Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht

HDF5 VOL Plugin for JULEA

vorgelegt von

Johannes Coym

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Wirtschaftsinformatik
Matrikelnummer:	6693524
Betreuer:	Dr. Michael Kuhn

Hamburg, 2019-03-15

Contents

1 Introduction

The target of this project was to create an HDF5 VOL Plugin for the storage framework JULEA which splits the stored data between multiple storage servers optimized for different types of data. So in the first parts there will be an explanation what exactly JULEA and HDF5 are and why they should be brought together. In the second chapter the planning of the project will be shown, while the third chapter will go into the implementation of the plugin. In the last part there will be a benchmark to compare how the implementation performs compared to the reference HDF5 design.

1.1 JULEA

JULEA is a flexible storage framework for HPC which targets to simplify storage systems in HPC. A key feature of JULEA is the splitting of data and metadata while having easily exchangable backends for data and metadata and even supporting the addition of more backends. Another key feature of JULEA are the multiple included clients with the support to add even more.

The important clients for this project are the distributed object client and the Key Value(KV) Client. The distributed Object Client is a client for larger amounts of data which distributes the data over all available servers. The KV Client is made for metadata and uses pairs of a key and a value which will typically be written to a database.

The config of JULEA then provides the option to specify different backends for objects and KVs.

1.2 HDF5

HDF5 is a file format specialized on large scientific data with a similar structure as a file system. There are 4 important object types in an HDF5 file. The first one is the File itself, which can only contains groups. Groups are like folders which could contain another group but also could contain datasets. A dataset contains a homogenous array of any dimensions which typically would contain larger data. A dataset typically also contains an attribute which consists of small data like metadata attached to the dataset.

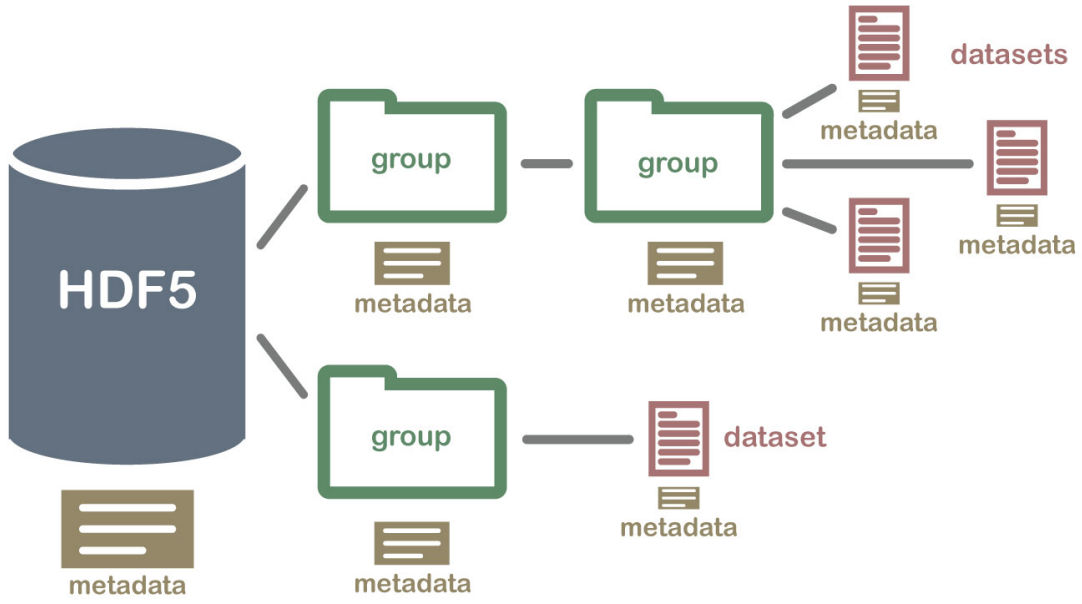


Figure 1.1: Example structure of an HDF5 file, `https://www.neonscience.org/sites/default/files/images/HDF5/hdf5_structure4.jpg`

2 Planning

To combine the strengths of JULEA and HDF5 the plan was to use the HDF5 VOL interface to create an VOL Plugin which would be inculded in JULEA. This plugin should make use of the multiple servers in JULEA to distribute the files so in the end this plugin will save all datasets in the distributed object servers while all the metadata will be saved with Key-Value Pairs.

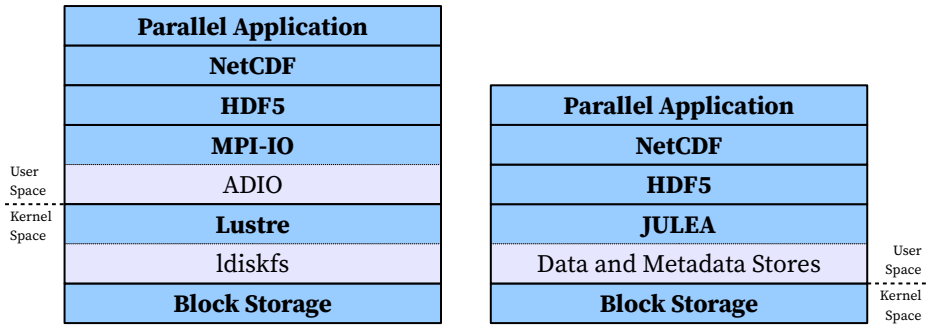


Figure 2.1: I/O Stack of an HDF5 application without and with JULEA

In Figure 1.1 the effect on the I/O Stack of a typical parallel application is shown. Without JULEA the I/O Stack has several more layers which need to be managed and maintained like for example MPI-IO and Lustre, with the latter of both even requiring to run in Kernel space.

The target of JULEA with the VOL Plugin at this point is, to directly take the input from the HDF5 Layer and managing everything to get the data from the Block Storage. So when the plugin is loaded ahead of the HDF5 Commands, it recieves all requests from HDF5 and manages that the datasets will be stored with distributed objects using the POSIX interface for example, while all metadata is stored in Key-Value Pairs using the LevelDB interface for example.

This should make the I/O Stack for parallel applications much easier to manage and possibly even increase the performance of the HDF5 layer since the data is split to several servers.

3 Implementation

For the implementation there are several functions for each of the 4 object types from HDF5 needed, as well as a struct for each of them, which contains all the information and pointers needed. Additionally functions which initialize and terminate the plugin are required, as well as functions returning the plugin type and info and a VOL class containig a list of all the functions provided directly to the VOL interface. At last there are also some functions in the background needed, which help encoding and decoding the types and spaces of datasets and attributes, create a path using the previous path and the name of the current object and several functions to serialize and deserialize the BSON files saved in the Key-Value Pairs.

File and group are the easiest object types since they only require 3 functions each and 1 or 2 strings in the struct to save name and path. Datasets and attributes on the other hand require 3 more functions for read-, write- and get-operations and need larger structs for the pointers to Key-Value Pairs and distributed objects.

Each of the functions from the HDF5 VOL implementation that get implemented (except `file_create` and `file_open`) either provide a pointer to the previous object(`create` and `open`) or provide a pointer to the object they've got to work with(`close`, `get`, `read`, `write`) in the first argument.

3.1 Files

```
1 static void *H5VL_jhdf5_file_create(const char *name,
    ↪ unsigned flags, hid_t fcpl_id, hid_t fapl_id, hid_t
    ↪ dxpl_id, void **req);
2 static void *H5VL_jhdf5_file_open(const char *name,
    ↪ unsigned flags, hid_t fapl_id, hid_t dxpl_id, void
    ↪ **req);
3 static herr_t H5VL_jhdf5_file_close(void *file, hid_t
    ↪ dxpl_id, void **req);
```

Listing 3.1: File operations

For the files only pretty simple create, open and close functions are needed, since the file is only needed to create the path of the objects contained in the file.

```
1 ginfo = (h5julea_fapl_t
    ↪ *)H5VL_jhdf5_fapl_copy(H5Pget_vol_info(fapl_id));
```

Listing 3.2: Copying the file operations

In the create function first a copy of the file operations has to be taken using the `fapl_id`(Listing 3.2). Afterwards memory has to be allocated for the struct and copy the name of the file in the struct, then the struct is returned, so HDF5 provides a pointer to the file everytime it gets referenced.

The procedure in the open function is exactly the same and in the close function all memory previously allocated in create or open is freed using the pointer to the struct that HDF5 provided.

3.2 Groups

```
1 static void *H5VL_jhdf5_group_create(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *name, hid_t
    ↪ gcpl_id, hid_t gapl_id, hid_t dxpl_id, void **req);
2 static void *H5VL_jhdf5_group_open(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *name, hid_t
    ↪ gapl_id, hid_t dxpl_id, void **req);
3 static herr_t H5VL_jhdf5_group_close(void *grp, hid_t
    ↪ dxpl_id, void **req);
```

Listing 3.3: Group operations

Groups need the same functions as files and they do pretty similar things, except they don't need a copy of the file operations. Instead of this they need the location which is another string that is saved in the struct like the name and created by combining the location of the previous element, which will be provided in `void *obj`, and its own name. This string is generated by the `create_path` function[1]. In the function is `loc_params` used to determine if `*obj` is a file or a group which makes the difference if only the name of the previous element or the location is needed.

Just like the file functions, the create and open functions work the same way and the close function will first free both strings and afterwards the struct itself.

[1]See Appendix A

3.3 Datasets

Datasets are far more complicated than files and groups and that's why additionally to the name and location the data size, the distributed object with its distribution and a Key-Value Pair in the struct for the dataset has to be saved. The following chart shows how the data and metadata that requires to be stored is split between the distributed object and the Key-Value Pair.

Data	JDistributedObject *object
Type	JKV *kv
Space	
Data Size	
Distribution	

Table 3.1: Storage configuration for Datasets

```
1 static void *H5VL_jhdf5_dataset_create(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *name, hid_t
    ↪ dcpl_id, hid_t dapl_id, hid_t dxpl_id, void **req);
2 static void *H5VL_jhdf5_dataset_open(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *name, hid_t
    ↪ dapl_id, hid_t dxpl_id, void **req);
3 static herr_t H5VL_jhdf5_dataset_read(void *dset, hid_t
    ↪ mem_type_id, hid_t mem_space_id, hid_t file_space_id,
    ↪ hid_t plist_id, void *buf, void **req);
4 static herr_t H5VL_jhdf5_dataset_get(void *dset,
    ↪ H5VL_dataset_get_t get_type, hid_t dxpl_id, void
    ↪ **req, va_list arguments);
5 static herr_t H5VL_jhdf5_dataset_write(void *dset, hid_t
    ↪ mem_type_id, hid_t mem_space_id, hid_t file_space_id,
    ↪ hid_t plist_id, const void *buf, void **req);
6 static herr_t H5VL_jhdf5_dataset_close(void *dset, hid_t
    ↪ dxpl_id, void **req);
```

Listing 3.4: Dataset operations

With datasets the create function also has to be more complex, since first the type and size of the data that will be stored in the dataset have to be generated. Afterwards the location of the dataset gets generated, the name of the dataset gets copied, the distribution and distributed object get created and they are all stored in the struct. At the end the JKV gets created and type, space, data size and distribution previously generated get saved in it.

The open function of the datasets is not, like in files and groups, identical with the create function, since there is no need to create a new distribution but instead most of the metadata out of the Key-Value Pair has to be loaded and a distributed object which could later be used to access the object on the server has to be created. Close on the other hand is a fairly simple function since it only cleans up everything saved in the struct and frees the memory afterwards.

The new part are the get, read and write functions with the last two of those functions being the simplest since for write it only requires taking the data to write provided in `*buf` and writing it to the distributed object in the struct. For read it works the other way around and first the data is read from the distributed object, then the output is written to `*buf`. The open function is used to provide the type or space that is saved in the JKV and the `get_type` parameter provides which of those should be returned. Then the JKV is read and the type or space in it gets decoded and returned.

3.4 Attributes

Attributes only have 2 Key-Value Pairs to save the data they need, with the data size being directly included in the JKV of the data. Since the attribute also doesn't need to have a distribution because everything is saved in JKVs, the second JKV only consists of type and space, as is shown in the following chart.

Data	JKV
Data Size	*kv
Type	JKV
Space	*ts

Table 3.2: Storage configuration for Attributes

```
1 static void *H5VL_jhdf5_attr_create(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *attr_name,
    ↪ hid_t acpl_id, hid_t aapl_id, hid_t dxpl_id, void
    ↪ **req);
2 static void *H5VL_jhdf5_attr_open(void *obj,
    ↪ H5VL_loc_params_t loc_params, const char *attr_name,
    ↪ hid_t aapl_id, hid_t dxpl_id, void **req);
3 static herr_t H5VL_jhdf5_attr_read(void *attr, hid_t
    ↪ dtype_id, void *buf, hid_t dxpl_id, void **req);
4 static herr_t H5VL_jhdf5_attr_write(void *attr, hid_t
    ↪ dtype_id, const void *buf, hid_t dxpl_id, void **req);
5 static herr_t H5VL_jhdf5_attr_get(void *obj,
    ↪ H5VL_attr_get_t get_type, hid_t dxpl_id, void **req,
    ↪ va_list arguments);
6 static herr_t H5VL_jhdf5_attr_close(void *attr, hid_t
    ↪ dxpl_id, void **req);
```

Listing 3.5: Attribute operations

Creating an attribute is pretty similar, but also a little different, from creating a dataset, since it also starts with getting(using the `acpl_id`) and encoding type and space of the attribute, but then it creates the two JKVs needed and saves type and space to one of them. The other one stays empty for the moment since it only gets its key in the create function, the value comes in the write function.

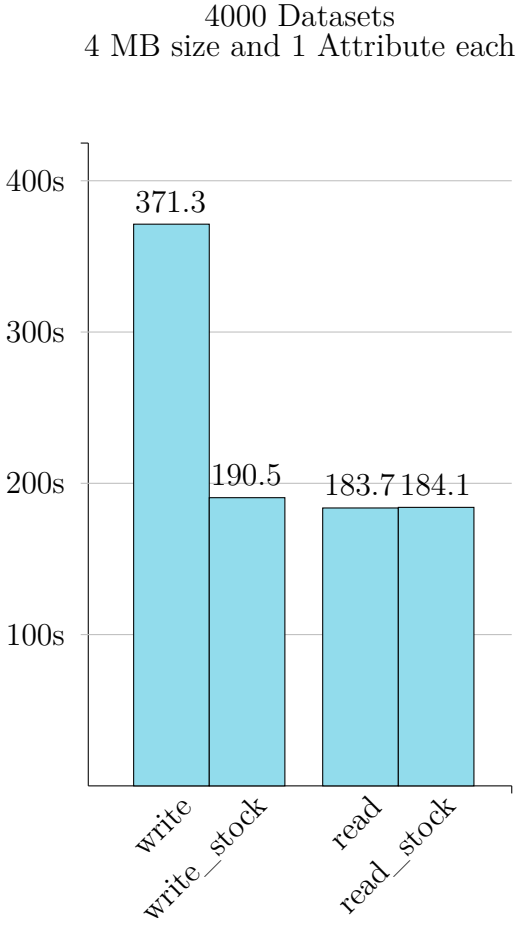
Like with the dataset, the open function is a little different since it creates both JKVs but only reads the data size from the main JKV. Close does the same thing it does with other object and just cleans everything up and frees the memory afterwards.

The write function first serializes the data provided in the `*buf` parameter into a BSON, then puts that BSON into the JKV. Read then simply inverts the operation again, gets the data from the JKV, deserializes the BSON and then returns it into `*buf`. The get function works exactly the same way as with the datasets.

4 Benchmark

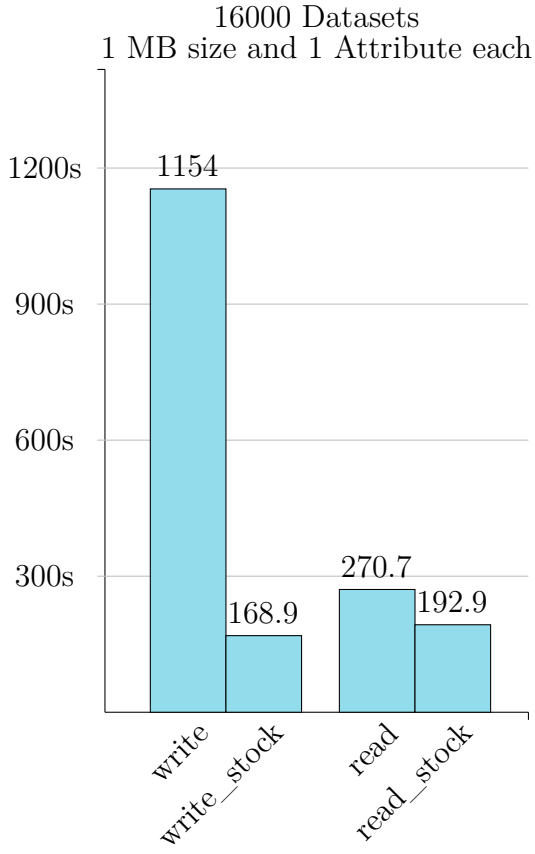
The benchmarks were run on a single node with a single hard drive, so this is naturally pretty limited by the speed of the hard drive. They were also configured to write about 16 GB of data each, so the data couldn't be cached since the node only has 12 GB of RAM. The benchmarks were then run with different sizes of datasets to simulate the performance with different types of data. Each benchmark size was run three times and the average of the runs was taken.

The first benchmark is the medium sized benchmark which consists of 4000 datasets with a size of 4 MB each and each dataset also has an attribute where a little amount of metadata is stored.



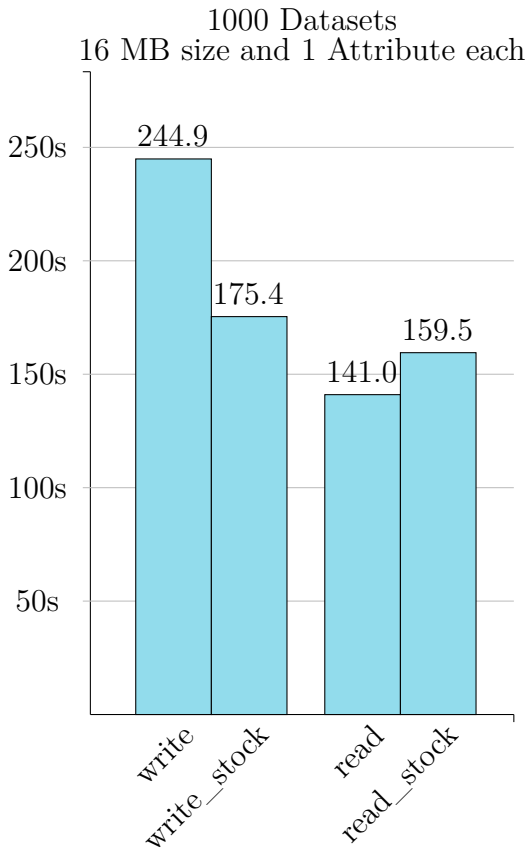
As the results show the plugin has a quite large overhead in writing compared to the stock implementation of HDF5, which is to be expected when comparing the plugin to writing directly to the drive. Although in reads of this size the overhead isn't noticable at all, with the plugin being, within margin of error, as fast as the stock implementation.

The second benchmark are the small sized datasets, which are only 1 MB each, but the HDF5 file consists of 16000 datasets with an attribute for each of them.



With the small sized benchmark the overhead on the writes with the plugin are much larger than on the medium sized, since the overhead multiplies with the amount of datasets. In the read benchmark the small sized datasets also impact the performance on the plugin since the overhead is more noticable with more datasets, but it is not nearly as bad as with the write.

The last benchmark conatins 1000 large datasets with a size of 16 MB and again one attribute each.



In the large sized benchmark the write overhead of the plugin is the smallest since there are only 1000 datasets, but it is still noticable. The read performance on the other hand is even better than with the stock implementation and this is where the advantage of splitting data and metadata already shows compared to just writing straight to the hard drive.

In conclusion the plugin always creates some overhead while writing, although it gets significantly less overhead the larger the data is. Reading on the other hand only has some overhead to the stock implementation while using small datasets and also improves with the size of the datasets to the point where it is even faster than the stock implementation.

5 Appendix A: Create Path Function

```
1  char* create_path(const char* name, char* prev_location) {
2      static const char* sep = "/";
3      char* res = NULL;
4
5      if (NULL != prev_location && NULL != name) {
6          res = (char*) malloc(strlen(prev_location) +
               ↪ strlen(sep) + strlen(name) + 1);
7          strcpy(res, prev_location);
8          strcat(res, sep);
9          strcat(res, name);
10     }
11     else {
12         ERRORMSG("Couldn't create path");
13     }
14     return res;
15 }
```

Listing 5.1: create_path

List of Figures

List of Listings

List of Tables