HDF5pp Documentation

Tom de Geus

Apr 26, 2018

Contents

CHAPTER 1

Contents

1.1 Usage

1.1.1 General example

The general structure of a program is

```
#include <iostream>
#include <vector>
#include <HDF5pp.h>

int main()
{
  std::vector<double> data = ...;

  H5p::File file = H5p::File("example.hdf5", "w");

  file.write("/path/to/data", data);

  std::vector<double> read = file.read<std::vector<double>>("/path/to/data");

  return 0;
}
```

Note: Although this library is header only, the HDF5 library should be linked. Therefore using either `h5c++` or CMake can be used, see *Compiling*.

1.1.2 Function overview

All functions are members of the File class:

```
H5p::File("/path/to/file", "mode");
```

The constructor takes two arguments: the file name and the read/write mode. For the latter there are three possibilities:

- `"w"`: write a new file or overwrite existing file.
- `"r"`: read from existing file.
- `"r+"` or `"a"`: read from and write to an existing file.

In addition it takes one option, the flush settings. The default `true` ensures the file to be flushed after each write operation, allowing external reading while the file is open.

Main functions:

- `void File::write("/path/to/data",...)`

  Write data (scalar, array, matrix, . . .). Can be overloaded with many different types, see *Overloaded types*.

- `Type File::read<Type>("/path/to/data")`

  Read data (scalar, array, matrix, . . .). Can be templated with many different types, see *Overloaded types*.

- `std::vector<size_t> shape("/path/to/data")`

  Return the shape of the data array.

- `size_t shape("/path/to/data", i)`

  Return the shape of the data array along axis `i`.

- `size_t size("/path/to/data")`

  Return the number of elements in the data array.

Support functions:

- `void File::unlink("/path/to/data")`

  Unlink a path. The dataset is removed when there are no more links to it. Warning: depending on the version of the HDF5 library, the space may not be freed from the file. In that case use `$ h5repack file1 file2` to create a new file without the unused data.

- `bool File::exists("/path/to/data")`

  Check if a path exists.

- `void File::createGroup("/path/to/group")`

  Create a group. Usually there is no need to call this function because the `write` function automatically creates all parent groups.

- `void File::flush()`

  Flush all buffers associated with a file to disk. Usually there is no need to call this function because the `write` function automatically flushes the file (this can be suppressed using the option of the File constructor).

1.1.3 Overloaded types

Note: If your type of choice is not present please submit an issue on GitHub, or file a pull request.

Basic types (size_t, double, . . .)

The examples below feature a `double`, which may be replaced with:

- `int`
- `size_t`
- `float`
- `double`
- `std::string`

Writing and or reading is done as follows:

```cpp
#include <iostream>
#include <vector>
#include <HDF5pp.h>

int main()
{
  double data = 10.;

  H5p::File file = H5p::File("example.hdf5", "w");

  file.write("/path/to/data", data);

  double read_data = file.read<double>("/path/to/data");

  return 0;
}
```

[`source: example.cpp`, `compile: CMakeLists.txt`]

Basic types, part of an expandable array (size_t, double, . . .)

In this case the scalar will be part of an array that automatically expands to contain new entries. The behavior is thus like allocating an array of arbitrary shape and then filling it item-by-item. The actual size is determined by the highest index specified. All entries in the array that have not been explicitly specified are assigned a default fill value. Note:

- One can read one value from, but also read the array as any array (i.e. using `file.read<std::vector<...>>(...)`).
- One can convince oneself about the size of the array using the standard tools (`file.size(...)` and `file.shape(...)`).
- At the first call the array some properties of the array are defined. At this time can choose the fill value (`fill_val`) and the size of the blocks in which the array is stored in the file (`chunk_size`). If one knows the ultimate size one can store in one chunk (most efficient). Otherwise one should choose a value which is high enough not to get a very scattered file, but low enough not to allocate a lot of space that is not used.

The examples below feature a `double`, which may be replaced with:

- `int`
- `size_t`
- `float`
- `double`

Writing and or reading is done as follows:

```
#include <iostream>
#include <vector>
#include <HDF5pp.h>

int main()
{
  H5p::File file = H5p::File("example.hdf5", "w");

  double data = 10.;
  size_t idx  = 0;

  file.write("/path/to/data", data, idx);

  data = 20.;
  idx  = 1;

  // "/path/to/data" is automatically expanded to contain the new entry
  file.write("/path/to/data", data, idx);

  // read one entry
  idx = 0;
  double read_entry = file.read<double>("/path/to/data", idx);

  // read entire array
  std::vector<double> read_data = file.read<std::vector<double>>("/path/to/data");

  return 0;
}
```

[source: example.cpp, compile: CMakeLists.txt]

std::vector

Writing a vector (and optionally its 'dimensions') is done as follows:

```
#include <iostream>
#include <vector>
#include <HDF5pp.h>

int main()
{
  H5p::File file = H5p::File("example.hdf5", "w");

  std::vector<double> data  = { 0., 1., 2., 3., 4., 5. };
  std::vector<size_t> shape = { 3 , 2 };

  file.write("/path/to/data", data, shape);

  std::vector<double> read_data  = file.read<std::vector<double>>("/path/to/data");
  std::vector<size_t> read_shape = file.shape("/path/to/data");

  return 0;
}
```

[source: example.cpp, compile: CMakeLists.txt]

Note: In the HDF5 archive the data is stored as a matrix. However, because `std::vector` is just an array the shape has to be extracted separately. For the richer classes below this is not necessary.

Reading with Python does allow direct interpretation of the matrix

```
import h5py
import numpy as np

f = h5py.File('example.hdf5','r')

print(f['/data'][...])
```

[source: `example.py`]

cppmat - multidimensional arrays

To enable this feature:

- Include cppmat before HDF5pp:

  ```
  #include <cppmat/cppmat.h>
  #include <HDF5pp.h>
  ```

- Define `HDF5PP_CPPMAT` somewhere before including HDF5pp:

  ```
  #define HDF5PP_CPPMAT
  #include <HDF5pp.h>
  #include <cppmat/cppmat.h>
  ```

Writing and reading matrices of arbitrary dimensions can be done as follows:

```
#include <iostream>
#include <cppmat/cppmat.h>
#include <HDF5pp.h>

int main()
{
  cppmat::matrix<double> data({2,3,4,5});

  // ... fill "data"

  H5p::File file = H5p::File("example.hdf5", "w");

  file.write("/path/to/data", data);

  cppmat::matrix<double> read_data = file.read<cppmat::matrix<double>>("/path/to/data
↪");

  return 0;
}
```

[source: `example.cpp`, compile: `CMakeLists.txt`]

Eigen - linear algebra library

To enable this feature:

- Include Eigen before HDF5pp:

  ```
  #include <Eigen/Eigen>
  #include <HDF5pp.h>
  ```

- Define `HDF5PP_EIGEN` somewhere before including HDF5pp:

  ```
  #define HDF5PP_EIGEN
  #include <HDF5pp.h>
  #include <Eigen/Eigen>
  ```

Writing and reading matrices or arrays can be done as follows:

```
#include <iostream>
#include <Eigen/Eigen>
#include <HDF5pp.h>

// alias row-major Eigen matrix
typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> MatD;

int main()
{
  MatD data(2,2);

  // ... fill "data"

  H5p::File file = H5p::File("example.hdf5", "w");

  file.write("/path/to/data", data);

  MatD read_data = file.read<MatD>("/path/to/data");

  return 0;
}
```

[`source: example.cpp`, `compile: CMakeLists.txt`]

1.2 Compiling

1.2.1 Introduction

This module is header only. So one just has to `#include <HDF5pp/HDF5pp.h>` (or only one of the submodules) somewhere in the source code, and to tell the compiler where the header-files are. For the latter, several ways are described below.

One should still link with the HDF5 libraries. This is briefly described in *Linking with the HDF5 libraries*.

Note: Before proceeding, some words about optimization. Of course one should use optimization when compiling the release of the code (`-O2` or `-O3`). But it is also a good idea to switch of the assertions in the code (mostly checks on size) that facilitate easy debugging, but do cost time. Therefore, include the flag `-DNDEBUG`. Note that this is all C++ standard. I.e. it should be no surprise, and it is always a good idea to do.

1.2.2 Manual compiler flags

GNU / Clang

Add the following compiler's arguments:

```
-I${PATH_TO_HDF5PP}/src -std=c++14
```

Note: (Not recommended)

If you want to avoid separately including the header files using a compiler flag, `git submodule` is a nice way to go:

1. Include this module as a submodule using

   `git submodule add https://github.com/tdegeus/HDF5pp.git`.

2. Replace the first line of this example by

   `#include "HDF5pp/src/HDF5pp/HDF5pp.h"`.

   If you decide to manually copy the header file, you might need to modify this relative path to your liking.

Or see *(Semi-)Automatic compiler flags*. You can also combine the `git submodule` with any of the below compiling strategies.

1.2.3 (Semi-)Automatic compiler flags

Install

To enable (semi-)automatic build, one should 'install' HDF5pp somewhere.

Install systemwide (depends on your privileges)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/HDF5pp/src/build
```

2. 'Build' HDF5pp. For the path above,

```
$ cmake ..
$ make install
```

   (If you've used another build directory, change the first command to `$ cmake /path/to/HDF5pp/src`)

Install in custom location (user)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/HDF5pp/src/build
```

2. 'Build' HDF5pp, to install it in a custom location. For the path above,

```
$ mkdir /custom/install/path
$ cmake .. -DCMAKE_INSTALL_PREFIX:PATH=/custom/install/path
$ make install
```

(If you've used another build directory, change the first command to `$ cmake /path/to/HDF5pp/src`)

3. Add the following path to your `~/.bashrc` (or `~/.zshrc`):

```
export PKG_CONFIG_PATH=/custom/install/path/share/pkgconfig:$PKG_CONFIG_PATH
```

Note: (Not recommended)

If you do not wish to use `CMake` for the installation, or you want to do something custom. You can, of course. Follow these steps:

1. Copy the file `src/HDF5pp.pc.in` to `HDF5pp.pc` to some location that can be found by `pkg_config` (for example by adding `export PKG_CONFIG_PATH=/path/to/HDF5pp.pc:$PKG_CONFIG_PATH` to the `.bashrc`).
2. Modify the line `prefix=@CMAKE_INSTALL_PREFIX@` to `prefix=/path/to/HDF5pp`.
3. Modify the line `Cflags: -I${prefix}/@INCLUDE_INSTALL_DIR@` to `Cflags: -I${prefix}/src`.
4. Modify the line `Version: @HDF5PP_VERSION_NUMBER@` to reflect the correct release version.

Compiler arguments from 'pkg-config'

Instead of `-I...` one can now use

```
`pkg-config --cflags HDF5pp` -std=c++14
```

as compiler argument.

Compiler arguments from 'cmake'

Add the following to your `CMakeLists.txt`:

```
set(CMAKE_CXX_STANDARD 14)

find_package(PkgConfig)

pkg_check_modules(HDF5PP REQUIRED HDF5pp)
include_directories(${HDF5PP_INCLUDE_DIRS})
```

1.2.4 Linking with the HDF5 libraries

Using the h5c++ executable

The `h5c++` executable provides a wrapper around your compiler, with all flags set correctly to use HDF5. To compile the following suffices:

```
h5c++ `pkg-config --cflags HDF5pp` -std=c++14 example.cpp
```

Using cmake

The following basic structure of `CMakeLists.txt` can be used:

```
cmake_minimum_required(VERSION 2.8.12)

# define a project name
project(example)

# define empty list of libraries to link
set(PROJECT_LIBS "")

# enforce the C++ standard
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# set optimization level and switch of assertions (set to your liking)
set(CMAKE_BUILD_TYPE Release)
add_definitions(-DNDEBUG)

# load pkg-config
find_package(PkgConfig)

# find HDF5
find_package(HDF5 COMPONENTS CXX REQUIRED)
include_directories(${HDF5_INCLUDE_DIRS})
set(PROJECT_LIBS ${HDF5_LIBS} ${HDF5_LIBRARIES})

# find HDF5pp
pkg_check_modules(HDF5PP REQUIRED HDF5pp)
include_directories(${HDF5PP_INCLUDE_DIRS})

# create executable
add_executable(${PROJECT_NAME} example.cpp)

# link libraries
target_link_libraries(${PROJECT_NAME} ${PROJECT_LIBS})
```

1.3 Notes for developers

1.3.1 Make changes / additions

Be sure to run and verify all examples! All existing examples should pass, while new examples should be added to document and check any new functionality.

1.3.2 Create a new release

1. Update the version number specified in `HDF5pp.h` using `HDF5PP_WORLD_VERSION`, `HDF5PP_MAJOR_VERSION`, `HDF5PP_MINOR_VERSION`.
2. Upload the changes to GitHub and create a new release there (with the correct version number).

CHAPTER 2

Teaser

This header-only module provides a very simple interface to store data to a HDF5-file. Using this library writing an HDF5 file becomes as simple as:

```
#include <iostream>
#include <vector>
#include <HDF5pp.h>

int main()
{
  std::vector<double> data = ...;

  H5p::File file = H5p::File("example.hdf5","w");

  file.write("/path/to/data", data);

  return 0;
}
```

Note: Compilation of this example for example with

```
h5c++ -std=c++14 `pkg-config --cflags HDF5pp` example.cpp
```

But probably most easily using CMake. See *Compiling* for more details.

CHAPTER 3

Disclaimer

Download: .zip file | .tar.gz file.

(c - MIT) T.W.J. de Geus (Tom) | tom@geus.me | www.geus.me | github.com/tdegeus/HDF5pp