLUGIN_PATH.

Linux

```
export LD_LIBRARY_PATH=$VOL_DIR/install/lib:$H5_DIR/install/lib:$ABT_DIR/install/lib:$LD_
↪LIBRARY_PATH
export HDF5_PLUGIN_PATH="$VOL_DIR/install/lib"
export HDF5_VOL_CONNECTOR="async under_vol=0;under_info={}"
```

MacOS

```
export DYLD_LIBRARY_PATH=$VOL_DIR:$H5_DIR/install/lib:$ABT_DIR/install/lib:$DYLD_LIBRARY_
↪PATH
export HDF5_PLUGIN_PATH="$VOL_DIR/install/lib"
export HDF5_VOL_CONNECTOR="async under_vol=0;under_info={}"
```

Note: For some Linux systems, e.g., Ubuntu, `LD_PRELOAD` needs to be set to point to the shared libraries.

1.2.4 Test

1. Compile and run test codes

1.1 Using CMake

Tests are compiled by default when building async VOL with CMake. Running the tests can be done with the ctest command:

```
ctest
```

1.2 Using Makefile

```
cd $VOL_DIR/test
Edit "Makefile":
    Copy a sample Makefile (Makefile.cori, Makefile.summit, Makefile.macos), e.g., "cp␣
↪Makefile.summit Makefile", Makefile.summit should work for most linux systems
    Update H5_DIR, ABT_DIR and ASYNC_DIR to the correct paths of their installation␣
↪directory
    (Optional) update the compiler flag macros: DEBUG, CFLAGS, LIBS, ARFLAGS
    (Optional) comment/uncomment the correct DYNLIB & LDFLAGS macro
make

// Run serial and parallel tests
make check

// Run the serial tests only
make check_serial
```

If any test fails, check `async_vol_test.err` in the test directory for the error message.

Note: Running the automated tests requires Python3.

If the system is not using mpirun to launch MPI tasks, edit mpirun_cmd in pytest.py with the corresponding MPI launch command.

Some file systems do not support file locking, an error `file create failed` may occur and can be fixed with `export HDF5_USE_FILE_LOCKING=FALSE`, which disables the HDF5 file locking. One can also disable HDF5 file locking when compiling HDF5.

1.3 Implicit mode

This mode is only recommended for testing. The implicit mode allows an application to enable asynchronous I/O through setting the environemental variables *Set Environmental Variables* and without any major code change. By default, the HDF5 metadata operations are executed asynchronously, and the dataset operations are executed synchronously.

Note: Due to the limitations of the implicit mode, we highly recommend applications to use the explicit mode for the best I/O performance.

1.4 Explicit mode

This mode is recommended to get the full benefits of async VOL, however, it requires application code changes to use the HDF5 asynchronous and event set APIs.

1. (Required) Set async VOL environment variables

See *Set Environmental Variables*

2. (Required) Init MPI with MPI_THREAD_MULTIPLE

Parallel HDF5 involves MPI collecive operations in many of its internal metadata operations, and they can be executed concurrently with the application's MPI operations, thus we require to initialize MPI with `MPI_THREAD_MULTIPLE` support. Change `MPI_Init (argc, argv)` in your application's code to:

```
MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
```

3. (Required) Use event set and new async API to manage asynchronous I/O operations, see API section for a complete of APIs.

More detailed description on how to enable async VOL can be found in Hello Async Section.

```
// Create event set for tracking async operations
es_id = H5EScreate();
fid = H5Fcreate_async(.., es_id);
did = H5Dopen_async(fid, .., es_id);
H5Dwrite_async(did, .., es_id);
H5Dclose_async(did, .., es_id);
H5Fclose_async(fid, .., es_id);
// Wait for operations in event set to complete
H5ESwait(es_id, H5ES_WAIT_FOREVER, &num_in_progress, &op_failed);
// Close the event set (must wait first)
H5ESclose(es_id);
```

Warning: The buffers used for H5Dwrite can only be changed after H5ESwait unless async VOL double buffering is enabled, see subsection 5 below.

4. (Optional) Error handling with event set

Although it is listed as optional, it is highly recommended to integrate the asynchronous I/O error checking into the application code.

```
// Check if event set has failed operations (es_err_status is set to true)
status = H5ESget_err_status(es_id, &es_err_status);
// Retrieve the number of failed operations in this event set
H5ESget_err_count(es_id, &es_err_count);
// Retrieve information about failed operations
H5ESget_err_info(es_id, 1, &err_info, &es_err_cleared);
// Inspect and handle the error if there is any
...
// Free memory
H5free_memory(err_info.api_name);
H5free_memory(err_info.api_args);
H5free_memory(err_info.app_file_name);
H5free_memory(err_info.app_func_name);
```

5. (Optional) Async VOL double buffering

Applications may choose to have async VOL to manage the write buffer consistency. When enabled, async VOL will automatically makes a memory copy of the buffer for data writes. This increases the runtime memory usage but relieves the burden for the application to manage the double buffering. The copy is automatically freed after the background asynchronous write completes.

5.1 Building with CMake

```
Add -DCMAKE_ENABLE_WRITE_MEMCPY=1 to the cmake command
```

5.2 Building with Makefile

```
Add -DENABLE_WRITE_MEMCPY=1 to the end of the CFLAGS line in src/Makefile
```

Note: Async vol checks available system memory before its double buffer allocation at runtime, using get_avphys_pages() and sysconf(). When there is not enough memory for duplicating the current write buffer, it will not allocate memory and force the current write to be synchronous.

With the double buffering enabled, users can also specify how much memory is allowed for async VOL to allocate, with can be set through an environment variable. When the limit is reached during runtime, async VOL will skip the memory allocation and execute the write synchronously, until previous duplicated buffers are freed after their operation compeleted.

```
export HDF5_ASYNC_MAX_MEM_MB=max_total_async_vol_memory_allocation_in_mb
```

6. (Optional) Include the header file if async VOL internal API is used (see Async VOL APIs section) This is rarely needed by an application.

```
#include "h5_async_vol.h"
```

7. (Optional) Finer control of asynchronous I/O operation

When async VOL is enabled, each HDF5 operation is recorded and put into a task queue and returns without actually executing it. The async VOL detects whether the application is busy issuing HDF5 I/O calls or has moved on to other tasks (e.g., computation). If it finds no HDF5 function is called within a short period (600 ms by default), it will start the background thread to execute the tasks in the queue. This is mainly due to the global mutex from the HDF5, allowing only one thread to execute the HDF5 operations at a given time to maintain its internal data consistency.

The application status detection can avoid an effectively synchronous I/O when the application thread and the async VOL background thread acquire the mutex in an interleaved fashion. However, some applications may have larger time gaps between HDF5 function calls and experience partially asynchronous behavior. To mitigate this, we provide a way by setting an environment variable that informs async VOL to queue the operations and not start their execution until file/group/dataset close time.

When set properly, it make async VOL especially effective for applications that periodically output (write-only) data, e.g., writing checkpoint files periodically.

```
// Start execution at file close time
export HDF5_ASYNC_EXE_FCLOSE=1
// Start execution at group close time
export HDF5_ASYNC_EXE_GCLOSE=1
// Start execution at dataset close time
export HDF5_ASYNC_EXE_DCLOSE=1
```

Async VOL has overhead to manage the asynchronous I/O tasks, and if an application issues a large number of small I/O operations (e.g. scalar attribute create, write, close), the async VOL overhead may be comparable to those operations, and thus resulting in slower I/O performance. We provide an option to disable the asynchronous execution of the small I/O operations and only execute the dataset operations asynchronously, by setting the following environment variable:

```
export HDF5_ASYNC_DISABLE_IMPLICIT_NON_DSET_RW=1
```

1.5 HDF5 Async APIs

HDF5-1.13 and later versions added a number of new APIs that allow applications to take advantage of the asynchronous I/O feature provided by the asynchronous I/O VOL connector. They are part of the HDF5 public header, so users only need to include the HDF5 header file (hdf5.h) to use them.

1.5.1 EventSet APIs

The EventSet APIs can be used to create, manage, and retrieve information from an event set.

```
// H5EScreate creates a new event set and returns a corresponding event set identifier.
hid_t H5EScreate(void);

// H5ESwait waits for all operations in an event set (es_id) and timeout after (timeout)␣
↪nanoseconds.
// The timeout value is accumulated for all operations in the event set.
// It will also return the number of operations (num_in_progress), and indicate whether␣
↪any error
// occurred (err_occurred).
herr_t H5ESwait(hid_t es_id, uint64_t timeout, size_t *num_in_progress, hbool_t *err_
↪occurred);

// H5ESclose terminates access to an event set (es_id).
herr_t H5ESclose(hid_t es_id);

// H5EScancel attempt to cancel all operations in an eventset (es_id), some operations␣
↪may be already
// in progress and cannot be cancelled, which is recoreded (num_not_canceled), as well␣
↪as if any error
// occurred (err_occurred).
herr_t H5EScancel(hid_t es_id, size_t *num_not_canceled, hbool_t *err_occurred);

// H5ESget_count retrieves number of events in an event set (es_id).
herr_t H5ESget_count(hid_t es_id, size_t *count);

// H5ESget_err_status checks if an event set (es_id) has failed operations.
herr_t H5ESget_err_status(hid_t es_id, hbool_t *err_occurred);

// H5ESget_err_count retrieves the number of failed operations in an event set (es_id).
H5_DLL herr_t H5ESget_err_count(hid_t es_id, size_t *num_errs);

// H5ESget_err_info retrieves information about failed operations in an event set (es_
↪id).
// The strings retrieved for each error info must be released by calling H5free_memory.
herr_t H5ESget_err_info(hid_t es_id, size_t num_err_info, H5ES_err_info_t err_info[],␣
↪size_t *err_cleared);

// H5ESfree_err_info frees the error info
herr_t H5ESfree_err_info(size_t num_err_info, H5ES_err_info_t err_info[]);
```

1.5.2 Explicit Asynchronous I/O APIs

The explicit asynchronous I/O APIs looks very similar to the original synchronous HDF5 I/O functions, they have "_async" as the suffix in the function names, and have an additional event set ID added as the last parameter.

File operations

```
hid_t H5Fcreate_async(const char *filename, unsigned flags, hid_t fcpl_id, hid_t fapl_id,
↪ hid_t es_id);

hid_t H5Fopen_async(const char *filename, unsigned flags, hid_t access_plist, hid_t es_
↪id);

hid_t H5Freopen_async(hid_t file_id, hid_t es_id);

herr_t H5Fflush_async(hid_t object_id, H5F_scope_t scope, hid_t es_id);

herr_t H5Fclose_async(hid_t file_id, hid_t es_id)
```

Group operations

```
hid_t H5Gcreate_async(hid_t loc_id, const char *name, hid_t lcpl_id, hid_t gcpl_id, hid_
↪t gapl_id, hid_t es_id);

hid_t H5Gopen_async(hid_t loc_id, const char *name, hid_t gapl_id, hid_t es_id);

herr_t H5Gget_info_async(hid_t loc_id, H5G_info_t *ginfo, hid_t es_id);

herr_t H5Gget_info_by_name_async(hid_t loc_id, const char *name, H5G_info_t *ginfo,
                                 hid_t lapl_id, hid_t es_id);

herr_t H5Gget_info_by_idx_async(hid_t loc_id, const char *group_name, H5_index_t idx_
↪type,
                                H5_iter_order_t order, hsize_t n, H5G_info_t *ginfo,
                                hid_t lapl_id, hid_t es_id);

herr_t H5Gclose_async(hid_t group_id, hid_t es_id);
```

Dataset operations

```
hid_t H5Dcreate_async(hid_t loc_id, const char *name, hid_t type_id, hid_t space_id,
                      hid_t lcpl_id, hid_t dcpl_id, hid_t dapl_id, hid_t es_id);

hid_t H5Dopen_async(hid_t loc_id, const char *name, hid_t dapl_id, hid_t es_id);

hid_t H5Dget_space_async(hid_t dset_id, hid_t es_id);

herr_t H5Dread_async(hid_t dset_id, hid_t mem_type_id, hid_t mem_space_id, hid_t file_
↪space_id,
```

(continues on next page)

(continued from previous page)

```
                    hid_t dxpl_id, void *buf, hid_t es_id);

herr_t H5Dwrite_async(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, hid_t es_id);

herr_t H5Dset_extent_async(hid_t dset_id, const hsize_t size[], hid_t es_id);

herr_t H5Dclose_async(hid_t dset_id, hid_t es_id);
```

Attribute operations

```
herr_t H5Aclose_async(hid_t attr_id, hid_t es_id);

hid_t H5Acreate_async(hid_t loc_id, const char *attr_name, hid_t type_id, hid_t space_id,
                      hid_t acpl_id, hid_t aapl_id, hid_t es_id);

hid_t H5Acreate_by_name_async(hid_t loc_id, const char *obj_name, const char *attr_name,␣
↪hid_t type_id,
                              hid_t space_id, hid_t acpl_id, hid_t aapl_id, hid_t lapl_
↪id, hid_t es_id);

herr_t H5Aexists_async(hid_t obj_id, const char *attr_name, hbool_t *exists, hid_t es_
↪id);

herr_t H5Aexists_by_name_async(hid_t loc_id, const char *obj_name, const char *attr_name,
                               hbool_t *exists, hid_t lapl_id, hid_t es_id);

hid_t H5Aopen_async(hid_t obj_id, const char *attr_name, hid_t aapl_id, hid_t es_id);

hid_t H5Aopen_by_idx_async(hid_t loc_id, const char *obj_name, H5_index_t idx_type, H5_
↪iter_order_t order,
                           hsize_t n, hid_t aapl_id, hid_t lapl_id, hid_t es_id);

hid_t H5Aopen_by_name_async(hid_t loc_id, const char *obj_name, const char *attr_name,␣
↪hid_t aapl_id,
                            hid_t lapl_id, hid_t es_id);

herr_t H5Aread_async(hid_t attr_id, hid_t dtype_id, void *buf, hid_t es_id);

herr_t H5Arename_async(hid_t loc_id, const char *old_name, const char *new_name, hid_t␣
↪es_id);

herr_t H5Arename_by_name_async(hid_t loc_id, const char *obj_name, const char *old_attr_
↪name,
                               const char *new_attr_name, hid_t lapl_id, hid_t es_id);

herr_t H5Awrite_async(hid_t attr_id, hid_t type_id, const void *buf, hid_t es_id);
```

Link operations

```
herr_t H5Lcreate_hard_async(hid_t cur_loc_id, const char *cur_name, hid_t new_loc_id,
                            const char *new_name, hid_t lcpl_id, hid_t lapl_id, hid_t es_
↪id);

herr_t H5Lcreate_soft_async(const char *link_target, hid_t link_loc_id, const char *link_
↪name,
                            hid_t lcpl_id, hid_t lapl_id, hid_t es_id);

herr_t H5Ldelete_async(hid_t loc_id, const char *name, hid_t lapl_id, hid_t es_id);

herr_t H5Ldelete_by_idx_async(hid_t loc_id, const char *group_name, H5_index_t idx_type,
                              H5_iter_order_t order, hsize_t n, hid_t lapl_id, hid_t es_
↪id);

herr_t H5Lexists_async(hid_t loc_id, const char *name, hbool_t *exists, hid_t lapl_id,␣
↪hid_t es_id);

herr_t H5Literate_async(hid_t group_id, H5_index_t idx_type, H5_iter_order_t order,␣
↪hsize_t *idx_p,
                        H5L_iterate2_t op, void *op_data, hid_t es_id);
```

Object operations

```
hid_t H5Oopen_async(hid_t loc_id, const char *name, hid_t lapl_id, hid_t es_id);

hid_t H5Oopen_by_idx_async(hid_t loc_id, const char *group_name, H5_index_t idx_type,
                           H5_iter_order_t order, hsize_t n, hid_t lapl_id, hid_t es_id);

herr_t H5Oget_info_by_name_async(hid_t loc_id, const char *name, H5O_info2_t *oinfo,
                                 unsigned fields, hid_t lapl_id, hid_t es_id);

herr_t H5Ocopy_async(hid_t src_loc_id, const char *src_name, hid_t dst_loc_id, const␣
↪char *dst_name,
                     hid_t ocpypl_id, hid_t lcpl_id, hid_t es_id);

herr_t H5Oclose_async(hid_t object_id, hid_t es_id);

herr_t H5Oflush_async(hid_t obj_id, hid_t es_id);

herr_t H5Orefresh_async(hid_t oid, hid_t es_id);
```

1.6 Async VOL APIs

Besides the HDF5 EventSet and asynchronous I/O operation APIs, the async VOL connector also provides convinient functions for finer control of the asynchronous I/O operations. Application developers should be very careful with these APIs as they may cause unexpected behavior when not properly used. The "h5_async_lib.h" header file must be included in the application's source code and the static async VOL library (libasynchdf5.a) must be linked.

- Set the `disable implicit` flag to the property list, which will force all HDF5 I/O operations to be synchronous, even when the HDF5's explicit `_async` APIs are used.

```
herr_t H5Pset_fapl_disable_async_implicit(hid_t fapl, hbool_t is_disable);
herr_t H5Pset_dxpl_disable_async_implicit(hid_t dxpl, hbool_t is_disable);

// Retrieve the status of disable implicit mode (true when implicit is disabled).
herr_t H5Pget_fapl_disable_async_implicit(hid_t fapl, hbool_t *is_disable);
herr_t H5Pget_dxpl_disable_async_implicit(hid_t dxpl, hbool_t *is_disable);
```

Note: The `disable implicit` flag only becomes effective when the corresponding `fapl` or `dxpl` is actually used by another HDF5 function call, e.g., with `H5Fopen` or `H5Dwrite`. When a new `fapl` or `dxpl` is used by any HDF5 function without setting the `disable implict` flag, e.g., `H5P_DEFAULT`, it will reset the mode back to asynchronous execution.

- Pause/restart all async operations.

```
// Retrieve the pause flag.
herr_t H5Pget_dxpl_pause(hid_t dxpl, hbool_t *is_pause);

// Restart all paused operations, takes effect immediately.
herr_t H5Fstart(hid_t file_id, hid_t dxpl_id);
herr_t H5Dstart(hid_t dset_id, hid_t dxpl_id);
```

- Set and get a delay time for all async operations.

```
// Set a delay time (microseconds) to property list that adds a delay for asynchronous I/
↪O operations.
herr_t H5Pset_dxpl_delay(hid_t dxpl, uint64_t time_us);

// Retrieve the delay time.
herr_t H5Pget_dxpl_delay(hid_t dxpl, uint64_t *time_us);
```

Note: The operations are only delayed when the dxpl is used by another HDF5 function call.

- Convinient APIs for other stacked VOL connectors

Warning: Following APIs are not intended for application use.

```
// Set a dependency parent to a request, should only be used by a another stacked VOL.
herr_t H5async_set_request_dep(void *request, void *parent_request);
```

(continues on next page)

(continued from previous page)

```
// Start all paused operations, should only be used by a another stacked VOL.
herr_t H5async_start(void *request);
```

1.7 Using Async I/O VOL

The HDF5 asynchronous I/O VOL connector (async I/O VOL) allows applications to fully or partially hide the I/O time by overlapping it with the computation. This page provides more information on how applications can take advantage of it.

The async I/O VOL can be used in `explicit` and `implicit` modes.

To use the `implicit` mode, see Implicit Mode

The recommended mode for applications to use the async I/O VOL is ``explicit mode", which is described below.

1.8 Explicit Mode

Using the async I/O VOL is ``explicit mode" requires modifying the application code to use HDF5 asynchronous I/O APIs. This includes three steps:

1) adding the EventSet API calls,

2) switching the existing HDF5 I/O calls to their asynchronous version

3) ensuring data consistency.

1.8.1 EventSet API

The EventSet APIs allows tracking and inspecting multiple asynchronous I/O operations and avoids the burden for the developer to manage the individual operation. An event set (ID) is an in-memory object that is created by the application and functions similar to a "bag" holding request tokens from one or more asynchronous I/O operations. Every asynchronous I/O operation must be associated with an event set, thus the application must create one before the first HDF5 operation:

```
es_id = H5EScreate();
```

The application can then use this event set ID (`es_id`) for all subsequent HDF5 I/O operations. One can also create multiple event set IDs for different operations. Because the HDF5 functions are non-blocking when async VOL is enabled, the application should also wait for all asynchronous operations to finish before exiting, with:

```
H5ESwait(es_id, H5ES_WAIT_FOREVER, &n_running, &op_failed);
```

The H5ESwait is also required before closing an event set, with:

```
H5ESclose(es_id);
```

1.8.2 HDF5 Asynchronous I/O API

The HDF5-1.13 and later versions provide an asynchronous version of all the HDF5 I/O operations. A complete list of them can be found in the API section. Applications need to switch their existing HDF5 function calls to their asynchronous version, which can be done by adding "_async" to the end of the HDF5 function name and adding an event set ID as the last parameter to the function parameter list. One can also maintain both the original synchronous calls and asynchronous with a MACRO and decides which to use at compile time, e.g.,:

```
#ifndef ENABLE_HDF5_ASYNC
H5Fcreate(fname, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
#else
H5Fcreate_async(fname, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT, es_id);
#endif
```

1.8.3 Data Consistency

Once switched to use the event set and asynchronous APIs, the application developer must re-examine the code to ensure data consistency as all the HDF5 I/O calls are non-blocking.

For data write, the application must not modify or free the write buffer until the write operation has been executed by the background thread. This can be done by calling `H5ESwait` before the buffer reuse or free, or by double buffering with alternating one buffer for the asynchronous write and another for new data. Users may also choose to have async VOL to perform the double buffering automatically by adding `-DENABLE_WRITE_MEMCPY` to the `CFLAGS` in the Makefile, however, this requires double the buffer size in memory while the I/O operation is queued, and should be used with caution (the extra memory copy is automatically freed after the operation completes).

For data read, an `H5ESwait` function is needed before the data is accessed for the asynchronous operation to complete and fill the user's buffer. It is recommended to issue asynchronous read operations as early as possible and has their execution overlap with other tasks of the application to.

1.9 An Example

An application may have the following HDF5 operations to write data:

```
// Synchronous file create
fid = H5Fcreate(...);

// Synchronous group create
gid = H5Gcreate(fid, ...);

// Synchronous dataset create
did = H5Dcreate(gid, ..);

// Synchronous dataset write
status = H5Dwrite(did, ..);

// Synchronous dataset read
status = H5Dread(did, ..);
...

// Synchronous file close
```

(continues on next page)

(continued from previous page)

```
H5Fclose(fid);

// Continue to computation
```

which can be converted to use async VOL as the following:

```
// Create an event set to track async operations
es_id = H5EScreate();

// Asynchronous file create
fid = H5Fcreate_async(.., es_id);

// Asynchronous group create
gid = H5Gcreate_async(fid, .., es_id);

// Asynchronous dataset create
did = H5Dcreate_async(gid, .., es_id);

// Asynchronous dataset write
status = H5Dwrite_async(did, .., es_id);

// Asynchronous dataset read
status = H5Dread_async(did, .., es_id);

...

// Asynchronous file close
status = H5Fclose_async(fid, .., es_id);

// Continue to computation, overlapping with asynchronous operations
...

// Finished computation, Wait for all previous operations in the event set to complete
H5ESwait(es_id, H5ES_WAIT_FOREVER, &n_running, &op_failed);

// Close the event set
H5ESclose(es_id);
...
```

Note: More details on using the `explicit` mode are available at Explicit Mode

1.10 Best Practices

1.10.1 Overlap time to hide I/O latency

To take full advantage of the async I/O VOL connector, applications should have enough non-I/O time for the asynchronous operations to overlap with. This can often be achieved to separate the I/O phases of an application from the compute/communication phase. The compute/communication phases have to be long enough to hide the latency in an I/O phase. When there is not enough compute/communication time to overlap, async I/O may only hide partial I/O latency.

1.10.2 Inform async VOL on access pattern

By default, the async VOL detects whether the application is busy issuing HDF5 I/O calls or has moved on to other tasks (e.g., computation). If it finds no HDF5 function is called within a short wait period (600 ms by default), it will start the background thread to execute the tasks in the queue. Such status detection can avoid an effectively synchronous I/O when the application thread and the async VOL background thread acquire the HDF5 global mutex in an interleaved fashion. However, some applications may have larger time gaps than the default wait period between HDF5 function calls and experience partially asynchronous behavior. To avoid this, one can set the following environment variable to disable its active ‘’wait and check” mechnism and inform async VOL when to start the async execution, this is especially useful for checkpointing data.

```
// Start execution at file close time
export HDF5_ASYNC_EXE_FCLOSE=1
// Start execution at group close time
export HDF5_ASYNC_EXE_GCLOSE=1
// Start execution at dataset close time
export HDF5_ASYNC_EXE_DCLOSE=1
```

Warning: This option requires the application developer to ensure that no deadlock occurs.

1.10.3 Mix sync and async operations

It is generally discouraged to mix sync and async operations in an application, as deadlocks may occur unexpectedly. If it is unavoidable, we recommend to separate the sync and async operations as much as possible (ideally using different HDF5 file IDs, even they are opearting on the same file) and set the following FAPL property for the sync operations:

```
fapl_sync = H5Pcreate(H5P_FILE_ACCESS);
hbool_t is_disable = true;
if (H5Pinsert2(fapl_sync,  "gov.lbl.async.disable.implicit", sizeof(hbool_t),
               &is_disable, NULL, NULL, NULL, NULL, NULL, NULL) < 0) {
    fprintf(stderr, "Disable hdf5 async failed!\n");
}
fid_sync = H5Fcreate(fname, H5F_ACC_TRUNC, H5P_DEFAULT, fapl_sync);
H5Pclose(fapl_sync)
```

1.10.4 Forced synchronous execution

Due to the nature of some HDF5 APIs such as H5*exists, H5*get*, H5*set*, they must be executed synchronously and immediately in order to provide the correct return value back to the application, even asynchronous I/O VOL is used. This may lead to an effectively synchronous performance, especially when they are interleaved with other operations that can be executed asynchronously. It is recommended that applications either avoid these calls (by caching HDF5 IDs) or place them at the very beginning of the I/O phase.

1.11 Debug

Async VOL provides a number of options for users to debug potential issues when using it.

1.11.1 Logging

By default, async VOL operates in silent mode and does not print out any message indicating that the asynchronous I/O feature has been enabled. Users may not be certain whether their application is truly utilizing async VOL. It is also possible that the application is not fully converted to use the HDF5 async APIs, and with some operations executed synchronously, only a very limited I/O performance improvement can be observed.

To ensure that async VOL is properly enabled and that the application is using HDF5 async APIs correctly, developers can comment out the following line in h5_async_vol.c, recompile async VOL, and run their application.

```
#define ENABLE_LOG 1
```

The async VOL will then output logging messages to stderr such as the following:

```
[ASYNC VOL LOG] ASYNC VOL init
[ASYNC VOL LOG] entering async_file_create
[ASYNC ABT LOG] entering async_file_create_fn
[ASYNC ABT LOG] Argobots execute async_file_create_fn success
[ASYNC VOL LOG] entering async_file_optional
[ASYNC ABT LOG] entering async_file_optional_fn
[ASYNC ABT LOG] Argobots execute async_file_optional_fn success
...
```

If these messages were not seen, it is likely that the environment variables described in "Getting Started" are not set properly.

Additionally, if the application has an HDF5 I/O function call (e.g. H5Dwrite) that does not show up in the logging messages ("async_dataset_write_fn success"), it usually means that call was not converted to using async API and being executed synchronously.

1.11.2 Debug Message

The logging messages provide a simple way to verify that async VOL is working, however, getting more detailed information on how that I/O operations are executed asynchronously is often needed to get to the root cause when the performance improvement fell short of the expectation. More detailed debug messages can be enabled by commenting out the following line in h5_async_vol.c:

```
#ENABLE_DBG_MSG 1
```

Following is an example of the messages printed out. By default, when running in parallel, only rank 0 will print out the messages in order to avoid duplicated messages from all ranks.

Note: User can change the value of the ASYNC_DBG_MSG_RANK macro in h5_async_vol.c to let a specific rank output the messages to *stderr*. Setting the value to -1 results in each rank output their debug message to a *file* with "async.log.$rank" as the file name.

```
[ASYNC VOL DBG] set implicit mode to 0
[ASYNC VOL DBG] create and append [0x2aaaab73fb2a] to new REGULAR task list
[ASYNC VOL DBG] entering push_task_to_abt_pool
[ASYNC VOL DBG] push task [0x2aaaab73fb2a] to Argobots pool
[ASYNC VOL DBG] 0 tasks already in Argobots pool
[ASYNC VOL DBG] leaving push_task_to_abt_pool
[ASYNC VOL DBG] entering push_task_to_abt_pool
[ASYNC VOL DBG] leaving push_task_to_abt_pool
[ASYNC VOL DBG] async_file_create waiting to finish all previous tasks
[ASYNC ABT DBG] async_file_create_fn: trying to aquire global lock
[ASYNC ABT DBG] async_file_create_fn: global lock acquired
```

Note: Messages with "[ASYNC VOL DBG]" are printed by the application thread entering async VOL, while "[ASYNC ABT DBG]" messages are printed by the background thread when executing the functions.

1.12 Known Issues

1.12.1 Slow performance with metadata heavy workload

Async VOL has additional overhead due to its internal management of asynchronous tasks and the background thread execution. If the application is metadata-intensive, e.g. create thousands of groups, datasets, or attributes, this overhead (~0.001s per operation) becomes comparable to the creation time, and could result in worse performance. There may also be additional overhead due to the *wait and check* mechnism unless `HDF5_ASYNC_EXE_*` is set.

1.12.2 ABT_thread_create SegFault

When an application has a large number of HDF5 function calls, an error like the following may occur:

```
*** Process received signal ***
Signal: Segmentation fault: 11 (11)
Signal code: (0)
Failing at address: 0x0
[ 0] 0 libsystem_platform.dylib 0x00007fff20428d7d _sigtramp + 29
[ 1] 0 ??? 0x0000000000000000 0x0 + 0
[ 2] 0 libabt.1.dylib 0x0000000105bdbdc0 ABT_thread_create + 128
[ 3] 0 libh5async.dylib 0x00000001064bde1f push_task_to_abt_pool + 559
```

This is due to the default Argobots thread stack size being too small, and can be resovled by setting the environement variable:

```
export ABT_THREAD_STACKSIZE=100000
```

1.12.3 EventSet ID with Multiple Files

Currently each event set ID should only be associated with operations of one file, otherwise there can be unexpected errors from the HDF5 library. This patch (applies to the HDF5 develop branch) may be needed when an HDF5 iteration error occurs.

1.12.4 Attribute Open After Close

Due to an issue in the HDF5 library handling HDF5 VOL objects, calling H5Aopen to the same attribute that was previously close may cause an error like the following:

```
../../src/H5Fint.c:664: H5F__get_objects_cb: Assertion `obj_ptr' failed.
```

This patch (applies to the HDF5 develop branch) is a temporary fix for the issue.

1.12.5 Synchronous H5Dget_space_async

When an application calls H5Dget_space_async, and uses the dataspace ID immediately, a deadlock may occur occationally. Thus we force synchronous execution for H5Dget_space_async. To re-enable its asynchronous execution, set the following environement variable:

```
export HDF5_ASYNC_DISABLE_DSET_GET=0
```

1.13 Copyright

1.14 License

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search