libhio: Optimizing IO on Cray XC Systems With DataWarp

Nathan T. Hjelm, Cornell Wright
Los Alamos National Laboratory
Los Alamos, NM
{*hjelmn, cornell*}*@lanl.gov*

***Abstract*—High performance systems are rapidly increasing in size and complexity. To keep up with the *Input/Output* (IO) demands of *High Performance Computing* (HPC) 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 new Hierarchical IO (HIO) library: libhio. In this paper we present the motivation behind the development, the design, and features of libhio. We detail optimizations made in libhio to support checkpoint/restart IO workloads on the Trinity supercomputer, a Cray XC-40 at Los Alamos National Lab. We compare the large scale whole file read/write performance or IOR when using libhio against using POSIX when writing to Cray *DataWarp*™.**

***Keywords*-libhio; DataWarp; Cray; Aries; XC40; Burst-Buffer; SSD; IO**

I. Introduction

High performance computing systems are rapidly increasing in size and complexity. Unfortunately, the performance of IO storage systems have not kept pace with the compute performance. To keep up with the IO demands of applications and to provide improved functionality, performance and cost, IO subsystems are also increasing in complexity. This has lead to the introduction of tiered storage systems that include small fast storage devices known as burst-buffers. To help applications to utilize and exploit the increased functionality and improved performance in this more complex environment, we developed a new Hierarchical Input/Output (HIO) library: *libhio*.

libhio is an open-source[1] user-space software library intended for writing data to hierarchical data storage systems. These systems may be comprised of one or more logical layers including parallel file systems, burst buffers, and local memory. libhio provides a number of features including:

- An interface that is always thread-safe. No application-level locking is necessary to use libhio.
- Support for burst-buffers implemented by the Cray *DataWarp* product.
- Support for managing available burst-buffer space.
- Support for scheduling automatic drain from burst-buffers to more permanent data roots (PFS).
- A simple interface that provides a minimal POSIX-like IO interface. This is intended to make integration of *libhio* into applications easier. Where necessary we broke from POSIX semantics to allow for optimization within libhio.
- Allows for user specification of filesystem specific "hints" while attempting to provide good defaults.
- A configuration interface that allows applications to specify configuration options via the environment, Application Programing Interface (API) calls, and file parsing.
- Supports transparent (when possible) fall back on other destinations if part of the IO hierarchy fails. For example, falling back on a parallel file system on failure of a burst-buffer.
- Provide full support for existing IO usage models including $n \to 1$ (shared file), $n \to n$ (file-per-process), and $n \to m$. Also provide support for refactoring IO to change $n \to 1$ and $n \to n$ to $n \to m$.
- An abstract namespace with internal support to re-factor the output to a single or multiple POSIX files.
- Provide support for querying performance characteristics.

The rest of this paper is organized as follows. Section II provides the motivation behind *libhio* and some of the goals that came out of the development. Section III provides an overview of the design and current implementation of *libhio*. Section IV outlines the test environment and presents benchmarks using the IOR benchmark. Sections V and VI conclude with related and future work.

II. Background and Motivation

A. DataWarp

DataWarp™[1] is the Cray burst-buffer product in use on the Trinity supercomputer. The product consists of a hardware product made up of *Solid State Disk*s (SSDs) distributed throughout the cluster and an API and software function for accessing the SSD. per job space allocation, transferring data between the burst buffer and paralel file system (staging), and configuration and administrative control. Space on the burst-buffer can be allocated in number of configurations; persistent, cache, dedicated, and shared striped. For this paper we will focus only on shared striped

[1]The source code to *libhio* can be found at github: github.com/hpc/libhio

allocations as they are the most relevant to the checkpoint/restart usage model. In shared striped mode a burst-buffer filesytem is created per compute job across one or more *DataWarp* nodes. An application can request that data be staged into the burst-buffer before the compute job starts. Once the data stage-in has completed the filesystem is mounted on each compute node allocated to the job using Cray's DVS product. When the compute job has completed data is staged out of of the burst-buffer filesystem to the parallel filesystem. The burst-buffer filesystem is then destroyed and the *DataWarp* resources are returned. *DataWarp* APIs or work-load manager directive are used to indicate which files or directories need to be staged in or out of the burst-buffer filesystem.

B. Motivation

The installation of a new *DataWarp* storage system as part of the Trinity[4] supercomputer at Los Alamos National Lab (LANL) was the primary motivation for the development of a new IO library. To make the best use of *DataWarp*, API calls are required to indicate when and which file needs to be migrated from the job specific *DataWarp* filesystem to the Lustre parallel filesystem. These calls could have been added to applications in two ways; directly modify each application to make the calls, or implement a software layer that makes the API calls on behalf of the application. Though each of these approaches require directly modifying applications we chose to implement a new software layer. We chose this approach because we can not rely on the same burst-buffer architecture and APIs being available on future systems.

C. Goals

Additional goals were created for this new software layer:

- Easy to integrate with existing applications. The target applications do not use IO middlewhere and use standard POSIX, C, or fortran APIs to write data.
- Implement IO best practices for POSIX-like filesystems once instead of per application.

III. Design

This section provides a high-level overview of the design and features of *libhio*.

A. Namespace

libhio does not guarantee a POSIX namespace but is instead designed around the concept of an abstract namespace. The libhio namespace consists of four components:

- **Context:** All data managed by an hio instance.
- **Dataset:** A complete collection of output associated with a particular type of data. For example, all files of an $n \rightarrow n$ restart dump.
- **Identifier:** Particular instance of a dataset. This is a positive integer and is expected to be an increasing sequence.
- **Element:** Named data within a *libhio* dataset.

B. 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 defined to be collective across all participating processes. A participating process is defined as a process that is a member of the Message Passing Interface (MPI) communicator passed to the `hio_init_mpi()` function. 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.

Data roots can be set by setting either the *HIO_data_roots* environment variable or by setting the *data_roots* configuration variable on a context object using *hio_config_set_value()*. It is only valid set this variable before the first call to *hio_dataset_open()* on a context. The *data_roots* configuration variable is a comma-delimited list of data roots.

If an application specified more that one data root *libhio* will automatically switch between data roots in the event of a failure. If a particular data root fails the application is notified by an error code 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 retrying the dataset write with an alternative data root. The objective 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 prepending the data root path with the module name followed by a ":". As of libhio version 1.4.0.0 the available modules are *datawarp* and *posix*. If no module is specified then a module appropriate for the data root will be chosen automatically. To use the default *DataWarp* path on a supported system the application only needs to specify one of the special strings: *datawarp*, or *dw*. Module name specification is case-insensitive so DataWarp is treated the same as datawarp.

C. 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 and freed with the `hio_dataset_free()` function. *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 with the highest identifier or most recent modification data respectively. 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 across all participating processes.

D. Element

Elements are named data within a dataset. Elements are opened using the `hio_element_open()` call and closed using `hio_element_close()`. *libhio* provides APIs for blocking and non-blocking read and write for both contiguous and strided data. None of the element APIs are collective. Overlapping writes are not supported by *libhio* and it is up to the application to ensure that conflicting writes are not issued.

E. Configuration and Performance Variables

libhio provides a flexible configuration interface. The basic unit of configuration used by libhio for configuration are control variables. Control variables are simple "key = value" pairs that allow applications to control specific libhio behavior and fine-tune performance. A control variable may apply globally or to specific HIO objects such as contexts or datasets. Control variables can be set via environment variables, API calls, or configuration files. For scalability, all configuration is applied at the MPI process with rank 0 in the MPI communicator used to create the context. Configuration is then propagated to all other MPI processes during *hio_dataset_open*. If any variable is set via multiple mechanisms the final value will be set according to the following precedence:

1) System administrator overrides (highest)
2) libhio API calls
3) Environment variables of the form *HIO_variable_name*
4) Configuration files (lowest)

Per context or dataset specific values for variables take precedence over globally set values.

1) File Configuration: Configuration files can be used to set variable values globally or within a specific context or dataset. Configuration files are passed as a parameter to the context creation routines (*hio_init_mpi()* and *hio_init_single()*. They can be divided into sections using keywords specified within []'s. The keywords currently recognized by libhio are: global, context:context_name, or dataset:dataset_name. Variable values not within a section, by default, apply globally.

If desired an application can set a prefix for any line in the configuration file that is to be parsed by libhio. This allows the application to add hio specific configuration to an existing file. Any characters appearing after a # character are treated as comments and ignored by libhio unless the # character appears at the beginning of the line and is part of the application specified line prefix (eg #HIO).

F. 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* variable on the dataset object. Valid values for this variable are *auto*, *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.

G. Output Format

When writing datasets to a POSIX file system *libhio* provides support for multiple file formats. The current *libhio* version (1.4.0.0) supports two backend file formats; basic, and file_per_node. Additional file formats may become available in future releases. *libhio* can be configured to use one of three POSIX compatible APIs for reading from and writing to POSIX fileystems. The API used can be specified by setting the *posix_file_api* variable. Valid values for this value are *posix*, *stdio*, and *pposix* which correspond to POSIX (read/write), C streaming (fread/fwrite), and positioned POSIX (pread/pwrite). This section provides high-level details on the output modes and provides some basic advice on when each mode might be more appropriate.

1) Basic Output Mode: The first file mode supported by *libhio* was the basic file mode. When using this mode the files on disc directly map to the elements written as part of the dataset and the file offsets used for reads and writes directly correspond to application element offsets. Within a particular dataset instance this translates to a file per element for datasets using shared offsets and a file per element per node for unique offsets. This mode has the added lowest overhead and is recommended if the application can already make optimal use of the underlying filesystem.

2) File-per-node: We recognized that with a typical checkpoint/restart workload that most output is created defensively to protect against systems failures. This means that most checkpoint/restart output is written but never read. Additionally, checkpoint/restart files are typically opened only in a read-only or write-only mode and no overlapping writes are performed. These realizations lead to the development of an additional output mode aimed at optimizing application writes. When using this mode *libhio* creates a single file file per node for each dataset instance. This optimization is modeled directly after the way *plfs*[2][3] optimizes writes.

When writing element data a stripe aligned portion of the single shared file is reserved using local shared memory for coordination between local MPI processes. The element name, application offset, node id, and file offset are recorded and saved to a file when the dataset is closed. When opening a dataset instance for reading we create a distributed table of this "meta-data" using MPI-3 Remote Memory Access (RMA). This allows relatively quick lookup of data without needing to locally cache all of the meta-data for a dataset instance. This is necessary as the meta-data associated with a dataset instance can get quite large.

There are several advantages to using this output mode. When writing a dataset with unique element offsets the file-per-node offset mode can lead to an overall reduction the the total filesystem meta-data load. For datasets using a shared offset mode it reduces contention. We can, additionally, disable Lustre file locking when utilizing this mode due to the data layout. The downside of this mode is that it creates *libhio* meta-data on the order of the number of writes. This mode is not recommended for applications that perform many small writes to non-contiguous offsets.

IV. Performance Evaluation

This section details the hardware setup and software used to evaluate the performance of *libhio*. We then presents the performance results of running the IOR benchmark using both the file-per-node and basic output modes against *DataWarp* with a file per process and a single shared file.

A. IOR Benchmark

IOR is a benchmark developed at LLNL that executes a synthetic parallel IO workload using various backends. The backends include POSIX and MPI/IO. We pulled version of IOR from git with master hash *aa604c1* and added a backend to support *libhio*. We also added support for setting the *libhio* output mode via the -O IOR command-line option.

B. Experimental Setup

The system used for the IOR benchmarks was Trinity Phase 2. Trinity Phase 2 is a Cray XC-40 located at LANL and operated by Alliance for Computing at Extreme Scale (ACES). The system is comprised of 8909 compute nodes each with a single Intel Knights Landing (KNL) processor with 68 cores and 4 hyper-threads per core. Nodes are connected with the Cray *Aries* network. The system has 234 Cray *DataWarp* service nodes connected directly the the *Aries* network. Each *DataWarp* node contains 2 6 TB Intel P3608 SSDs. Each SSD is connected to the service node with a PCIe x4 interface. More details on the architecture of Trinity and its *DataWarp* can be found in [4]. The total aggregate maximum bandwidth when using all 234 *DataWarp* nodes is 1228.5 GiB/sec.

The software environment used for all runs is as follows. CLE: 6.0 UP03, *libhio* 1.4.0.0, Open MPI git hash *6886c12*, and gcc 6.1.0. All *DataWarp* allocations used were configured in striped mode using all 234 *DataWarp* nodes with a 1 TiB total size. *DataWarp* staging was disabled and the positioned POSIX (pread, pwrite) APIs were used for all benchmarks. We attempted to run *libhio* with Cray MPICH but ran into issues with the current version of their optimized RMA implementation.

All IOR benchmark runs were configured as follows:

- Four readers/writers per node
- 1 MiB block size. -t 1m
- 8 GiB per process / 32 GiB per node. -b 8g
- Ordering inter-file set to 1. -C

C. File Per Process

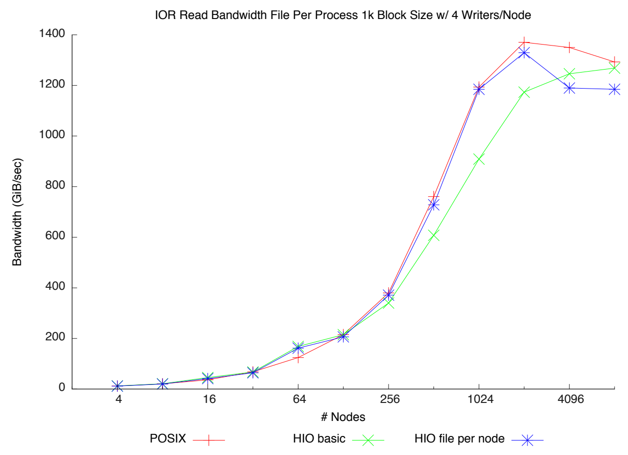
Figures 1(a) and 1(b) show the read and write performance of IOR writing a file per process using POSIX and *libhio*. For these runs the POSIX and *libhio* backends showed similar performance for reads and writes at all node counts. For write both *libhio* in basic mode and POSIX achieved a maximum of 1000 GiB/sec or 81% of the maximum bandwidth at 8192 nodes. The write speed using *libhio* in file-per-node mode achieved 918 GiB/sec or 74% of the maximum bandwidth. The reduction in bandwidth for file-per-node is not unexpected for this case due the way data is layed out in this mode. Read speeds topped out at 1370 GiB/sec, 1173 GiB/sec, and 1329 GiB/sec for POSIX, *libhio* basic, and *libhio* file_per_node respectively. We could not determine why the read speed exceeding the maximum theoretical throughput of the available *DataWarp* nodes.

D. Shared File

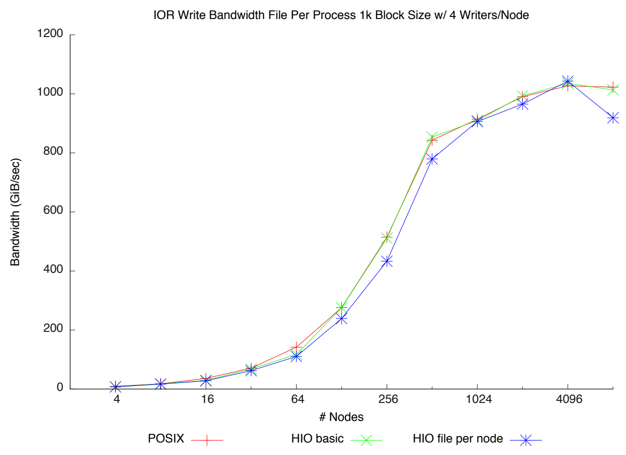
Figures 2(a) and 2(b) show the read and write performance of IOR writing a single shared file using POSIX and *libhio*. When reading or writing a single shared file *libhio* in file_per_node mode achieves higher write speeds than either POSIX or *libhio* in basic mode. The maximum write speeds of 850 GiB/sec, 900 GiB/sec, and 1000 GiB/sec were achieve using POSIX, *libhio* basic, and *libhio* file_per_node. These are 69%, 73%, and 81% of the maximum bandwidth. For writing a single shared file there is a clear advantage to using the file-per-node output node when writing to *DataWarp*. The maximum read speeds were 1132 GiB/sec 1245 GiB/sec, and 1303 GiB/sec. This shows that even though there is overhead in reading the meta-data it does not reduce the read bandwidth. This may not be the case when using unoptimized MPI RMA implementations.

V. Related Work

Many of the ideas and optimizations behind *libhio* are present in other IO libraries and user-space filesystems. *plfs*[2][3] is a user-space filesystem created at LANL that refactors application IO to optimize it for the underlying filesystem. 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.

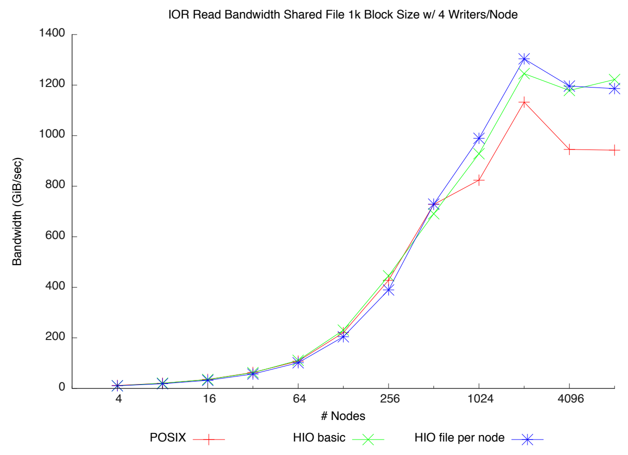


(a) File Per Process Read Bandwidth

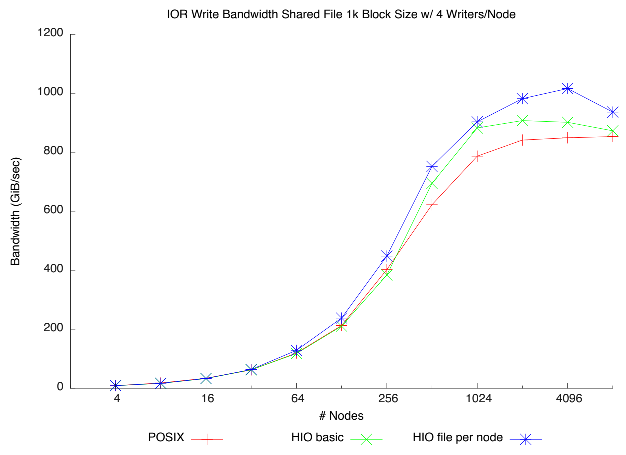


(b) File Per Process Write Bandwidth

Figure 1: Plots showing the read and write bandwidth of IOR accessing a file per rank using the POSIX and *libhio* backends.



(a) Shared File Read Bandwidth



(b) Shared File Write Bandwidth

Figure 2: Plots showing the read and write bandwidth of IOR accessing a single shared file using the POSIX and *libhio* backends.

The API of *libhio* shares many of the same design decisions as those made by ADIOS[5]. *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.

VI. Conclusions and Future Work

In this paper, we presented details on *libhio*, a new IO library for hierarchal storage systems. We detailed the motivations that lead to the development of *libhio* and describe the current design. We showed that using *libhio* with *DataWarp* gives good overall performance under artificial IO workloads driven by the IOR IO benchmark both for writing a single shared file and a file per process. When using the file_per_node optimization in *libhio* $N \rightarrow 1$ workloads showed a 25% increase in write speed and a 10% increase in read speed.

In the future, we plan to investigate additional improvements to *libhio*. We will look at adapting *libhio* to support additional burst-buffer implementations including distributed burst-buffers without a globally visible file space. We intend to investigate the performance of *libhio* with real application workloads on both *DataWarp* and Lustre filesystem. These investigation will likely lead to further optimization within *libhio*. When the two halves of Trinity are connected we will also verify that the performance of *libhio* scales with the combined *DataWarp* filesystem. We may also look at implementing some of the features of *libhio*, such as automatic staging, in ADIOS.

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. Additionally, the authors would like the thank Alfred Torrez at LANL for providing details how to configure the IOR benchmarks. LA-UR-17-23841.

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] 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.

[3] 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.

[4] K. S. Hemmert, M. W. Glass, S. D. Hammond, R. Hoekstra, M. Rajan, S. Dawson, M. Vigil, D. Grunau, J. Lujan, D. Morton *et al.*, "Trinity: Architecture and early experience," 2016.

[5] 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.