Projekt Parallelrechnerevaluation
HDF5 VOL Plugin

Paul Hölzen 6673477
Olga Kravtsova 6511760

betreut von Jakob Lüttgau und Eugen Betke

24. Oktober 2016

Inhaltsverzeichnis

1 Einleitung

1.1 Präambel

HDF5 stellt grundsätzlich ein Datenmodell vor, das komplexe Daten effizient speichert und verwaltet. Die I/O-Zugriffsmuster bei Programmen, die das HDF5 Datenformat in Anspruch nehmen, sind allerdings nicht immer optimal. Bei den zufälligen Zugriffen können es entweder relativ lange I/O-Paths zu SSD oder generell längere Zugriffszeiten auf HDD sein, die Leistung beeinträchtigen. Das kann bei großen Datenmengen zu erheblichem Zeitverlust führen. Es wird deswegen ein Weg gesucht die Anzahl der Zugriffe auf eine Festplatte möglichst zu reduzieren und somit die Leistungskapazität zu steigern.

Eine mögliche Lösung für relativ kleine Aufgaben und einen großen Arbeitsspeicher wäre die Daten in den Arbeitsspeicher zu verlegen und somit Festplattenzugriffe komplett zu vermeiden. Das Ziel des Projekts war ein Plugin für die Virtual Object Layer (VOL) der HDF5 zu entwickeln, die Daten in den Arbeitsspeicher lädt, um preliminare Vorabmessungen zu machen, die einen solchen Ansatz prüfen. Im Projekt wurden nur APIs implementiert, die für diese Vorabmessungen notwendig waren, nämlich H5F, H5G und H5D (siehe Abschnitte 3.1, 3.2 und 3.3). Dabei wurden nur wesentliche Funktionen implementiert. Zum Testen von Schreib- und Lesevorgängen wurde der `HF5_NATIVE_INT` Datatyp implementiert, der eine Menge ganzer Zahlen umfasst, ähnlich dem C-Typen `int`. Es kann dabei nur das komplette Dataset gelesen oder geschrieben werden `H5S_ALL`.

Für einen Leistungsvergleich von HDF5 und HDF5 mit VOL Plugin wird ein anschließender Performancetest (Benchmark) durchgefürt, dessen Resultate im 4. Kapitel dokumentiert sind. In diesem Benchmark werden die Laufzeiten der Funktionen bei Aufruf durch die Standard HDF5 Implementierung und durch das HDF5-VOL-Plugin auf SSD gemessen. Außerdem werden für HDF5 Tests auf SSD und HDD durchgeführt.

1.2 HDF5

HDF5 steht für Hierarchical Data Format Version 5 und ist ein von der HDF Group entwickeltes Dateiformat [3]. Das Format verwendet innerhalb der Datei eine baumartige Struktur, die aus Groups und Datasets besteht. Dabei sind die Datasets Objekte, die Dateiinhalt oder Metadaten enthalten und die Endknoten des gerichteten Strukturgraphen darstellen und Groups, die entweder andere Groups oder Datasets enthalten können sind die Verzweigungen. Dadurch entsteht ein Strukturgraph wie in Abbildung 2.1. Es darf per Definition durch die HDF-Group auch Zyklen in diesem Strukturgraphen geben.
Jede HDF5 Datei (File) enthält einen solchen Graphen aus mindestens einer Gruppe, der sogenannten Root-Gruppe, in der Abbildung “/”.

1.3 VOL

Die VOL, kurz für Virtual Object Layer, ist eine Ebene, die API Calls von den oberen Ebenen abfängt und bearbeitet [2]. Hierbei kann durch verschiedene Implementationen von VOL Plugins bestimmt werden, wie die Datenobjekte im Speicher verwaltet werden. Um das umzusetzen müssen also die API-Calls von HDF in unserem Plugin abgefangen werden und an sogenannte Callback Funktionen weitergegeben werden, die dann entsprechende Operationen mit den Resourcen durchführen.

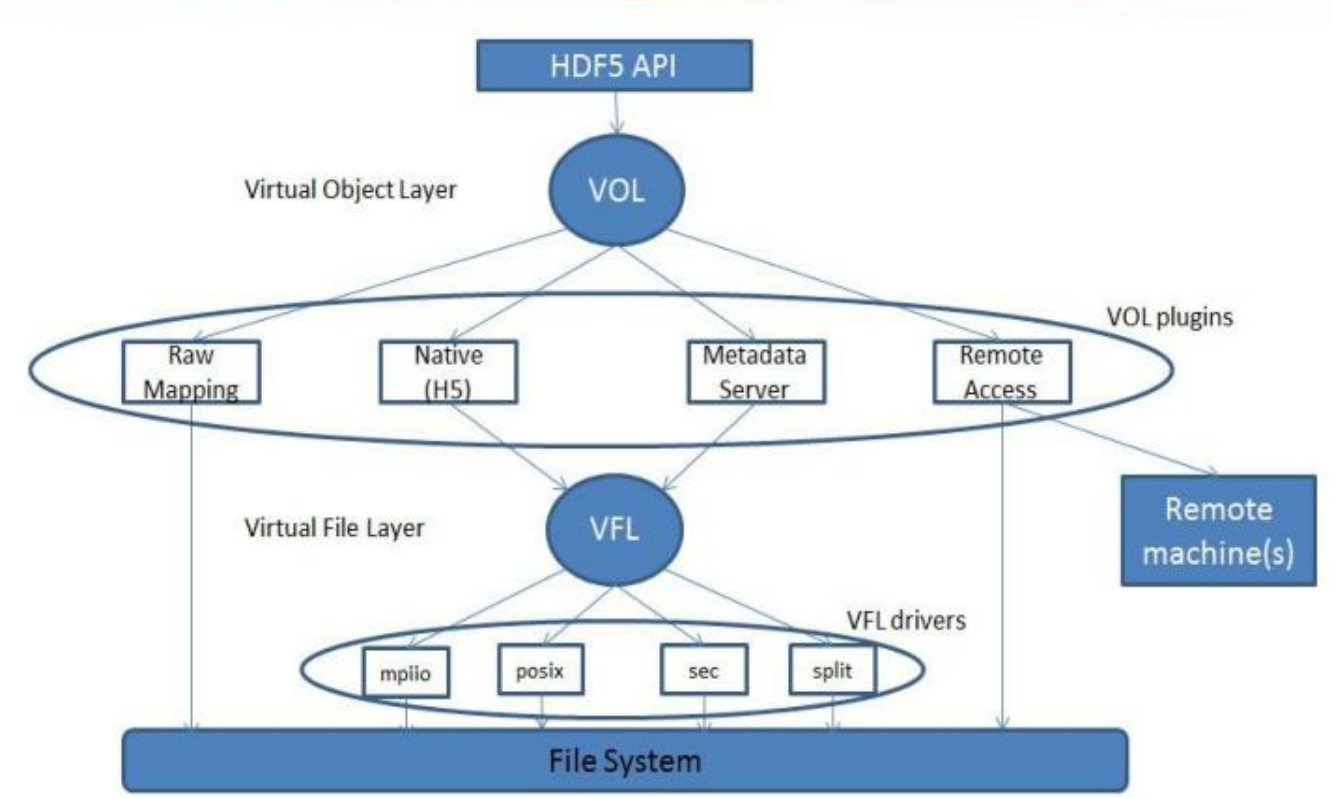

Abbildung 1.1: VOL Struktur [2]

2 Planung

In diesem Kapitel fassen wir die Überlegungen zusammen, die der eigentlichen Entwicklung vorangegangen sind.

2.1 Grundlage

Als Ausgangspunkt für die Entwicklung eines solchen Plugins stand uns das Gerüst eines C Projekts zur Verfügung. In diesem mussten noch die geeigneten Funktionen und Typen definiert werden um HDF5 Objekte erstellen, modifizieren und löschen zu können.

Die Hauptaufgaben sind dabei:

- H5F - File
  - Dateien mit einem Namen und einer Root-Group erstellen können
  - Dateien öffnen und schließen können
- H5G - Group
  - Gruppen mit einem Namen und einer Sammlung von Untergruppen in einer Parent-Gruppe erstellen können
  - Informationen über die Gruppe zurückgeben
  - Gruppen öffnen und schließen können
- H5D - Dataset
  - Datasets mit einem Namen in einer Gruppe erstellen können
  - Informationen über das Dataset zurückgeben
  - Datasets öffnen und schließen können
  - Daten in Datasets schreiben und lesen können
- Strukturen um mit diesen Objekten arbeiten zu können
  - C Typen für die drei oben genannten Objekte
  - Einen Typen Object als “Superclass” von Groups und Datasets

Dabei ist jedes dieser Elemente durch eine eigene C Quelle definiert in der die angebotenen Funktionen definiert werden. Zum Beispiel gibt es die Datei `m-group.c` in der Gruppenspezifische Funktionen implementiert sind, wie unter Anderem `memvol_group_create`. Außerdem gibt es noch einen C Header, in dem mittels C Struct die Datentypen und gegebenenfalls Enumeratoren definiert sind. Dieser wird im ganzen Projekt eingebunden und verwendet.

2.2 Design

Um die Datenstruktur eines HDF Dateibaumes umzusetzen, halten wir uns an das Schema wie es auf Abbildung 2.1 zu sehen ist. Eine Datei hat also eine Root-Gruppe '/' und diese, sowie jede andere Gruppe besitzt eine Liste von Untergruppen. Um das Plugin möglichst effizient und schnell zu machen, haben wir uns für den GLib Datentyp `GHashTable` entschieden um die untergeordneten Objekte zu referenzieren. Die Zugriffszeiten sind hierbei unter Verwendung der Gruppennamen als Hashes deutlich schneller als bei sortierten oder unsortierten Listen oder Arrays.

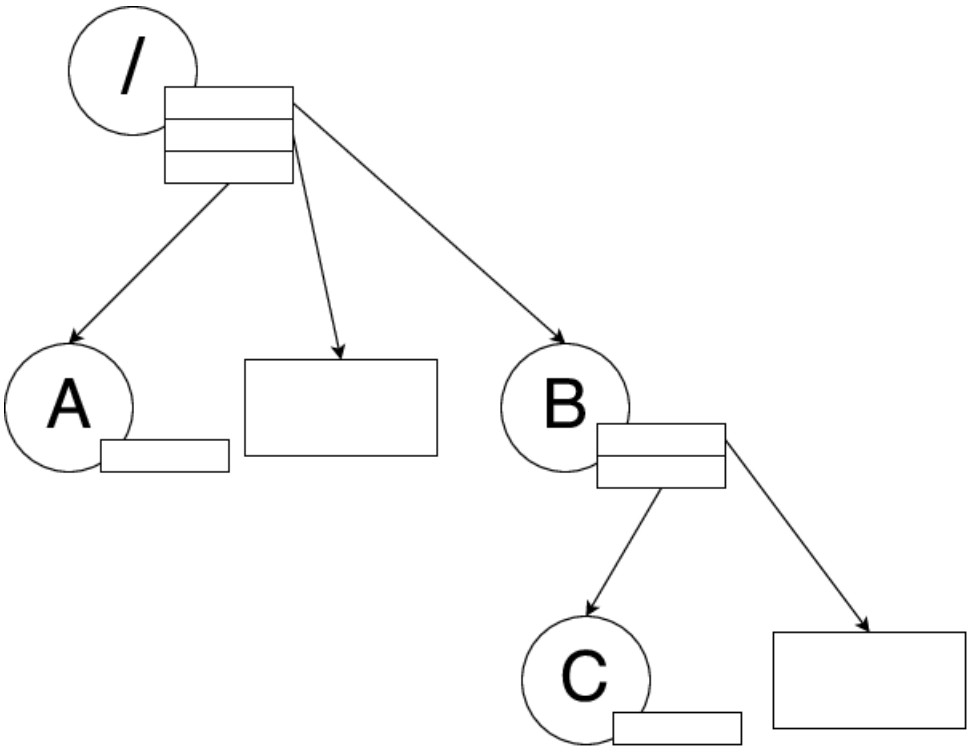

Abbildung 2.1: HDF Strukturgraph

3 Entwicklung

Es hat sich gezeigt, dass eine vollständige Implementation aller Funktionen eines HDF VOL Plugins den Rahmen des Projektes sprengen würde, sodass wir uns auf die Features beschränkt haben, die benötigt werden um am Ende einen Performancetest der Lese- und Schreibzugriffe durchzuführen.

Unser Entwicklungsprozess des Plugins lässt sich in drei grobe Phasen unterteilen: Implementation von Files und Groups als Grundgerüst für die Datei, Implementation von Datasets um Dateiinhalte speichern zu können und letztendlich die Implementation der restlichen Funktionalität, die für die anschließenden Benchmarks benötigt wird.

3.1 File

Für die Implementation der HDF-File haben wir zunächst im `memvol-internal.h` Header ein Struct `memvol_file_t` per typedef definiert. Als Member haben wir die benötigten Attribute eines File-Objektes hinzugefügt: Einen Character-Array `name` und die Root-Gruppe `memvol_group_t* root_group`.

```
1 typedef struct {
2     memvol_group_t* root_group;
3     char* name;
4 } memvol_file_t;
```

Ist nun der Header per `#include` eingebunden, können wir den Struct-Namen als Typ zum Definieren oder Casten von Variablen verwenden. Des weiteren haben wir, ebenfalls im `memvol-internal.h` Header, den Typ `memvol_object_t` als struct definiert, der als Member einen void-Pointer erhält in dem Pointer auf entweder Groups oder Datasets gehalten werden, sowie ein Feld in dem festgehalten wird, welchen Typ das Objekt hat auf das der Pointer zeigt. Wir haben der Übersichtlichkeit halber für dieses Feld einen Enumerator `memvol_object_type` definiert, der die Werte `GROUP_T` und `DATASET_T` umfasst.

```
1 typedef enum {
2     GROUP_T,
3     DATASET_T
4 } memvol_object_type;
5
6 typedef struct {
7     memvol_object_type type;
8     void* subclass;
9 } memvol_object_t;
```

Die Callback Funktionen für die File sind in `m-file.c` definiert.
Sie umfassen `memvol_file_create, memvol_file_get, memvol_file_open` und `memvol_file_close`.

3.1.1 memvol_file_create

Soll eine HDF-File erstellt werden, wird als Rückgabewert ein void-Pointer erwartet. Anstatt hier einen Pointer auf ein Objekt des im Header definierten `memvol_file_t` Typs zurückzugeben, übergeben wir einen `memvol_object_t` Pointer, in dem die Root-Group der erzeugten Datei als `subclass` Member abgespeichert ist. So kann der Pointer aus der Rückgabe der Create-Methode im Anwendungsfall verwendet werden um Gruppen direkt in der Root-Group zu erzeugen (siehe 3.2.1).
Um ein Objekt des `memvol_file_t` Structs zu erzeugen muss zunächst Speicher für das Objekt und alle seine Member allociert werden. Danach erst können Werte zugewiesen und Variablen initialisiert werden. So wird der als Parameter übergebene Name dem entsprechenden Member zugewiesen. Die Root-Group wird ebenfalls initialisiert (siehe 3.2). Im Sonderfall der Root-Gruppe einer Datei wird diese sich selbst als Child hinzugefügt.
Zusätzlich allocieren wir Speicher für den eigentlichen Rückgabewert, den `memvol_object_t` Pointer und initialisieren dessen Typ-Member mit dem Wert `GROUP_T`, da wir ja nur auf die Root-Group verweisen und nicht auf das ganze File-Objekt. Dementsprechend weisen wir dem `subclass` Member den Pointer auf die Root-Group zu und geben einen Pointer auf dieses Object-Objekt zurück.
Die erstellte Datei wird außerdem noch in einer `GHashTable file_table` abgelegt, um eine Übersicht aller bereits erstellten Dateien zu haben.
Diese können dann mittels `g_hash_table_lookup` über ihren Namen gesucht und zugegriffen werden.

3.1.2 memvol_file_open

Die Methode zum Öffnen einer Datei durchsucht die Tabelle aller existierenden Files nach dem Namen und gibt einen Pointer auf die Datei zurück.

```
1 ||memvol_file_t* ret = g_hash_table_lookup(file_table, name);
```

Wurde die Datei noch nicht erstellt, und konnte somit nicht in der `file_table` gefunden werden, gibt die Methode eine Fehlermeldung aus.

3.1.3 memvol_file_get

Die get-Methode der File gibt verschiedene Informationen über die Datei zurück. Dabei kann über den Parameter `get_type` bestimmt werden, welche Rückgabe gebraucht wird. Er hat folgende definierte Werte:

- `H5VL_FILE_GET_FAPL` - gibt die *File Access Property List* zurück
- `H5VL_FILE_GET_FCPL` - gibt die *File Creation Property List* zurück
- `H5VL_FILE_GET_INTENT` - gibt den Status der Read/Write Permission-Flag der Datei zurück
- `H5VL_FILE_GET_NAME` - gibt den Namen der Datei zurück
- `H5VL_FILE_GET_OBJ_COUNT` - gibt die Anzahl der Objekte in der Datei zurück
- `H5VL_FILE_GET_OBJ_IDS` - gibt eine Liste von Pointern auf die Objekte der Datei zurück
- `H5VL_OBJECT_GET_FILE` - gibt einen Pointer auf die Datei slebst zurück

Diese Funktion ist in unserem Plugin nicht vollständig implementiert. Wie am Anfang des Kapitels bereits gesagt, wurden nur die Funktionen umgesetzt, die am Ende im Performancetest benötigt werden.

3.1.4 memvol_file_close

Beim Schließen einer Datei haben wir uns entschieden keine Operationen vorzunehmen. Zuerst wollten wir an dieser Stelle den gesamten Speicher, der von den Strukturen der File belegt wird frei zu machen indem wir `free` und `g_hash_table_destroy` verwenden. Dies ist aber in mehreren Punkten nicht sinnvoll und sogar problematisch. So müssten zum Beispiel alle Daten der Datei irgendwo gesichert werden, damit diese nicht vollständig verloren gehen. Außerdem müsste neuer Speicher alloziert werden sobald die Datei wieder geöffnet werden soll. Dies alles ist unpraktikabel und so wird in unserem Fall einfach alles im Speicher gelassen und nur ein gültiger Rückgabewert zurückgegeben.

3.2 Group

Gruppen haben ebenfalls ihren eigenen Datentyp `memvol_group_t` im `memvol-internal.h` Header. Dieser besitzt Member für den Namen und die Menge an Untergruppen, die in einer `GHashTable` festgehalten werden.

```
1 typedef struct {
2     GHashTable* children;
3     char* name;
4 } memvol_group_t;
```

Die Callback Funktionen für Groups sind in `m-group.c` definiert.
Sie umfassen `memvol_group_create, memvol_group_get, memvol_group_open` und `memvol_group_close`.

3.2.1 memvol_group_create

Wie schon bei der File muss auch für die Erstellung einer Group zuerst Speicher für das Struct und alle seine Member allociert und anschließend initialisiert werden. Der Name der Gruppe wird, genau wie bei der Datei als Parameter übergeben und kann dem Member `name` nach der Allocation des Speichers einfach zugewiesen werden.
Für den `children` Member muss nicht erst Speicher allociert werden, da die GLib Funktion `g_hash_table_new` dies automatisch übernimmt. Da wir den Gruppennamen als Hash verwenden wollen erstellen wir die Hashtable mit Strings als Suchschlüsseln:

```
1 group->children = g_hash_table_new(g_str_hash, g_str_equal);
```

Wenn eine Gruppe erstellt wird, wird im Funktionsaufruf der Pointer der Parent-Group übergeben, in der sie erzeugt werden soll. Um diese Beziehung umzusetzen muss die neue Gruppe ihrer Parent-Group als Untergruppe hinzugefügt werden. Dazu Casten wir den übergebenen `memvol_object_t` Pointer auf den benötigten Typen `memvol_group_t`, wenn der `type` Member vom Typ `GROUP_T` ist. So erhalten wir einen Pointer auf die Parent-Gruppe und können nun der `GHashTable children` die neue Gruppe durch den Aufruf

```
1 g_hash_table_insert(parent_group->children, strdup(name), object);
```

hinzufügen. Dabei ist `object` ein Pointer vom Typ `memvol_object_t` dessen `type` Member `GROUP_T` ist und der als `subclass` die neue Gruppe hält. Dieser ist auch der Rückgabewert am Ende der Methode.

3.2.2 memvol_group_get

Die Methode `memvol_group_get` hat zwei Modi, die vom übergebenen Parameter `get_type` bestimmt werden:

- `H5VL_GROUP_GET_GCPL` - gibt die *Group Creation Property List* zurück
- `H5VL_GROUP_GET_INFO` - gibt Informationen über die Gruppe zurück

Zum Übergeben der Argumente verwendet die Methode das Macro `va_list`. Aus dieser können mittels eines weiteren Macros `va_args` Werte extrahiert werden.

Wie auch die get-Funktion der File wurde auch diese nicht vollständig implementiert. Die Rückgabewerte, sowie die Unterscheidung der genannten Fälle sind umgesetzt, nicht jedoch die Funktionalität der Funktion. Dies ist für den abschließenden Benchmark Test jedoch unerheblich.

3.2.3 memvol_group_open

Die Methode zum Öffnen einer Gruppe wird, ähnlich wie die zum Erstellen, mit einem Pointer auf die Parent-Group als Parameter aufgerufen und gibt anschließend einen Object-Pointer zurück, der auf die zu öffnende Gruppe verweist. Dazu wird die `GHashTable` von Untergruppen der Parent-Group nach dem Namen der zu öffnenden Gruppe durchsucht:

```
1 memvol_object_t* ret = g_hash_table_lookup(parent->children, name);
```

Konnte kein Eintrag mit dem Hash `name` gefunden werden, wird eine Fehlermeldung ausgegeben, ansonsten der Pointer auf das gefundene `memvol_object_t` Objekt.

3.2.4 memvol_group_close

Beim Schließen einer Gruppe wird zunächst über die `GHashTable` mit Kindern iteriert und der Speicher jeder Instanz eines Datasets frei gemacht (siehe auch 3.3.6). Sind diese somit aus der Tabelle gelöscht, werden die restlichen enthaltenen Werte mit `g_hash_table_destroy` ebenfalls aus dem Speicher entfernt und die anderen Member des Group-Structs werden gelöscht. Zum Schluss werden alle Member der Gruppe sowie die Variable, die den Pointer auf diese hielt, auf `NULL` gesetzt.

3.3 Dataset

Datasets sind als das Struct `memvol_dataset_t` im Header `memvol-internal.h` definiert. Sie besitzen Member für einen Namen, den Datentyp des Datenbuffers, den Dataspace, einen Pointer auf Data selbst, einen Pointer auf die Gruppe in der es erstellt wurde, sowie eine Link Creation Property List.

```
1 typedef struct {
2     char* name;
3     hid_t datatype;
4     hid_t dataspace;
5     hid_t lcpl;
6     void* data;
7     memvol_object_t* loc_group;
8 } memvol_dataset_t;
```

Dataspace enthält Informationen über die Anzahl von Dimensionen und deren Größen. Die Link Creation Property List enthält Information um Verbindungen zu Gruppen zu erstellen. In dieser Implementation wird `loc_group` als Strukturelement verwendet. Dies kann weggelassen werden, wenn Links realisiert werden. Eine Realisierung wie hier ergibt keinen wesentlichen Laufzeitunterschied zur Datasetstruktur ohne `loc_group`, erlaubt aber die Aktualisierung von Daten in HashTabellen.

Die Callback Funktionen für Datasets sind in `m-dataset.c` definiert. Sie umfassen `memvol_dataset_create`, `memvol_dataset_get`, `memvol_dataset_read`, `memvol_dataset_write`, `memvol_dataset_open` und `memvol_dataset_close`.

3.3.1 memvol_dataset_create

Analog zu Files und Gruppen allocieren wir zuerst Speicher für unser neues Dataset. Bei der Erzeugung des Datasets werden Identifikatoren von Datentyp, Datenspace und Link Creation Property List aus Property Listen extrahiert und für weitere Verwendung unter `datatype`, `dataspace` und `lcpl` gespeichert. Location group wird in `loc_group` gespeichert. Den Namen des neuen Datensets kopieren wir aus dem Eingabeparameter in den `name` Member des Dataset Structs. Der Pointer auf Data wird zuerst auf `NULL` gesetzt. Speicher für die Daten wird beim ersten Schreibvorgang allociert. Ein neues Datenset wird in die Hashtabelle von der Group in der es erzeugt wurde hinzugefügt.

```
1 g_hash_table_insert(parent_group->children, strdup(name), dset_object);
```

Die Methode gibt `dset_object` als Pointer auf `memvol_object_t` zurück, dessen `type` Member auf `DATASET_T` gesetzt ist und einen Pointer auf die Instanz der `memvol_dataset_t` Struktur in `subclass` enthält.

3.3.2 memvol_dataset_get

Wie auch schon bei der get-Methode der File oder Group wird auch beim Dataset in einem Parameter ein Typ übergeben, der bestimmt, welche Arten von Informationen zurückgegeben werden:

- `H5VL_DATASET_GET_DAPL` - gibt die *Dataset Access Property List* zurück
- `H5VL_DATASET_GET_DCPL` - gibt die *Dataset Creation Property List* zurück
- `H5VL_DATASET_GET_OFFSET` - gibt das Offset des Dataset zurück
- `H5VL_DATASET_GET_SPACE` - gibt den Dataspace zurück
- `H5VL_DATASET_GET_SPACE_STATUS` - gibt den Status des Dataspace zurück
- `H5VL_DATASET_GET_STORAGE_SIZE` - gibt die Größe der Daten zurück
- `H5VL_DATASET_GET_TYPE` - gibt den Datentyp zurück in dem Daten abgespeichert sind

Auch diese Methode verwendet das Macro `va_list` um die Parameter für die verschiedenen Modi der Funktion zu übergeben.

3.3.3 memvol_dataset_read

Die Methode `memvol_dataset_read` liest Data aus dem Container in der Dataset-Struktur in einen Buffer. Dafür muss sicher gestellt werden, dass das Dataset nicht die Größe des Buffers übersteigt, um eventuelle Sicherheitslücken zu vermeiden. Wie auch beim `memvol_dataset_write` müssen die Daten von der gleichen Klasse von Datatype sein. Es wurde der Lesevorgang nur für den `H5T_NATIVE_INT` Datatype implementiert, der wie der C-Typ `int` einige ganze Zahlen umfasst. Der Dataspace-Identifikator des Containers und des Buffers sollen `H5S_ALL` sein, was bedeutet, dass der komplette Buffer und das komplette Dataset im Schreibvorgang einbezogen sind.

```
1 n_points = H5Sget_simple_extent_npoints(dataset->dataspace);
```

Die Daten werden aus `dataset->data` in den Buffer kopiert.

```
1 memcpy(buf, dataset->data, n_points * size);
```

3.3.4 memvol_dataset_write

Wie bereits unter 3.3.1 erwähnt, wird erst beim ersten Schreiben auf ein Datenset der benötigte Speicher allociert. Die Größe dieses Speicherplatzes berechnet sich aus der Anzahl von Elementen im Data-Array und deren Größe:

```
1 /*size of datatype in dataset*/
2 size = H5Tget_size(dataset->datatype);
3
4 /*number of points in dataset*/
5 n_points = H5Sget_simple_extent_npoints(dataset->dataspace);
6
7 if(dataset->data == NULL) {
8    dataset->data =  malloc(n_points * size);
9 }
```

Die Datentypen müssen zur gleichen Klasse gehören, was mit Hilfe von assert geprüft wird. Im Projekt wurde nur eine Schreibmethode für das Lesen des ganzen Datasets aus dem übergebenen Buffer ins Dataset implementiert. Wie bei `memvol_dataset_read` sollen die Daten vom Typ `H5T_NATIVE_INT` sein.

Am Ende des Schreibvorgangs wird der Eintrag in der Hash-Tabelle der Parent-Group aktualisiert.

```
1 memvol_object_t* parent = dataset->loc_group;
2 memvol_group_t* parent_group = (memvol_group_t*)parent->subclass;
3 g_hash_table_insert(parent_group->children, strdup(dataset->name),
     object);  // insertion of the actual dataset in the table
```

3.3.5 memvol_dataset_open

Zum Öffnen eines Datasets bekommt die Methode einen Pointer auf die Gruppe, in der das Dataset gesucht werden soll. Es wird mit `g_hash_table_lookup` nach dem Namen des Datasets in der `GHashTable` der Group gesucht und ein Pointer auf das Ergebnis zurückgegeben.

3.3.6 memvol_dataset_close

Auch diese Methode sollte die Resourcen befreien, was aber hier aus konzeptuellen Gründen nicht möglich ist. Speicher wird erst befreit, wenn die Parent-Grouppe des Datasets geschlossen wird. Dafür werden alle Einträge der `GHashTable` durchsucht und falls `memvol_object_t` vom Typ `DATASET_T` ist, werden die einzelnen Member der `memvol_dataset_t` Struktur mit der `free` Methode befreit und auf `NULL` gesetzt. Die Funktionalität von `memvol_dataset_close` wird an `memvol_group_close` übergeben.

4 Benchmark

4.1 Aufbau

Nachdem vorangegangene Modelle und Konzepte Implementiert waren, haben wir einen Performance Test in Form einer Benchmark durchgeführt. Dabei wird die Laufzeit der einzelnen Operationen wie zum Beispiel das Erstellen einer Datei gemessen. Interessant sind für uns hauptsächlich jene Laufzeiten, die mit Speicherzugriffen verbunden sind, wie das Schreiben oder Lesen von Daten eines Datasets. Diese Laufzeiten wollen wir dann mit denen der Standard HDF5 Implementation vergleichen und erwarten eine signifikante Steigerung, da die Zugriffszeiten auf den RAM Speicher, den wir in diesem Projekt verwenden, wesentlich niedriger sein sollten als die auf gewöhnlichen HDD oder sogar SSD Speicher.

Für die Messung der tatsächlichen Laufzeit der Anweisungen haben wir die `time.h` Header der C-Standardbibliothek verwendet. Diese bieten die Funktion `clock_gettime(CLOCK_MONOTONIC, t)`, mit der der aktuelle "Zeitpunkt" gespeichert werden kann. Dabei wird sowohl ein Wert in Sekunden als auch in Nanosekunden gespeichert. Wir haben jeweils einmal vor und einmal nach der Ausführung des zu messenden Codes diesen Zeitstempel festgehalten und anschließend die Differenz der Nanosekunden-Werte berechnet.

```
1  struct timespec begin;
2  struct timespec end;
3  unsigned long delta_t;
4
5  clock_gettime(CLOCK_MONOTONIC, &begin);
6
7  /* zu messender Code */
8
9  clock_gettime(CLOCK_MONOTONIC, &end);
10 delta_t = end.tv_nsec - begin.tv_nsec;
```

Die Leistungstests wurden durchgeführt auf zwei verschiedenen Systemen um das Verhalten auf unterschiedlicher Hardware beobachten zu können. In den Grafiken mit *HDD* beschriftet sind Tests auf einem Tuxedo Laptop mit folgenden Spezifikationen:

RAM:	7,7 GiB
CPU:	Intel(R) Core(TM) i5-4210U CPU 1.70GHz x 4
HDD-Speicher	

Die anderen mit *SSD* beschrifteten Tests wurden auf einem AOKHOME Laptop mit den folgenden Spezifikationen ausgeführt:

RAM: 7,7 GiB
CPU: Intel(R) Core(TM) i7-5500U CPU 2.40GHz x 2
SSD-Speicher

Beide Systeme liefen für die Tests auf 64-bit Ubuntu 16.04 LTS.

Es wird außerdem unterschieden zwischen der Standardimplementation von HDF und unserem Plugin. Diese Unterscheidung wird durch *standard* und *memvol* gekennzeichnet.

Um ein möglichst breites Spektrum von Testwerten zu erhalten haben wir die Tests mit Datasets von variierender Größe durchgeführt. In den Grafiken sind die verschiedenen Größen *10x10, 50x50* und *100x100* aufgeführt, was jeweils die Dimensionen des Dataspace darstellt.

4.2 Durchführung

Im Benchmarktest haben wir in jedem Durchlauf für die verschieden großen Datasets jeweils eine neue File erstellt, in deren Root-Gruppe ein leeres Dataset erstellt werden. Dabei hat es den Datatype `H5T_NATIVE_INT`, die restlichen Parameter sind auf den Standardwert `H5P_DEFAULT` gesetzt. Anschließend wird ein zuvor mit aufsteigenden Integern gefülltes zweidimensionales Array in das Dataset geschrieben. Dabei wird, wie in Sektion 4.1 beschrieben, die Laufzeit gemessen. Die Messwerte für die Schreibvorgänge auf den verschieden großen Datasets sind in den Abbildungen 4.1, 4.2 und 4.3 zu sehen.
Nachdem die Daten geschrieben wurden, lesen wir sie im nächsten Schritt des Tests aus. Auch diese Funktion wird mit der gleichen Methode gemessen. Die Ergebnisse sind in den Abbildungen 4.4, 4.5 und 4.6 zu sehen.

Zuletzt wird bei jedem Testdurchgang die erstellte Datei wieder geschlossen und alle Daten gelöscht, damit alle Tests unter den gleichen Vorraussetzungen starten.

Die oben genannten Schritte werden jeweils zwei mal ausgeführt. Einmal wird die Standardimplementierung von HDF verwendet, beim zweiten Mal laden wir unser Plugin. Der Test wurde zur Erhebung einer geeigneten Menge an Testwerten insgesamt ca. 900 Mal ausgeführt.

4.3 Auswertung

Abbildung 4.1 zeigt die Benchmark Messungen für das Schreiben von einem 10x10-Datenarray auf ein Dataset. Die Werte der y-Achse sind hierbei in Nanosekunden, auf der x-Achse sind die oben erläuterten Konfigurationen aufgetragen. In der Grafik ist ein deutlicher Zeitunterschied von den rund 18 Microsekunden des Standard HDF auf beiden Speichermedien zu unter 5 Microsekunden beim Memvol Plugin.

In Abbildung 4.2 zu sehen ist die Schreibdauer von Datasets der Größe 50x50. Auch hier sind die Werte der y-Achse Nanosekunden und die x-Achse zeigt die verschiedenen Testfälle. Die Messwerte von rund 20 Microsekunden des Standard HDF5 auf der HDD ändern sich hier kaum im Vergleich zur ersten Abbildung. Die Werte der Standardimplementierung auf der SSD hingegen sind im Schnitt schneller geworden, wobei es viele Ausreißer nach oben gibt. Unsere Implementation hat sich etwas verlangsamt aber hält sich im Schnitt noch unter 10 Microsekunden.

Abbildung 4.3 zeigt den Schreibvorgang auf das dritte und größte Dataset mit den Dimensionen 100x100. Bei der Standard HDF Variante auf der HDD lässt sich hier bis auf zwei starke Ausreißer eine Laufzeit von ungefähr 25 Microsekunden beobachten, auf der SSD reichen die Werte von unter 20 Microsekunden bis hin zu über 30. Das Memvol-Plugin ist mit durchschnittlich 10 bis 20 Microsekunden jedoch immer noch schneller als beide Alternativen.

Abbildungen 4.4, 4.5 und 4.6 zeigen die read Funktion auf Datasets der Größen 10x10, 50x50 und 100x100. Bei den kleinsten der getesteten Datasets kommt es zu den größten Unterschieden zwischen den Messwerten. Unser Plugin braucht hier mit 8 Microsekunden doppelt so lange wie die beiden Varianten der Standardimplementation, wobei die SSD Variante, wie bei allen vorigen Beispielen, noch schneller ist als die HDD.

In Abbildung 4.5 rücken die Werte etwas näher zusammen. Während sich das Memvol-Plugin nur wenig verlangsamt dauert der Lesevorgang bei der Standard Variante von HDF auf der HDD fast 2 Microsekunden länger, auf der SSD verändert sich der Schnitt kaum, doch es gibt zunehmend Ausreißer nach oben, die bis zu 9 Microsekunden betragen.

In Abbildung 4.6 verstärkt sich die eben genannte Tendenz noch. Das Memvol-Plugin und die HDD Standard HDF Implementierung liegen fast gleichauf bei ungefähr 10 Microsekunden. Die Ausführung vom Standard-HDF auf der SSD ist jedoch mit durchschnittlich knapp über 5 Microsekunden immer noch deutlich schneller.

Die Schlussfolgerung aus diesen Messwerten ist, dass sich unsere Erwartung auf schnellere Speicherzugriffe durch die Verwendung des Arbeitsspeichers nur teilweise bestätigt hat. Während die Laufzeit der Schreibenden Zugriffe immer deutlich schneller waren als die der Varianten, die den Sekundärspeicher nutzen, so war doch die benötigte Zeit beim Lesen in allen drei getesteten Fällen größer.

Zu beachten ist hierbei jedoch, dass die Skalen der drei Abbildungen 4.4, 4.5 und 4.6 kleinere Wertemengen umfassen und somit die Unterschiede größer aussehen als sie sind.

Es lässt sich in allen sechs Grafiken erkennen, dass die Laufzeiten des Memvol-Plugins bei Schreibvorgängen immer grob 50% der Zeit benötigen, während die Standard Variante von HDF5 bei den Lesevorgängen grob 50% der Laufzeit des VOL Pugins braucht.

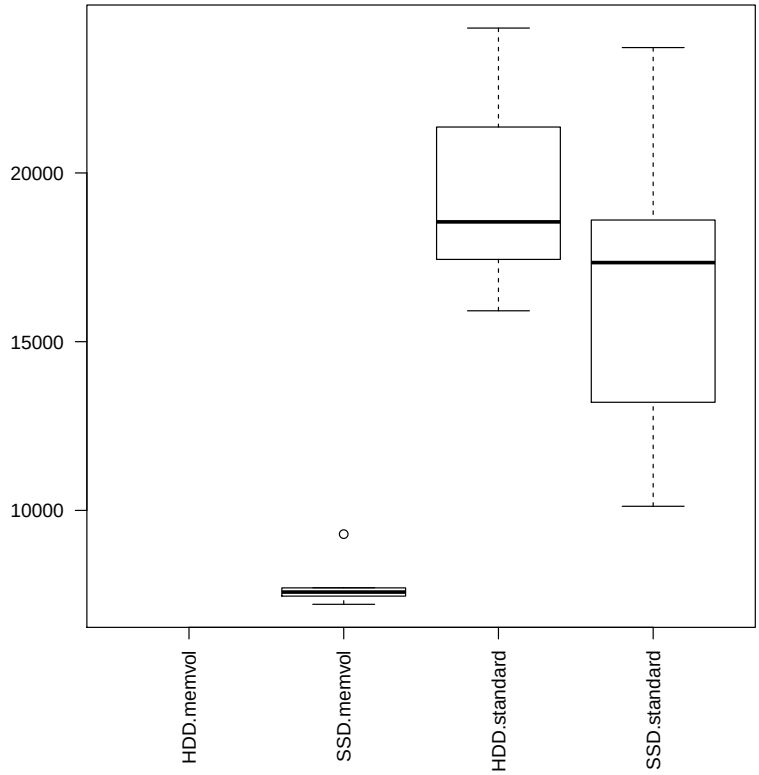

Abbildung 4.1: Dataset Write - Größe 10x10

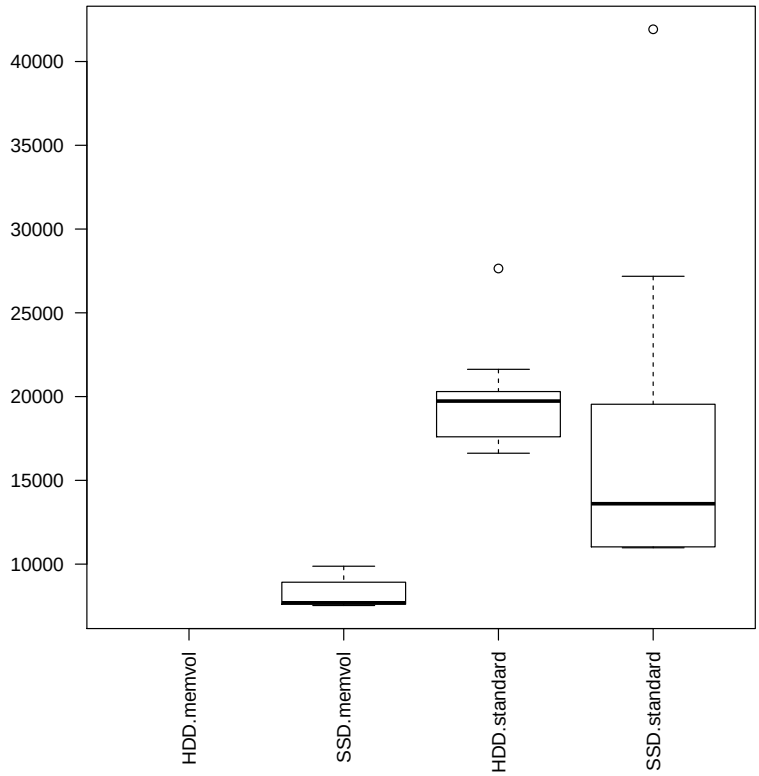

Abbildung 4.2: Dataset Write - Größe 50x50

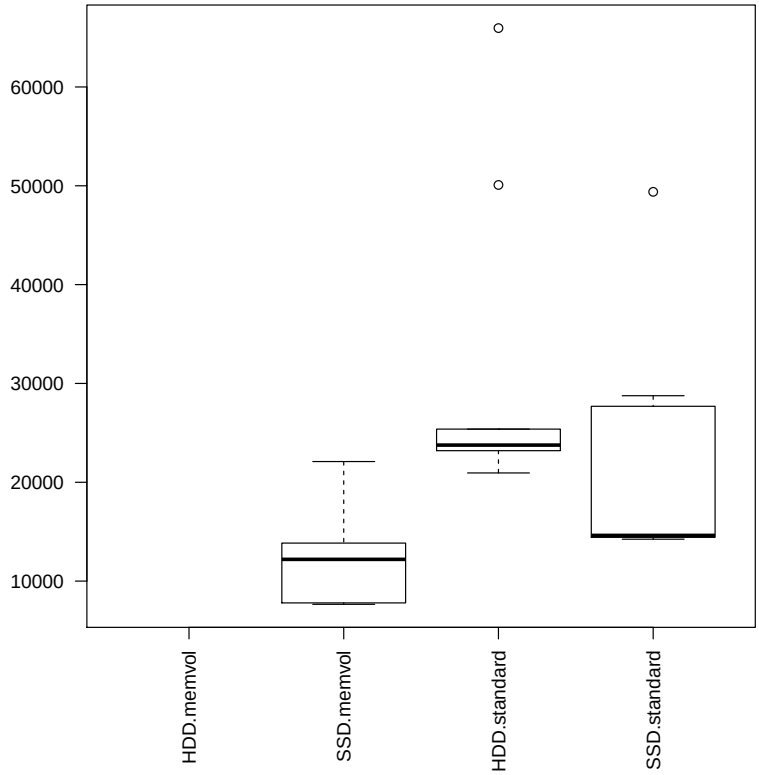

Abbildung 4.3: Dataset Write - Größe 100x100

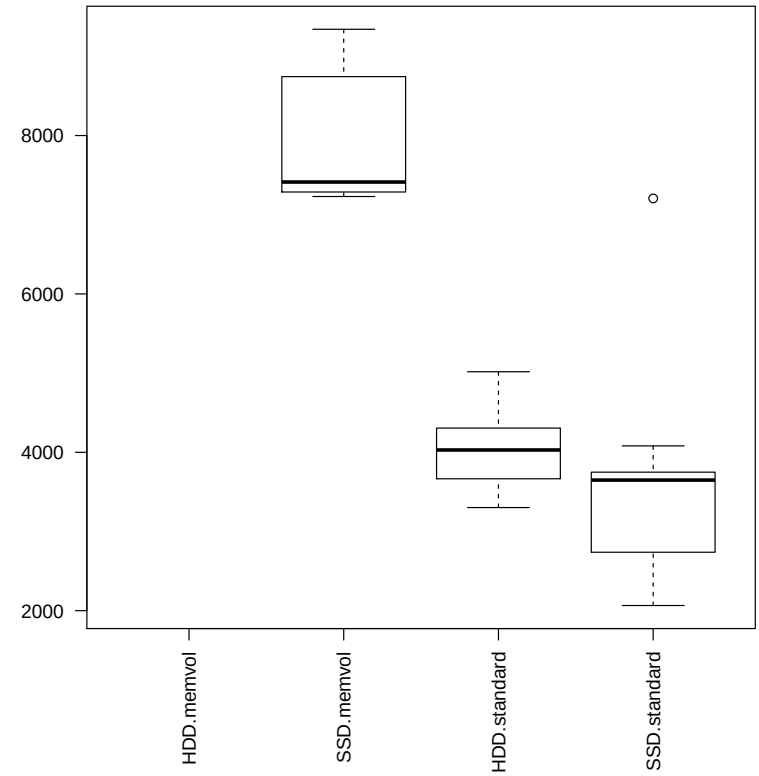

Abbildung 4.4: Dataset Read - Größe 10x10

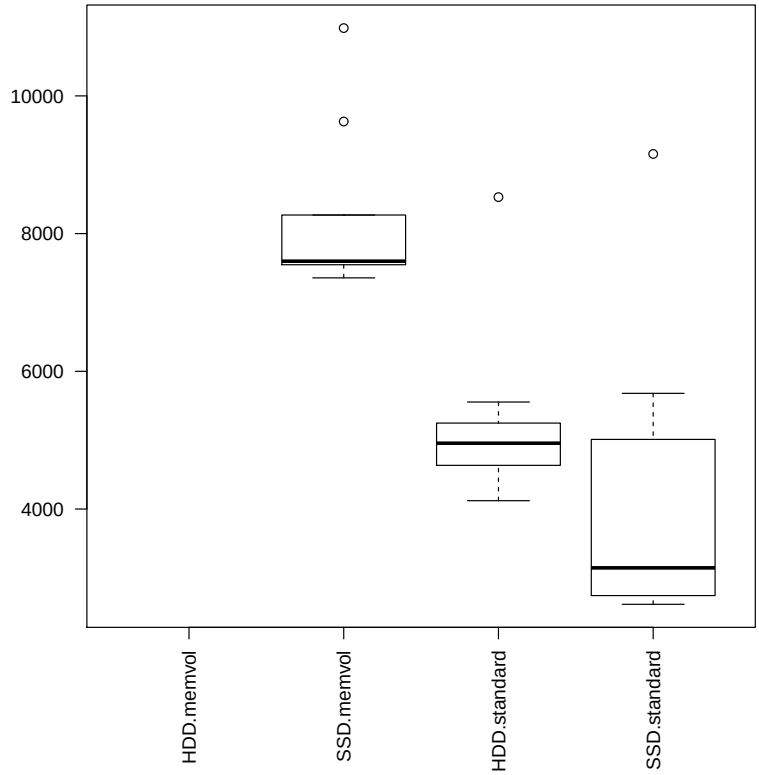

Abbildung 4.5: Dataset Read - Größe 50x50

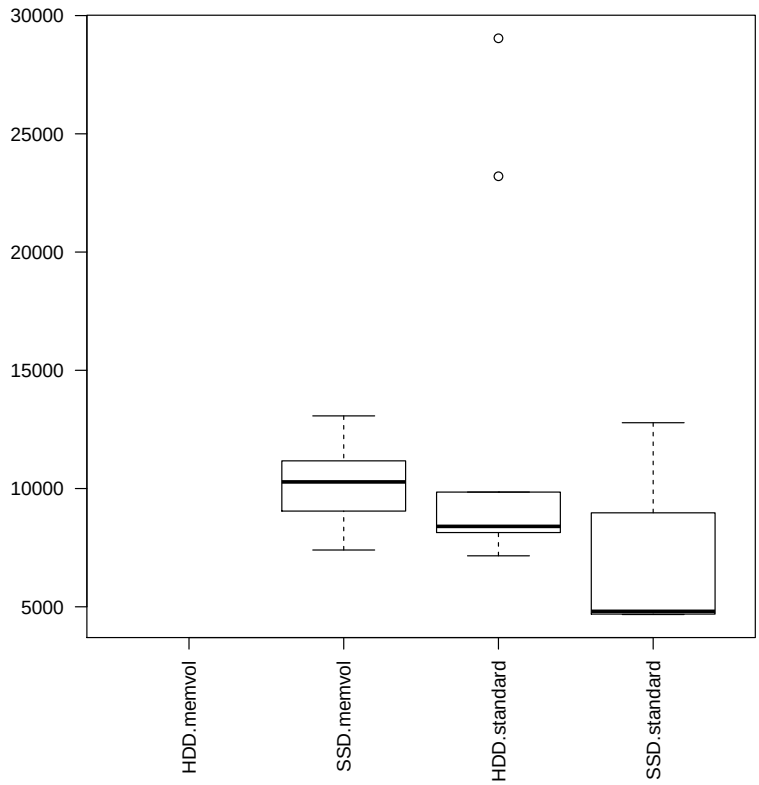

Abbildung 4.6: Dataset Read - Größe 100x100

5 Zusammenfassung, Reflexion und Ausblick

In den fünf Monaten, die wir an diesem Projekt gearbeitet haben, haben wir an folgenden Punkten gearbeitet:

- Einarbeiten in die Funktionsweise und API von HDF5
- Installation von HDF5 und Vertrautmachen mit der Struktur des C Projektes
- Implementation der Callback Methoden und der zugehörigen Datenstrukturen
- Abschließender Benchmark/Performance-Test

Das Ergebnis dieses Projektes ist einerseits ein funktionsfähiges Plugin für HDF5, das die Zugriffszeiten der Standardimplementation teilweise unterschreitet. Außerdem haben wir Einiges im Bereich der Speicherverwaltung ebenso wie den Umgang mit HDF5 gelernt und viele Erfahrungen zur Teamarbeit in größeren Softwareprojekten mitnehmen können.

Eine Mögliche Fortsetzung des Projekts ist einerseits die Hinzunahme der restlichen Funktionalität, wie zum Beispiel `memvol_group_get`. Außerdem eine Erweiterung der Schreib- und Lesefunktion auf Datasets, sodass verschiedene Schreib- und Lesevarianten umgesetzt werden können, wie zum Beispiel partielles Schreiben.

Literaturverzeichnis

[1] Mohamad Chaarawi. User Guide for Developing a Virtual Object Layer Plugin. 2014.

[2] Mohamad Chaarawi, Quincey Koziol. RFC: Virtual Object Layer. 2014.

[3] The HDF Group. Hierarchical Data Format, version 5, 1997-2016. /HDF5/.

[4] The HDF Group. HDF5 User's Guide, 2016. 1.10.0.