API and Usage of libhio on XC-40 Systems

Nathan T. Hjelm, Howard Pritchard
Los Alamos National Laboratory
Los Alamos, NM
{*hjelmn, howardp*}*@lanl.gov*

***Abstract*—High performance systems are rapidly increasing in size and complexity. To keep up with the Input/Output (IO) demands of applications and to provide improved functionality, performance and cost, IO subsystems are also increasing in complexity. To help applications to utilize and exploit increased functionality and improved performance in this more complex environment, we developed a user-space Hierarchical Input/Output (HIO) library: libhio. The current version of libhio (v1.4.1) provides a number of features relevant to XC-40 systems including an HD5 plugin and support for Lustre and DataWarp filesystems.**

In this paper we discuss the libhio Application Programming Interface (API) and its usage on XC-40 systems with both DataWarp and Lustre filesystems. In addition, we describe the HDF5 plugin and report its performance. Planned improvements to the HIO HDF5 VFD will also be discussed.

***Keywords*-libhio; DataWarp; Cray; Aries; XC40; HDF5; Parallel IO**

I. INTRODUCTION

High performance systems are rapidly increasing in size and complexity. To keep up with the Input/Output (IO) demands of applications and to provide improved functionality, performance and cost, IO subsystems are also increasing in complexity. To help applications to utilize and exploit increased functionality and improved performance in this more complex environment, we developed a user-space Hierarchical Input/Output (HIO) library: libhio. The current version of libhio (v1.4.1.1) provides the following features relevant to XC-40 systems:

- Support for automatic or fine-grained staging or data from the DataWarp filesystem to the parallel file system.
- Support for automatic space management on the DataWarp filesystem.
- Support for setting file striping on both DataWarp and Lustre.
- Provides a Lustre-optimized data layout for $n \rightarrow 1$ IO workloads.
- Support for both per-job and persistent DataWarp allocations.

In this paper we discuss the libhio Application Programming Intrface (API) and its usage on XC-40 systems with both DataWarp and Lustre filesystems.

In addition to the libhio API, the libhio package also provides an HDF5 interface via an HIO HDF5 virtual file driver (VFD). We describe the minor modifications that need to be made to use the HIO VFD in applications currently using HDF5.

The rest of this paper is organized as follows. Section III provides an overview of the API provided by libhio and HDF5 support. Section VI details the expected usage and configuratione of libhio on Cray XC-40 systems. Finally, section VIII gives related works and concludes with future work.

II. OVERVIEW AND BACKGROUND

This section provides background and a high-level overview of libhio. This overview includes some of the design goals and features relevant to Cray XC 40 systems.

A. About libhio

libhio is a user-space software package developed to support writing application data to hierarchical data storage systems. These are systems that may be comprised of one or more logical layers including parallel file systems, burst buffers (such as Cray DataWarp), and local memory or Solid State Disks (SSDs). At the lowest level, libhio provides a POSIX-like interface for reading and writing application data. This interface includes blocking and non-blocking read/write calls with both contiguous as well as strided (read scatter/write gather) Input/Output (IO) operations. There is currently no support in libhio to directly provided a description of stored data. If desired an application can use libhio as a backend to HDF5. The library also provides support for automatic space managment on burst buffer storage systems as well as automated fallback to alternate write locations if a data store becomes unavailable.

libhio does not support parallel IO but instead supports parallel-aware IO. This is needed to ensure the dataset completion semantics guaranteed by libhio. At this time libhio supports Message Passing Interface (MPI). Other parallel programming models may be added in a future version.

B. Motivation

The primary motivation for creating libhio was the deployment of DataWarp[1] at Los Alamos National Laboratory (LANL). This new "burst-buffer" storage architecture was expected to require application modification to handle the

movement of data from the DataWarp filesystem to Lustre. By developing a new library and modifying applications to use libhio we can add DataWarp support to applications and also buffer them from changes as burst-buffer IO stacks evolve and mature.

C. Namespace

libhio was designed to support both current and future data storage technologies. We expect in the future data storage systems may not have a POSIX-like "filesystem". To support these non-POSIX storage systems we designed libhio around an abstract namespace. Each level of the abstract namespace is represented by a libhio object and a set of APIs. The namespace and associate libhio C-language object types are as follows:

- **Context:** *hio_context_t* All data managed by an hio instance.
- **Dataset:** *hio_dataset_t* A complete collection of files associated with a particular type of data. For example, all files of an $n \rightarrow n$ restart file. Each dataset has an associated identifier that identifies the dataset instance.
- **Element:** *hio_element_t* Named data storage unit with a dataset.

All libhio object types are derived from the opaque C-language object *hio_object_t*. Any API taking an *hio_object_t* will accept any of the above types.

D. Data Layout

In general libhio provides no guarantees of the physical structure of a dataset. In the case of POSIX-like filesystems, however, the current implementation of libhio will store all data associated with a context, dataset, identifier triple in the sub-directory: *context_name.hio/dataset_name/identifier/*. This behavior can be modified by changing a configuration variable. See section VI.

This directory and its sub-directories contain application data as well as the configuration and performance information associated with the dataset. It may also, in some cases, contain information mapping the dataset data from the underlying files to the application offset space. Future releases of libhio may change this behavior but will remain backward compatible.

E. HDF5 Plugin

An HDF5 plugin was developed to enable applications using HDF5 to exploit libhio's capabilities. The plugin implements a *virtual file device*[2] to map the HFDF5 file format to libhio's namespace and data layout. HDF5 files written using the plugin can be used with HDF5 utilities such as *hd5dump*. The plugin is compatible with both HDF5 1.8 and 1.10 releases, as well as Cray-packaged versions of HDF5. Minor modifications to HDF5 applications are required to make use of the plugin. The plugin's extensions to the HDF5 api are described in Section III.

III. Application Programming Interface

This section gives a high-level overview of the APIs provided by libhio for configuration and each level of the abstract namespace. A subset of libhio APIs can be found in Table I. A complete description of the libhio API can be found in the documentation provided with libhio.

The APIs in libhio are defined to be local or collective. A collective API must be called by all participating MPI processes. An participating process is defined as an MPI process that is a member of the MPI Communicator passed to one of the context initialization functions. Unless specified libhio API calls are not collective.

IV. Configuration Interface

libhio provides a flexible configuration interface. The basic units of configuration in libhio are known as control variables. These variable are simple "key = value" pairs that allow applications to control specific libhio behavior and fine-tune performance. Variables apply to specific libhio objects including contexts, datasets, and elements. All control variables set on the lowest ranked participating MPI processe are propogated to all other participating MPI processes during collective libhio object APIs. Some variables, such as debugging verbosity and tracing, can be set on an object after one of these collective points to modify behavior of a specific MPI process. libhio object configuration can be modified through environment variables, input files (user specified and global system configuration), and configuration APIs. If a variable is set via multiple mechanisms the final value will be set according to the following precedence:

1) System configuration (not yet implemented, highest)
2) libhio APIs
3) Environment variables of the form *HIO_variable_name*
4) User-specified configuration file (lowest)

Variables set using the specific context, dataset, or element name take precidence over globally-set variables.

1) File Configuration: During context creation a user can specify a file that contains values for configuration variables. These values can be specified globally and apply to any libhio object or they can be specified to apply to a named libhio object. The user-specified configuration file is passed as a parameter to the context creation APIs; `hio_init_mpi` and `hio_init_single`. The configuration file can be divided into sections using keywords specified within square brackets ([and]). The keywords currently recognized by libhio are: global, context:context_name, dataset:dataset_name, and element:element_name. Variable values not within a section, by default, apply globally. Generally, any line beginning with a # character is considered a comment and ignored by the configuration parser. For a configuration example see Figure 1.

If desired an application can also specify a string prefix that will be used to only match specific lines in the input file.

This allows the application to add hio specific configuration parameters to an existing file. This string prefix may include the # character.

2) Environment Configuration: libhio also supports using environment variables for configuration. These variable are of the form:

- *HIO_variable_name* - For variables that apply globally.
- *HIO_context_context_name_variable_name* - For context specific variables.
- *HIO_dataset_dataset_name_variable_name* - For dataset specific variables on all contexts.
- *HIO_dataset_context_name_dataset_name_variable_name* - For dataset specific variables on a specific context.

3) Configuration APIs: libhio also provides APIs to get the number of configutation variables (`hio_config_get_count`), info about configuration variables (`hio_config_get_info`), and get (`hio_config_get_value`) and set (`hio_config_set_value`) the value of a configuration variable.

A. Context

A context encompasses all of an application's interaction with libhio. Contexts are created by the `hio_init_single` and `hio_init_mpi` functions and destroyed by the `hio_fini` function. These functions are collective. Each context can be associated with one more data storage destinations called data roots.

1) Data Roots: libhio data destinations are knows as data roots. Currently, the only destinations supported by libhio are POSIX-like filesystems including Lustre, Panasas, and DataWarp. Support for additional destinations will be added as they are needed.

The data roots associated with a context can be set by setting the "data_roots" configuration variable on a context or by setting the *HIO_data_roots* environment variable. The *data_roots* configuration variable is a comma-delimited list of data roots. It is only valid to set this variable before the first call to *hio_dataset_open()* on a context. After that point the value is fixed.

If an application specifies more that one data root libhio will automatically switch between data roots in the event of a failure. If a data root fails the application is notified by the error code `HIO_ERR_IO_PERMANENT` on return from a libhio API function. The application can choose how to proceed. This includes delaying writing a dataset to a future time or reopening the dataset using the next available data root. The objective of this feature is to allow the application to make progress when possible in the face of filesystem failures with minimal logic embedded in the application itself.

When specifying a data root the user can alternately specify a module name prepended to the data root path followed by a ":". As of libhio v1.4.1 the available modules are *datawarp* and *posix*. If no module is specified then a module appropriate for the data root will be chosen automatically if possible. To use the default DataWarp path on a supported system the application only needs to specify one of the special strings: *datawarp*, or *dw*. Persistent DataWarp allocations can be specified by appending "-name" where name is the name of the persistent allocation. For example, if the persistent allocation is named "foo" the user can use this allocation by adding "datawarp-foo" to the list of data roots. Module name specification is case-insensitive so DataWarp is treated the same as datawarp.

The default data roots are set to all available DataWarp allocations followed by the current working directory. Per-job DataWarp allocations take precident over persistent allocations.

B. Dataset

A dataset is a complete collection of output associated with a particular type of data (ie. restart). Each dataset is uniquely identified with a string name and an integer identifier ($>= 0$). Dataset objects are created with the `hio_dataset_alloc` function. This function takes a dataset name, dataset identifier, flags and a dataset mode. The flags are used to specify whether the dataset should be open for reading (`HIO_FLAG_READ`) or writing (`HIO_FLAG_WRITE`). libhio does not currently support opening a dataset for both read and write for some configurations. Additional flags that can be specified at dataset allocation time are create (`HIO_FLAG_CREATE`) and truncate (`HIO_FLAG_TRUNC`). The truncate flag is equivalent of calling `hio_dataset_unlink()` on the dataset instance before opening it and implies create.

Two dataset modes are supported: `HIO_SET_ELEMENT_UNIQUE` and `HIO_SET_ELEMENT_SHARED`. These modes correspond to $N \rightarrow N$ and $N \rightarrow 1$ IO respectively. A unique element mode gives a unique offset space for all elements in a dataset for each participating MPI process. The shared mode shares the element offset space between all participaring MPI processes.

The dataset identifier is an integer greater than or equal to zero that identifies the current dataset instance. libhio provides two special identifiers for `hio_dataset_alloc`; HIO_DATASET_ID_HIGHEST and HIO_DATASET_ID_NEWEST. Using one of these special identifiers will cause libhio to open the dataset insance with the highest identifier or most recent modification time respectively. These special identifiers are not valid when creating a new dataset.

Once a dataset object has been allocated it can be opened with `hio_dataset_open`. Opened datasets must be closed with `hio_dataset_close`. Opening and closing datasets are collective operations. Dataset objects are freed with the `hio_dataset_free` function.

```
# these are global
    context_base_verbose = 10
[global]
# these are also global
    data_roots = DW, posix:/lscratch1,/lscratch2
[context:foo]
# apply only to context "foo"
    data_roots = posix:/lscratch1
[dataset:restart]
# apply to dataset "restart" in any context
    stripe_width = 4M
```

Figure 1: Example libhio configuration script without a prefix

A dataset instance can be destroyed with `hio_dataset_unlink`. This function takes the dataset name, dataset identifier, and an unlink mode. The unlink mode can be one of: `HIO_UNLINK_MODE_CURRENT`, `HIO_UNLINK_MODE_FIRST`, or `HIO_UNLINK_MODE_ALL`. These modes correspond to removing the dataset instance from to the active data root, the first data root containing the instance, and all data roots.

C. Element

Dataset elements are the lowest object in the libhio hierarchy. They represent named data within a dataset. A dataset can have zero or more elements. Dataset elements are opened with `hio_element_open` and closed with `hio_element_close`. libhio provides functions for both blocking and non-blocking contiguous read and write of dataset elements. It also provides both strided read (read scatter) and strided write (write gather). libhio does not currently support conflicting (overlapping) writes to the same element when writing an element in a dataset opened with a shared offset mode. The granularity for writes is on bytes. It is up to the application to ensure that no conflicting writes are issued. None of the functions within the set of element APIs are collective. With the current version of libhio the flags field to `hio_element_open` must be 0 as no flags are currently defined for dataset elements.

V. LIBHHIO HDF5 PLUGIN INTERFACE

In order to use libhio's HDF5 plugin, minor additions must be made to the application. At a minimum, the following plugin methods must be invoked by the application prior to opening an HDF5 file:

1) call the plugin initialization method - `H5FD_hio_init`,
2) initialize an *hio_settings_t* object using `H5FD_hio_settings_init`,
3) set an *identifier*, if desired, and *element* for the HDF5 file using `H5FD_hio_set_setid` and `H5FD_hio_set_elem_name`, respectively,
4) Set the MPI Communicator using `H5FD_set_comm` if accessing the HDF5 file in shared mode,
5) Create an HDF5 file access property list using `H5Pcreate(H5P_FILE_ACCESS)` and associate the *hio_settings_t* object created earlier with this property list using `H5Pset_fapl_hio`,
6) Create the HDF5 file using *H5Fcreate* using this file access property list,
7) Access the file using regular HDF5 methods,
8) Closs the file using regular HDF5 methods,
9) call the plugin finalization method - `H5FD_hio_term`.

A libhio configuration file may also be associated with an HDF5 file using the `H5FD_hio_set_config` and `H5FD_hio_set_config_prefix` methods. The plugin supports libhios' contiguous and strided file access modes, as well as the dataset's *dataset_mode*.

An example code is provided in the libhio distribution.

VI. XC-40 USAGE

This section describes the typical usage of libhio on Cray XC-40 systems both with and without DataWarp. In addition, we also describe some of the common configuration options for these systems. A more complete list of configuration options is available in the documentation provided with libhio.

A. DataWarp Support

The DataWarp support in libhio provides a portable way to use the features of the DataWarp file system. When a DataWarp data root is used libhio will, by default, mark any successfully written dataset as eligible for stage out at the end of the job. Additionally, to increase robustness in face of potential datawarp filesystem failure libhio will periodically mark a completed dataset to be immediately staged out to the parallel filesystem. This behavior can be modified by setting the `datawarp_stage_mode` configuration variable on the dataset object. Valid values for this variable are auto (default), end_of_job, disable, and immediate. The target of any stage out operation is taken from the next available data root. *Ex. data_roots=datawarp,/lscratch2/foo/data* will stage complete datasets to the */lscratch2/foo/data* directory. The stage out target can be modified by setting

Return Type	API	Arguments
CONFIGURATION API		
hio_return_t	hio_config_set_value	hio_object_t object, const char *variable, const char *value
hio_return_t	hio_config_get_value	hio_object_t object, char *variable, char **value
hio_return_t	hio_config_get_count	hio_object_t object, int *count
hio_return_t	hio_config_get_info	hio_object_t object, int index, char **name, hio_config_type_t *type, bool *read_only
CONTEXT API		
hio_return_t	hio_init_single	hio_context_t *new_context, const char *config_file, const char *config_file_prefix, const char *context_name
hio_return_t	hio_init_mpi	hio_context_t *new_context, void *comm, const char *config_file, const char *config_file_prefix, const char *name
hio_return_t	hio_fini	hio_context_t *context
DATASET API		
hio_return_t	hio_dataset_alloc	hio_context_t context, hio_dataset_t *set_out, const char *name, int64_t set_id, int flags, hio_dataset_mode_t mode
hio_return_t	hio_dataset_free	hio_dataset_t *dataset
hio_return_t	hio_dataset_open	hio_dataset_t dataset
hio_return_t	hio_dataset_close	hio_dataset_t dataset
hio_return_t	hio_dataset_unlink	hio_context_t context, const char *name, int64_t set_id, hio_unlink_mode_t mode
ELEMENT API		
hio_return_t	hio_element_open	hio_dataset_t dataset, hio_element_t *element_out, const char *name, int flags
hio_return_t	hio_element_close	hio_element_t *element
ssize_t	hio_element_write	hio_element_t element, off_t offset, unsigned long reserved0, const void *ptr, size_t count, size_t size
ssize_t	hio_element_read	hio_element_t element, off_t offset, unsigned long reserved0, void *ptr, size_t count, size_t size
hio_return_t	hio_element_flush	hio_element_t element, hio_flush_mode_t mode

Table I: Subset of libhio API as of version v1.4.1 broken down by the libhio object type.

the `datawarp_stage_out_destination` configuration variable on the dataset.

B. Data Layout

When writing datasets to Lustre, DataWarp, or other POSIX-like file systems libhio provides multiple output options to optimize write IO performance. In the current libhio release (v1.4.1.1) three layouts are supported; basic, strided, and file_per_node. The basic layout writes a file per element (in shared element mode) or a file per element per MPI rank (in unique element mode). The file_per_node layout appends all application data to single file for each compute node and writes a layout file is in json format describing the location of the application data. A more detailed description of the layouts can be found in [3]. We recommend using basic mode for DataWarp filestems, and file_per_node mode for Lustre filesystems. The layout can be configured by setting the `dataset_file_mode` configuration variable.

C. POSIX Dataset

On POSIX filesystems a dataset is stored in hierarchial directory structure made up of the context name, dataset name, and dataset identifier. The default sub-directory has the form *context_name.hio/dataset_name/identifier/*. This behavior can be modified by setting the `dataset_posix_directory_mode` configuration variable on the dataset. The accepted values are hierarchial (default) and single. When set to single the sub-directory has the form *context_name.dataset_name.identifier.hiod*. When using DataWarp it is recommended that `dataset_posix_directory_mode` be set to single.

D. Striping

The striping parameters used when creating data files within a dataset can be set on supported filesystems by setting the `stripe_count` and `stripe_size` configuration variables on the dataset. This feature is currently only supported on Lustre filesystems and DataWarp support is awaiting software support from DataWarp software stack.

For datasets using the shared offset mode and basic layout the default striping parameters are set to use 90% of the available Lustre Object Storage Targets (OSTs) with a $1MB$ stripe size. With the file_per_node layout the striping is based on the block size, number of OSTs, and number of local MPI processes. Striping parameters must fall within the constraints set by the underlying filesystem.

When writing to a DataWarp filesystem the may need to be set for the stage-out location on Lustre. This is done automatically based on the dataset offset mode, file size, and Lustre OST count. A user can control the striping on files staged out to Lustre by setting the `datawarp_stage_out_stripe_count` and `datawarp_stage_out_stripe_count` configuration variables on the dataset.

E. Lustre Stripe Locking

The user can specify a Lustre locking mode using the `lock_mode` configuration variable. This variable is not expected to provide any performance benefit to datasets with the unique offset mode. As of libhio v1.4.1 the available modes are default, group, and no-expand. The default uses the best available mode while still guaranteeing byte-level write granularity. The group mode (default for the file_per_node) turns on Lustre group locking which effectively disables stripe locks. This mode can greatly improve performance but changes the required write conflict granularity from byte to page ($4kB$). This is due to underlying requirements from the filesystem.

F. Space Management

libhio supports automatic space management on DataWarp filesystems. This support is handled through the discovery and deletion of DataWarp resident libhio datasets. The maximum number of dataset instances left resident on DataWarp can be controlled by setting the `datawarp_keep_last` configuration variable. The default is to keep at most 1 complete dataset instance pending stage-out. When a dataset instance is closed if the number of resident datasets exceeds the threshold then libhio will either cancel (end_of_job stage mode) or wait for completion of (immediate stage mode) stage out of the oldest resident dataset instance. Once this is complete the dataset is unlinked from DataWarp.

VII. Related Work

Many of the features and optimizations found in libhio are present in other IO libraries. *plfs*[4][5] is a user-space filesystem, developed at LANL, that refactors application IO to optimize it for Lustre and other parallel filesystems. The *file_per_node* output mode is modeled directly after the optimizations used by *plfs* to refactor all IO as sequential IO on the underlying filesysem. Unlike *plfs*, libhio does not attempt to implement a user-space filesystem.

The API of libhio shares many of the same design characteristics as those of ADIOS[6]. libhio differs from ADIOS in the way IO is defined. Whereas libhio provides IO calls similar to POSIX (pointer, size, and offset) ADIOS has additional APIs to support describing the data being written. This description includes the type, dimensions, layout, size, and other details.

VIII. Future Work

In this paper, we presented details on the libhio API and its usage on Cray XC-40 systems. In the future, we plan to investigate improvements to libhio to improve performance.

A unique feature of libhio is its automatic management of space and the migration of datasets from temporary data stores, such as Cray DataWarp, and more permenant storage. We will look at adapting libhio to support additional burst-buffer implementations including distributed burst-buffers both with and without a globally visible file space. We intend to investigate adding additional capabilities including in-memory checkpointing.

Acknowledgment

The authors would like to thank Alliance for Computing at Extreme Scale (ACES) management and staff for their support. Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA. Los Alamos National Laboratory is operated by Los Alamos National Security, LLC for the NNSA. LA-UR-18-24391

References

[1] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. Wright, "Architecture and design of cray datawarp," *Cray User Group CUG*, 2016.

[2] "HDF5 virtual file layer," https://support.hdfgroup.org/HDF5/doc/TechNotes/VFL.html, Nov. 1999.

[3] N. Hjelm and C. Wright, "libhio: Optimizing io on cray xc systems with datawarp," in *Cray User Group CUG*, 2017.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE, 2009, pp. 1–12.

[5] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–5.

[6] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.