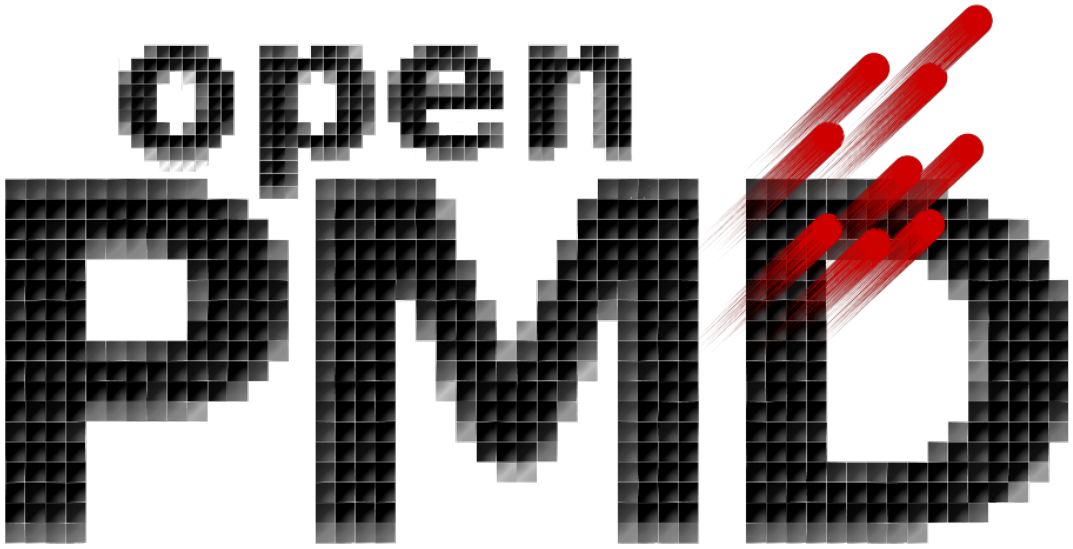

openPMD-api Documentation

Release 0.8.0-alpha

The openPMD Community

May 24, 2019

INSTALLATION

This library provides an abstract API for openPMD file handling. It provides both support for writing & reading into various formats and works both serial and parallel (MPI). Implemented backends include HDF5 and ADIOS1.

CHAPTER 1

Doxygen

The lastest Doxygen docs for the C++ API are published at:

http://www.openpmd.org/openPMD-api

CHAPTER 2

Supported openPMD Standard Versions

openPMD-api is a library using semantic versioning, starting with version 1.0.0.

The supported version of the openPMD standard are reflected as follows: `standardMAJOR.apiMAJOR.apiMINOR`.

openPMD-api version	supported openPMD standard versions
`1.0.0+`	`1.0.1-1.1.0` (not released yet)
`2.0.0+`	`2.0.0+` (not released yet)
`0.1.0-0.8.0` (alpha)	`1.0.0-1.1.0`

2.1 Installation

2.1.1 Installation

Choose *one* of the install methods below to get started:

Using the Spack Package

A package for openPMD-api is available on the Spack package manager.

```
# optional:                +python +adios1 -mpi
spack install openpmd-api
spack load -r openpmd-api
```

Using the Conda Package

A package for serial openPMD-api is available on the Conda package manager.

```
conda install -c conda-forge openpmd-api
```

Using the PyPI Package

A package for openPMD-api is available on the Python Package Index (PyPI).

Behind the scenes, this install method *compiles from source* against the found installations of HDF5, ADIOS and/or MPI (in system paths, from other package managers, or loaded via a module system, ...). The current status for this install method is *experimental*. Please feel free to report how this works for you.

```
# optional:             --user
pip install openpmd-api
```

or with MPI support:

```
# optional:                                 --user
openPMD_USE_MPI=ON pip install openpmd-api
```

From Source with CMake

You can also install `openPMD-api` from source with CMake. This requires that you have all *dependencies* installed on your system. The developer section on *build options* provides further details on variants of the build.

Linux & OSX

```
git clone https://github.com/openPMD/openPMD-api.git

mkdir openPMD-api-build
cd openPMD-api-build

# optional: for full tests
../openPMD-api/.travis/download_samples.sh

# for own install prefix append:
#   -DCMAKE_INSTALL_PREFIX=$HOME/somepath
# for options append:
#   -DopenPMD_USE_...=...
# e.g. for python support add:
#   -DopenPMD_USE_PYTHON=ON -DPYTHON_EXECUTABLE=$(which python)
cmake ../openPMD-api

cmake --build .

# optional
ctest

# sudo might be required for system paths
cmake --build . --target install
```

Windows

The process is basically similar to Linux & OSX, with just a couple of minor tweaks. Use `ps .. \openPMD-api\.travis\download_samples.ps1` to download sample files for tests (optional). Replace the last three commands with

```
cmake --build . --config Release

# optional
ctest -C Release
```

(continues on next page)

(continued from previous page)

```
# administrative privileges might be required for system paths
cmake --build . --config Release --target install
```

Post "From Source" Install

If you installed to a non-system path on Linux or OSX, you need to express where your newly installed library can be found.

Adjust the lines below accordingly, e.g. replace `$HOME/somepath` with your install location prefix in `-DCMAKE_INSTALL_PREFIX=...`. CMake will summarize the install paths for you before the build step.

```
# install prefix          |------------|
export CMAKE_PREFIX_PATH=$HOME/somepath:$CMAKE_PREFIX_PATH
export LD_LIBRARY_PATH=$HOME/somepath/lib:$LD_LIBRARY_PATH

#                change path to your python MAJOR.MINOR version
export PYTHONPATH=$HOME/somepath/lib/python3.5/site-packages:$PYTHONPATH
```

Adding those lines to your `$HOME/.bashrc` and re-opening your terminal will set them permanently.

Set hints on Windows with the CMake printed paths accordingly, e.g.:

```
set CMAKE_PREFIX_PATH=C:\\Program Files\openPMD;%CMAKE_PREFIX_PATH%
set PATH=C:\\Program Files\openPMD\Lib;%PATH%
set PYTHONPATH=C:\\Program Files\openPMD\Lib\site-packages;%PYTHONPATH%
```

2.1.2 Changelog

0.8.0-alpha

Date: 2019-03-09

Python mpi4py and Slice Support

We implemented MPI support for the Python frontend via `mpi4py` and added `[]`-slice access to `Record_Component` loads and stores. A bug requiring write permissions for read-only series was fixed and memory provided by users is now properly checked for being contiguous. Introductory chapters in the manual have been greatly extended.

Changes to "0.7.1-alpha"

Features

- Python:
  - mpi4py support added #454
  - slice protocol for record component #458

Bug Fixes

- do not require write permissions to open `Series` read-only #395
- loadChunk: re-enable range/extent checks for adjusted ranges #469
- Python: stricter contiguous check for user-provided arrays #458

- CMake tests as root: apply OpenMPI flag only if present #456

Other

- increase pybind11 dependency to 2.2.4+ #455
- Python: remove (inofficial) bindings for 2.7 #435
- CMake 3.12+: apply policy `CMP0074` for `<Package>_ROOT` vars #391 #464
- CMake: Optional ADIOS1 Wrapper Libs #472
- MPark.Variant: updated to 1.4.0+ #465
- Catch2: updated to 2.6.1+ #466
- NLohmann-JSON: updated to 3.5.0+ #467
- Docs:
  - PyPI install method #450 #451 #497
  - more info on MPI #449
  - new "first steps" section #473 #478
  - update invasive test info #474
  - more info on `AccessType` #483
  - improved MPI-parallel write example #496

0.7.1-alpha

Date: 2018-01-23

Bug Fixes in Multi-Platform Builds

This release fixes several issues on OSX, during cross-compile and with modern compilers.

Changes to "0.7.0-alpha"

Bug Fixes

- fix compilation with C++17 for python bindings #438
- `FindADIOS.cmake`: Cross-Compile Support #436
- ADIOS1: fix runtime crash with libc++ (e.g. OSX) #442

Other

- CI: clang libc++ coverage #441 #444
- Docs:
  - additional release workflows for maintainers #439
  - ADIOS1 backend options in manual #440
  - updated Spack variants #445

0.7.0-alpha

Date: 2019-01-11

JSON Support, Interface Simplification and Stability

This release introduces serial JSON (`.json`) support. Our API has been unified with slight breaking changes such as a new Python module name (`import openpmd_api` from now on) as well as re-ordered `store/loadChunk` argument orders. Please see our new "upgrade guide" section in the manual how to update existing scripts. Additionally, many little bugs have been fixed. Official Python 3.7 support and a parallel benchmark example have been added.

Changes to "0.6.3-alpha"

Features

- C++:
  - `storeChunk` argument order changed, defaults added #386 #416
  - `loadChunk` argument order changed, defaults added #408
- Python:
  - `import openPMD` renamed to `import openpmd_api` #380 #392
  - `store_chunk` argument order changed, defaults added #386
  - `load_chunk` defaults added #408
  - works with Python 3.7 #376
  - setup.py for sdist #240
- Backends: JSON support added #384 #393 #338 #429
- Parallel benchmark added #346 #398 #402 #411

Bug Fixes

- spurious MPI C++11 API usage in ParallelIOTest removed #396
- spurious symbol issues on OSX #427
- `new []`/`delete` mismatch in ParallelIOTest #422
- use-after-free in SerialIOTest #409
- fix ODR issue in ADIOS1 backend corrupting the `AbstractIOHandler` vtable #415
- fix race condition in MPI-parallel directory creation #419
- ADIOS1: fix use-after-free in parallel I/O method options #421

Other

- modernize `IOTask`'s `AbstractParameter` for slice safety #410
- Docs: upgrade guide added #385
- Docs: python particle writing example #430
- CI: GCC 8.1.0 & Python 3.7.0 #376
- CI: (re-)activate Clang-Tidy #423

- IOTask: init all parameters' members #420
- KDevelop project files to `.gitignore` #424
- C++:
  - `Mesh`'s `setAxisLabels|GridSpacing|GridGlobalOffset` passed as `const &` #425
- CMake:
  - treat third party libraries properly as `IMPORTED` #389 #403
  - Catch2: separate implementation and tests #399 #400
  - enable check for more warnings #401

0.6.3-alpha

Date: 2018-11-12

Reading Varying Iteration Padding Reading

Support reading series with varying iteration padding (or no padding at all) as currently used in PIConGPU.

Changes to "0.6.2-alpha"

Bug Fixes

- support reading series with varying or no iteration padding in filename #388

0.6.2-alpha

Date: 2018-09-25

Python Stride: Regression

A regression in the last fix for python strides made the relaxation not efficient for 2-D and higher.

Changes to "0.6.1-alpha"

Bug Fixes

- Python: relax strides further

0.6.1-alpha

Date: 2018-09-24

Relaxed Python Stride Checks

Python stride checks have been relaxed and one-element n-d arrays are allowed for scalars.

Changes to "0.6.0-alpha"

Bug Fixes

- Python:
  - stride check too strict #369

  - allow one-element n-d arrays for scalars in `store`, `make_constant` #314

Other

- dependency change: Catch2 2.3.0+
- Python: add extended write example #314

0.6.0-alpha

Date: 2018-09-20

Particle Patches Improved, Constant Scalars and Python Containers Fixed

Scalar records properly support const-ness. The Particle Patch load interface was changed, loading now all patches at once, and Python bindings are available. Numpy `dtype` is now a first-class citizen for Python `Datatype` control, being accepted and returned instead of enums. Python lifetime in garbage collection for containers such as `meshes`, `particles` and `iterations` is now properly implemented.

Changes to "0.5.0-alpha"

Features

- Python:
  - accept & return `numpy.dtype` for `Datatype` #351
  - better check for (unsupported) numpy array strides #353
  - implement `Record_Component.make_constant` #354
  - implement `Particle_Patches` #362
- comply with runtime constraints w.r.t. `written` status #352
- load at once `ParticlePatches.load()` #364

Bug Fixes

- dataOrder: mesh attribute is a string #355
- constant scalar Mesh Records: reading corrected #358
- particle patches: stricter `load( idx )` range check #363, then removed in #364
- Python: lifetime of `Iteration.meshes/particles` and `Series.iterations` members #354

Other

- test cases for mixed constant/non-constant Records #358
- examples: close handles explicitly #359 #360

0.5.0-alpha

Date: 2018-09-17

Refactored Type System

The type system for `Datatype::``s was refactored. Integer types are now represented by ``SHORT`, `INT`, `LONG` and `LONGLONG` as fundamental C/C++ types. Python support enters "alpha" stage with fixed floating point storage and `Attribute` handling.

Changes to "0.4.0-alpha"

Features

- Removed `Datatype::INT32` types with `::SHORT`, `::INT` equivalents #337
- `Attribute::get<...>()` performs a `static_cast` now #345

Bug Fixes

- Refactor type system and `Attribute` set/get
  - integers #337
  - support `long double` reads on MSVC #184
- `setAttribute`: explicit C-string handling #341
- `Dataset`: `setCompression` warning and error logic #326
- avoid impact on unrelated classes in invasive tests #324
- Python
  - single precision support: `numpy.float` is an alias for `builtins.float` #318 #320
  - `Dataset` method namings to underscores #319
  - container namespace ambiguity #343
  - `set_attribute`: broken numpy, list and string support #330

Other

- CMake: invasive tests not enabled by default #323
- `store_chunk`: more detailed type mismatch error #322
- `no_such_file_error` & `no_such_attribute_error`: remove c-string constructor #325 #327
- add virtual destructor to `Attributable` #332
- Python: Numpy 1.15+ required #330

0.4.0-alpha

Date: 2018-08-27

Improved output handling

Refactored and hardened for `fileBased` output. Records are not flushed before the ambiguity between scalar and vector records are resolved. Trying to write globally zero-extent records will throw gracefully instead of leading to undefined behavior in backends.

Changes to "0.3.1-alpha"

Features

- do not assume record structure prematurely #297
- throw in (global) zero-extent dataset creation and write #309

Bug Fixes

- ADIOS1 `fileBased` IO #297
- ADIOS2 stub header #302
- name sanitization in ADIOS1 and HDF5 backends #310

Other

- CI updates: #291
  - measure C++ unit test coverage with coveralls
  - clang-format support
  - clang-tidy support
  - include-what-you-use support #291 export headers #300
  - OSX High Sierra support #301
  - individual cache per build # 303
  - readable build names #308
- remove superfluous whitespaces #292
- readme: openPMD is for scientific data #294
- `override` implies `virtual` #293
- spack load: `-r` #298
- default constructors and destructors #304
- string pass-by-value #305
- test cases with 0-sized reads & writes #135

0.3.1-alpha

Date: 2018-07-07

Refined fileBased Series & Python Data Load

A specification for iteration padding in filenames for `fileBased` series is introduced. Padding present in read iterations is detected and conserved in processing. Python builds have been simplified and python data loads now work for both meshes and particles.

Changes to "0.3.0-alpha"

Features

- CMake:

  - add `openPMD::openPMD` alias for full-source inclusion #277
  - include internally shipped pybind11 v2.2.3 #281
  - ADIOS1: enable serial API usage even if MPI is present #252 #254
- introduce detection and specification `%0\d+T` of iteration padding #270
- Python:
  - add unit tests #249
  - expose record components for particles #284

Bug Fixes

- improved handling of `fileBased` Series and `READ_WRITE` access
- expose `Container` constructor as `protected` rather than `public` #282
- Python:
  - return actual data in `load_chunk` #286

Other

- docs:
  - improve "Install from source" section #274 #285
  - Spack python 3 install command #278

0.3.0-alpha

Date: 2018-06-18

Python Attributes, Better FS Handling and Runtime Checks

This release exposes openPMD attributes to Python. A new independent mechanism for verifying internal conditions is now in place. Filesystem support is now more robust on varying directory separators.

Changes to "0.2.0-alpha"

Features

- CMake: add new `openPMD_USE_VERIFY` option #229
- introduce `VERIFY` macro for pre-/post-conditions that replaces `ASSERT` #229 #260
- serial Singularity container #236
- Python:
  - expose attributes #256 #266
  - use lists for offsets & extents #266
- C++:
  - `setAttribute` signature changed to const ref #268

Bug Fixes

- handle directory separators platform-dependent #229
- recursive directory creation with existing base #261
- `FindADIOS.cmake`: reset on multiple calls #263
- `SerialIOTest`: remove variable shadowing #262
- ADIOS1: memory violation in string attribute writes #269

Other

- enforce platform-specific directory separators on user input #229
- docs:
  - link updates to https #259
  - minimum MPI version #251
  - title updated #235
- remove MPI from serial ADIOS interface #258
- better name for scalar record in examples #257
- check validity of internally used pointers #247
- various CI updates #246 #250 #261

0.2.0-alpha

Date: 2018-06-11

Initial Numpy Bindings

Adds first bindings for record component reading and writing. Fixes some minor CMake issues.

Changes to "0.1.1-alpha"

Features

- Python: first NumPy bindings for record component chunk store/load #219
- CMake: add new `BUILD_EXAMPLES` option #238
- CMake: build directories controllable #241

Bug Fixes

- forgot to bump `version.hpp`/`__version__` in last release
- CMake: Overwritable Install Paths #237

0.1.1-alpha

Date: 2018-06-07

ADIOS1 Build Fixes & Less Flushes

We fixed build issues with the ADIOS1 backend. The number of performed flushes in backends was generally minimized.

Changes to "0.1.0-alpha"

Bug Fixes

- SerialIOTest: `loadChunk` template missing for ADIOS1 #227
- prepare running serial applications linked against parallel ADIOS1 library #228

Other

- minimize number of flushes in backend #212

0.1.0-alpha

Date: 2018-06-06

This is the first developer release of openPMD-api.

Both HDF5 and ADIOS1 are implemented as backends with serial and parallel I/O support. The C++11 API is considered alpha state with few changes expected to come. We also ship an unstable preview of the Python3 API.

2.1.3 Upgrade Guide

0.7.0-alpha

Python

Module Name

Our module name has changed to be consistent with other openPMD projects:

```
# old name
import openPMD

# new name
import openpmd_api
```

`store_chunk` Method

The order of arguments in the `store_chunk` method for record components has changed. The new order allows to make use of defaults in many cases in order reduce complexity.

```
particlePos_x = np.random.rand(234).astype(np.float32)

d = Dataset(particlePos_x.dtype, extent=particlePos_x.shape)
electrons["position"]["x"].reset_dataset(d)

# old code
electrons["position"]["x"].store_chunk([0, ], particlePos_x.shape, particlePos_x)

# new code
electrons["position"]["x"].store_chunk(particlePos_x)
# implied defaults:
#                           .store_chunk(particlePos_x,
#                                        offset=[0, ],
#                                        extent=particlePos_x.shape)
```

`load_chunk` Method

The `loadChunk<T>` method with on-the-fly allocation has default arguments for offset and extent now. Called without arguments, it will read the whole record component.

```
E_x = series.iterations[100].meshes["E"]["x"]

# old code
all_data = E_x.load_chunk(np.zeros(E_x.shape), E_x.shape)

# new code
all_data = E_x.load_chunk()

series.flush()
```

C++

`storeChunk` Method

The order of arguments in the `storeChunk` method for record components has changed. The new order allows to make use of defaults in many cases in order reduce complexity.

```
std::vector< float > particlePos_x(234, 1.234);

Datatype datatype = determineDatatype(shareRaw(particlePos_x));
Extent extent = {particlePos_x.size()};
Dataset d = Dataset(datatype, extent);
electrons["position"]["x"].resetDataset(d);

// old code
electrons["position"]["x"].storeChunk({0}, extent, shareRaw(particlePos_x));

// new code
electrons["position"]["x"].storeChunk(particlePos_x);
/* implied defaults:
 *                           .storeChunk(shareRaw(particlePos_x),
 *                                       {0},
 *                                       {particlePos_x.size()})   */
```

`loadChunk` Method

The order of arguments in the pre-allocated data overload of the `loadChunk` method for record components has changed. The new order allows was introduced for consistency with `storeChunk`.

```
float loadOnePos;

// old code
electrons["position"]["x"].loadChunk({0}, {1}, shareRaw(&loadOnePos));

// new code
electrons["position"]["x"].loadChunk(shareRaw(&loadOnePos), {0}, {1});

series.flush();
```

The `loadChunk<T>` method with on-the-fly allocation got default arguments for offset and extent. Called without arguments, it will read the whole record component.

```
MeshRecordComponent E_x = series.iterations[100].meshes["E"]["x"];

// old code
auto all_data = E_x.loadChunk<double>({0, 0, 0}, E_x.getExtent());

// new code
auto all_data = E_x.loadChunk<double>();

series.flush();
```

2.2 Usage

2.2.1 First Write

Step-by-step: how to write scientific data with openPMD-api?

Include / Import

After successful *installation*, you can start using openPMD-api as follows:

C++11

```
#include <openPMD/openPMD.hpp>

// example: data handling
#include <numeric>  // std::iota
#include <vector>   // std::vector

namespace api = openPMD;
```

Python

```
import openpmd_api as api

# example: data handling
import numpy as np
```

Open

Write into a new openPMD series in `myOutput/data_<00...N>.h5`. Further file formats than `.h5` (HDF5) are supported: `.bp` (ADIOS1) or `.json` (JSON).

C++11

```
auto series = api::Series(
    "myOutput/data_%05T.h5",
    api::AccessType::CREATE);
```

Python

```
series = api.Series(
    "myOutput/data_%05T.h5",
    api.Access_Type.create)
```

Iteration

Grouping by an arbitrary, positive integer number `<N>` in a series:

C++11

```
auto i = series.iterations[42];
```

Python

```
i = series.iterations[42]
```

Attributes

Everything in openPMD can be extended and user-annotated. Let us try this by writing some meta data:

C++11

```
series.setAuthor(
    "Axel Huebl <a.huebl@hzdr.de>");
series.setMachine(
    "Hall Probe 5000, Model 3");
series.setAttribute(
    "dinner", "Pizza and Coke");
i.setAttribute(
    "vacuum", true);
```

Python

```
series.set_author(
    "Axel Huebl <a.huebl@hzdr.de>")
series.set_machine(
    "Hall Probe 5000, Model 3")
series.set_attribute(
    "dinner", "Pizza and Coke")
i.set_attribute(
    "vacuum", True)
```

Data

Let's prepare some data that we want to write. For example, a magnetic field $\vec{B}(i, j)$ slice in two dimensions with three components $(B_x, B_y, B_z)^\intercal$ of which the B_y component shall be constant for all (i, j) indices.

C++11

```
std::vector<float> x_data(
    150 * 300);
std::iota(
    x_data.begin(),
    x_data.end(),
    0.);

float y_data = 4.f;

std::vector<float> z_data(x_data);
for( auto& c : z_data )
    c -= 8000.f;
```

Python

```
x_data = np.arange(
    150 * 300,
    dtype=np.float
).reshape(150, 300)



y_data = 4.

z_data = x_data.copy() - 8000.
```

Record

An openPMD record can be either structured (mesh) or unstructured (particles). We prepared a vector field in 2D above, which is a mesh:

C++11

```
// record
auto B = i.meshes["B"];

// record components
```

(continues on next page)

(continued from previous page)

```
auto B_x = B["x"];
auto B_y = B["y"];
auto B_z = B["z"];

auto dataset = api::Dataset(
    api::determineDatatype<float>(),
    {150, 300});
B_x.resetDataset(dataset);
B_y.resetDataset(dataset);
B_z.resetDataset(dataset);
```

Python

```
# record
B = i.meshes["B"]

# record components
B_x = B["x"]
B_y = B["y"]
B_z = B["z"]

dataset = api.Dataset(
    x_data.dtype,
    x_data.shape)
B_x.reset_dataset(dataset)
B_y.reset_dataset(dataset)
B_z.reset_dataset(dataset)
```

Units

Ouch, our measured magnetic field data is in Gauss! Quick, let's store the conversion factor to SI (Tesla).

C++11

```
// conversion to SI
B_x.setUnitSI(1.e-4);
B_y.setUnitSI(1.e-4);
B_z.setUnitSI(1.e-4);

// unit system agnostic dimension
B.setUnitDimension({
    {api::UnitDimension::M,  1},
    {api::UnitDimension::I, -1},
    {api::UnitDimension::T, -2}
});
```

Python

```
# conversion to SI
B_x.set_unit_SI(1.e-4)
B_y.set_unit_SI(1.e-4)
B_z.set_unit_SI(1.e-4)
```

(continues on next page)

(continued from previous page)

```
# unit system agnostic dimension
B.set_unit_dimension({
    api.Unit_Dimension.M:  1,
    api.Unit_Dimension.I: -1,
    api.Unit_Dimension.T: -2
})
```

Tip: Annotating the dimensionality of a record allows us to read data sets with *arbitrary names* and understand their purpose simply by *dimensional analysis*.

Register Chunk

We can write record components partially and in parallel or at once. Writing very small data one by one is is a performance killer for I/O. Therefore, we register all data to be written first and then flush it out collectively.

C++11

```
B_x.storeChunk(
    api::shareRaw(x_data),
    {0, 0}, {150, 300});
B_z.storeChunk(
    api::shareRaw(z_data),
    {0, 0}, {150, 300});

B_y.makeConstant(y_data);
```

Python

```
B_x.store_chunk(x_data)


B_z.store_chunk(z_data)



B_y.make_constant(y_data)
```

Attention: After registering a data chunk such as `x_data` and `y_data`, it MUST NOT be modified or deleted until the `flush()` step is performed!

Flush Chunk

We now flush the registered data chunks to the I/O backend. Flushing several chunks at once allows to increase I/O performance significantly. After that, the variables `x_data` and `y_data` can be used again.

C++11

```
series.flush();
```

Python

```
series.flush()
```

Close

Finally, the Series is fully closed (and newly registered data or attributes since the last `.flush()` is written) when its destructor is called.

C++11

```
// destruct series object,
// e.g. when out-of-scope
```

Python

```
del series
```

2.2.2 First Read

Step-by-step: how to read openPMD data? We are using the examples files from openPMD-example-datasets (`example-3d.tar.gz`).

Include / Import

After successful *installation*, you can start using openPMD-api as follows:

C++11

```
#include <openPMD/openPMD.hpp>

// example: data handling & print
#include <vector>   // std::vector
#include <iostream> // std::cout
#include <memory>   // std::shared_ptr

namespace api = openPMD;
```

Python

```
import openpmd_api as api

# example: data handling
import numpy as np
```

Open

Open an existing openPMD series in `data<N>.h5`. Further file formats than `.h5` (HDF5) are supported: `.bp` (ADIOS1) or `.json` (JSON).

C++11

```
auto series = api::Series(
    "data%T.h5",
    api::AccessType::READ_ONLY);
```

Python

```
series = api.Series(
    "data%T.h5",
    api.Access_Type.read_only)
```

Iteration

Grouping by an arbitrary, positive integer number `<N>` in a series. Let's take the iteration `100`:

C++11

```
auto i = series.iterations[100];
```

Python

```
i = series.iterations[100]
```

Attributes

openPMD defines a kernel of meta attributes and can always be extended with more. Let's see what we've got:

C++11

```
std::cout << "openPMD version: "
    << series.openPMD() << "\n";

if( series.containsAttribute("author") )
    std::cout << "Author: "
        << series.author() << "\n";
```

Python

```
print("openPMD version: ",
      series.openPMD)

if( series.contains_attribute("author") )
    print("Author: ",
          series.author)
```

Record

An openPMD record can be either structured (mesh) or unstructured (particles). Let's read an electric field:

C++11

```
// record
auto E = i.meshes["E"];

// record components
auto E_x = E["x"];
```

Python

```
# record
E = i.meshes["E"]

# record components
E_x = E["x"]
```

Units

Even without understanding the name "E" we can check the dimensionality of a record to understand its purpose.

C++11

```
// unit system agnostic dimension
auto E_unitDim = E.unitDimension();

// ...
// api::UnitDimension::M

// conversion to SI
double x_unit = E_x.unitSI();
```

Python

```
# unit system agnostic dimension
E_unitDim = E.unit_dimension

# ...
# api.Unit_Dimension.M

# conversion to SI
x_unit = E_x.unit_SI
```

Note: This example is not yet written :-)

In the future, units are automatically converted to a selected unit system (not yet implemented). For now, please multiply your read data (`x_data`) with `x_unit` to covert to SI, otherwise the raw, potentially awkwardly scaled data is taken.

Register Chunk

We can load record components partially and in parallel or at once. Reading small data one by one is is a performance killer for I/O. Therefore, we register all data to be loaded first and then flush it in collectively.

C++11

```
// alternatively, pass pre-allocated
std::shared_ptr< double > x_data =
    E_x.loadChunk< double >();
```

Python

```
# returns an allocated but
# undefined numpy array
x_data = E_x.load_chunk()
```

Attention: After registering a data chunk such as `x_data` for loading, it MUST NOT be modified or deleted until the `flush()` step is performed! **You must not yet access** `x_data` **!**

Flush Chunk

We now flush the registered data chunks and fill them with actual data from the I/O backend. Flushing several chunks at once allows to increase I/O performance significantly. **Only after that**, the variable `x_data` can be read, manipulated and/or deleted.

C++11

```
series.flush();
```

Python

```
series.flush()
```

Data

We can now work with the newly loaded data in `x_data`:

C++11

```
auto extent = E_x.getExtent();

std::cout << "First values in E_x "
        "of shape: ";
for( auto const& dim : extent )
    std::cout << dim << ", ";
std::cout << "\n";
```

(continues on next page)

(continued from previous page)

```
for( size_t col = 0;
     col < extent[1] && col < 5;
     ++col )
    std::cout << x_data.get()[col]
              << ", ";
std::cout << "\n";
```

Python

```
extent = E_x.shape

print(
    "First values in E_x "
    "of shape: ",
    extent)


print(x_data[0, 0, :5])
```

Close

Finally, the Series is closed when its destructor is called. Make sure to have `flush()` ed all data loads at this point, otherwise it will be called once more implicitly.

C++11

```
// destruct series object,
// e.g. when out-of-scope
```

Python

```
del series
```

2.2.3 Serial Examples

The serial API provides sequential, one-process read and write access. Most users will use this for exploration and processing of their data.

Reading

C++

```
#include <openPMD/openPMD.hpp>

#include <iostream>
#include <memory>
#include <cstddef>
```

(continues on next page)

(continued from previous page)

```
using std::cout;
using namespace openPMD;

int main()
{
    Series series = Series(
        "../samples/git-sample/data%T.h5",
        AccessType::READ_ONLY
    );
    cout << "Read a Series with openPMD standard version "
         << series.openPMD() << '\n';

    cout << "The Series contains " << series.iterations.size() << " iterations:";
    for( auto const& i : series.iterations )
        cout << "\n\t" << i.first;
    cout << '\n';

    Iteration i = series.iterations[100];
    cout << "Iteration 100 contains " << i.meshes.size() << " meshes:";
    for( auto const& m : i.meshes )
        cout << "\n\t" << m.first;
    cout << '\n';
    cout << "Iteration 100 contains " << i.particles.size() << " particle species:
↪";
    for( auto const& ps : i.particles )
        cout << "\n\t" << ps.first;
    cout << '\n';

    MeshRecordComponent E_x = i.meshes["E"]["x"];
    Extent extent = E_x.getExtent();
    cout << "Field E/x has shape (";
    for( auto const& dim : extent )
        cout << dim << ',';
    cout << ") and has datatype " << E_x.getDatatype() << '\n';

    Offset chunk_offset = {1, 1, 1};
    Extent chunk_extent = {2, 2, 1};
    auto chunk_data = E_x.loadChunk<double>(chunk_offset, chunk_extent);
    cout << "Queued the loading of a single chunk from disk, "
            "ready to execute\n";
    series.flush();
    cout << "Chunk has been read from disk\n"
         << "Read chunk contains:\n";
    for( size_t row = 0; row < chunk_extent[0]; ++row )
    {
        for( size_t col = 0; col < chunk_extent[1]; ++col )
            cout << "\t"
                 << '(' << row + chunk_offset[0] << '|' << col + chunk_offset[1] <
↪< '|' << 1 << ")\t"
                 << chunk_data.get()[row*chunk_extent[1]+col];
        cout << '\n';
    }

    auto all_data = E_x.loadChunk<double>();
    series.flush();
    cout << "Full E/x starts with:\n\t{";
    for( size_t col = 0; col < extent[1] && col < 5; ++col )
        cout << all_data.get()[col] << ", ";
    cout << "...}\n";
```

(continues on next page)

(continued from previous page)

```
    /* The files in 'series' are still open until the object is destroyed, on
     * which it cleanly flushes and closes all open file handles.
     * When running out of scope on return, the 'Series' destructor is called.
     */
    return 0;
}
```

An extended example can be found in `examples/6_dump_filebased_series.cpp`.

Python

```
import openpmd_api


if __name__ == "__main__":
    series = openpmd_api.Series("../samples/git-sample/data%T.h5",
                                openpmd_api.Access_Type.read_only)
    print("Read a Series with openPMD standard version %s" %
          series.openPMD)

    print("The Series contains {0} iterations:".format(len(series.iterations)))
    for i in series.iterations:
        print("\t {0}".format(i))
    print("")

    i = series.iterations[100]
    print("Iteration 100 contains {0} meshes:".format(len(i.meshes)))
    for m in i.meshes:
        print("\t {0}".format(m))
    print("")
    print("Iteration 100 contains {0} particle species:".format(
        len(i.particles)))
    for ps in i.particles:
        print("\t {0}".format(ps))
    print("")

    E_x = i.meshes["E"]["x"]
    shape = E_x.shape

    print("Field E.x has shape {0} and datatype {1}".format(
          shape, E_x.dtype))

    chunk_data = E_x[1:3, 1:3, 1:2]
    # print("Queued the loading of a single chunk from disk, "
    #       "ready to execute")
    series.flush()
    print("Chunk has been read from disk\n"
          "Read chunk contains:")
    print(chunk_data)
    # for row in range(2):
    #     for col in range(2):
    #         print("\t({0}|{1}|{2})\t{3}".format(
    #             row + 1, col + 1, 1, chunk_data[row*chunk_extent[1]+col])
    #         )
    #     print("")

    all_data = E_x.load_chunk()
    series.flush()
    print("Full E/x is of shape {0} and starts with:".format(all_data.shape))
```

(continues on next page)

(continued from previous page)

```
    print(all_data[0, 0, :5])

    # The files in 'series' are still open until the object is destroyed, on
    # which it cleanly flushes and closes all open file handles.
    # One can delete the object explicitly (or let it run out of scope) to
    # trigger this.
    del series
```

Writing

C++

```
#include <openPMD/openPMD.hpp>

#include <iostream>
#include <memory>
#include <numeric>
#include <cstdlib>


using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    // user input: size of matrix to write, default 3x3
    size_t size = (argc == 2 ? atoi(argv[1]) : 3);

    // matrix dataset to write with values 0...size*size-1
    std::vector<double> global_data(size*size);
    std::iota(global_data.begin(), global_data.end(), 0.);

    cout << "Set up a 2D square array (" << size << 'x' << size
         << ") that will be written\n";

    // open file for writing
    Series series = Series(
        "../samples/3_write_serial.h5",
        AccessType::CREATE
    );
    cout << "Created an empty " << series.iterationEncoding() << " Series\n";

    MeshRecordComponent rho =
      series
          .iterations[1]
          .meshes["rho"][MeshRecordComponent::SCALAR];
    cout << "Created a scalar mesh Record with all required openPMD attributes\n";

    Datatype datatype = determineDatatype(shareRaw(global_data));
    Extent extent = {size, size};
    Dataset dataset = Dataset(datatype, extent);
    cout << "Created a Dataset of size " << dataset.extent[0] << 'x' << dataset.
↪extent[1]
         << " and Datatype " << dataset.dtype << '\n';

    rho.resetDataset(dataset);
    cout << "Set the dataset properties for the scalar field rho in iteration 1\n";

    series.flush();
```

(continues on next page)

(continued from previous page)

```
    cout << "File structure and required attributes have been written\n";

    Offset offset = {0, 0};
    rho.storeChunk(shareRaw(global_data), offset, extent);
    cout << "Stored the whole Dataset contents as a single chunk, "
            "ready to write content\n";

    series.flush();
    cout << "Dataset content has been fully written\n";

    /* The files in 'series' are still open until the object is destroyed, on
     * which it cleanly flushes and closes all open file handles.
     * When running out of scope on return, the 'Series' destructor is called.
     */
    return 0;
}
```

An extended example can be found in `examples/7_extended_write_serial.cpp`.

Python

```
import openpmd_api
import numpy as np


if __name__ == "__main__":
    # user input: size of matrix to write, default 3x3
    size = 3

    # matrix dataset to write with values 0...size*size-1
    data = np.arange(size*size, dtype=np.double).reshape(3, 3)

    print("Set up a 2D square array ({0}x{1}) that will be written".format(
        size, size))

    # open file for writing
    series = openpmd_api.Series(
        "../samples/3_write_serial_py.h5",
        openpmd_api.Access_Type.create
    )

    print("Created an empty {0} Series".format(series.iteration_encoding))

    print(len(series.iterations))
    rho = series.iterations[1]. \
        meshes["rho"][openpmd_api.Mesh_Record_Component.SCALAR]

    dataset = openpmd_api.Dataset(data.dtype, data.shape)

    print("Created a Dataset of size {0}x{1} and Datatype {2}".format(
        dataset.extent[0], dataset.extent[1], dataset.dtype))

    rho.reset_dataset(dataset)
    print("Set the dataset properties for the scalar field rho in iteration 1")

    series.flush()
    print("File structure has been written")

    rho[()] = data
```

(continues on next page)

(continued from previous page)

```
    print("Stored the whole Dataset contents as a single chunk, " +
          "ready to write content")

    series.flush()
    print("Dataset content has been fully written")

    # The files in 'series' are still open until the object is destroyed, on
    # which it cleanly flushes and closes all open file handles.
    # One can delete the object explicitly (or let it run out of scope) to
    # trigger this.
    del series
```

2.2.4 Parallel Examples

The following examples show parallel reading and writing of domain-decomposed data with MPI.

The Message Passing Interface (MPI) is an open communication standard for scientific computing. MPI is used on clusters, e.g. large-scale supercomputers, to communicate between nodes and provides parallel I/O primitives.

Reading

```
#include <openPMD/openPMD.hpp>

#include <mpi.h>

#include <iostream>
#include <memory>
#include <cstddef>


using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int mpi_size;
    int mpi_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);


    /* note: this scope is intentional to destruct the openPMD::Series object
     *       prior to MPI_Finalize();
     */
    {
        Series series = Series(
            "../samples/git-sample/data%T.h5",
            AccessType::READ_ONLY,
            MPI_COMM_WORLD
        );
        if( 0 == mpi_rank )
            cout << "Read a series in parallel with " << mpi_size << " MPI ranks\n
↪";
```

(continues on next page)

(continued from previous page)

```
        MeshRecordComponent E_x = series.iterations[100].meshes["E"]["x"];

        Offset chunk_offset = {
            static_cast< long unsigned int >(mpi_rank) + 1,
            1,
            1
        };
        Extent chunk_extent = {2, 2, 1};

        auto chunk_data = E_x.loadChunk<double>(chunk_offset, chunk_extent);

        if( 0 == mpi_rank )
            cout << "Queued the loading of a single chunk per MPI rank from disk, "
                    "ready to execute\n";
        series.flush();

        if( 0 == mpi_rank )
            cout << "Chunks have been read from disk\n";

        for( int i = 0; i < mpi_size; ++i )
        {
            if( i == mpi_rank )
            {
                cout << "Rank " << mpi_rank << " - Read chunk contains:\n";
                for( size_t row = 0; row < chunk_extent[0]; ++row )
                {
                    for( size_t col = 0; col < chunk_extent[1]; ++col )
                        cout << "\t"
                                << '(' << row + chunk_offset[0] << '|' << col + chunk_
↪offset[1] << '|' << 1 << ")\t"
                                << chunk_data.get()[row*chunk_extent[1]+col];
                    cout << std::endl;
                }
            }

            // this barrier is not necessary but structures the example output
            MPI_Barrier(MPI_COMM_WORLD);
        }
    }

    // openPMD::Series MUST be destructed at this point
    MPI_Finalize();

    return 0;
}
```

Writing

```
#include <openPMD/openPMD.hpp>

#include <mpi.h>

#include <iostream>
#include <memory>
#include <vector>   // std::vector


using std::cout;
using namespace openPMD;
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int mpi_size;
    int mpi_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    /* note: this scope is intentional to destruct the openPMD::Series object
     *       prior to MPI_Finalize();
     */
    {
        // global data set to write: [MPI_Size * 10, 300]
        // each rank writes a 10x300 slice with its MPI rank as values
        float const value = float(mpi_size);
        std::vector<float> local_data(
            10 * 300, value);
        if( 0 == mpi_rank )
            cout << "Set up a 2D array with 10x300 elements per MPI rank (" << mpi_
↪size
                 << "x) that will be written to disk\n";

        // open file for writing
        Series series = Series(
            "../samples/5_parallel_write.h5",
            AccessType::CREATE,
            MPI_COMM_WORLD
        );
        if( 0 == mpi_rank )
          cout << "Created an empty series in parallel with "
               << mpi_size << " MPI ranks\n";

        MeshRecordComponent mymesh =
            series
                .iterations[1]
                .meshes["mymesh"][MeshRecordComponent::SCALAR];

        // example 1D domain decomposition in first index
        Datatype datatype = determineDatatype<float>();
        Extent global_extent = {10ul * mpi_size, 300};
        Dataset dataset = Dataset(datatype, global_extent);

        if( 0 == mpi_rank )
            cout << "Prepared a Dataset of size " << dataset.extent[0]
                 << "x" << dataset.extent[1]
                 << " and Datatype " << dataset.dtype << '\n';

        mymesh.resetDataset(dataset);
        if( 0 == mpi_rank )
            cout << "Set the global Dataset properties for the scalar field mymesh␣
↪in iteration 1\n";

        // example shows a 1D domain decomposition in first index
        Offset chunk_offset = {10ul * mpi_rank, 0};
        Extent chunk_extent = {10, 300};
        mymesh.storeChunk(local_data, chunk_offset, chunk_extent);
        if( 0 == mpi_rank )
            cout << "Registered a single chunk per MPI rank containing its␣
↪contribution, "
```

(continues on next page)

(continued from previous page)

```
                    "ready to write content to disk\n";

        series.flush();
        if( 0 == mpi_rank )
            cout << "Dataset content has been fully written to disk\n";
    }

    // openPMD::Series MUST be destructed at this point
    MPI_Finalize();

    return 0;
}
```

2.2.5 All Examples

The full list of examples is contained in our `examples/` folder:

C++

- 1_structure.cpp: creating a first series
- 2_read_serial.cpp: reading a mesh
- 3_write_serial.cpp: writing a mesh
- 4_read_parallel.cpp: MPI-parallel mesh read
- 5_write_parallel.cpp: MPI-parallel mesh write
- 6_dump_filebased_series.cpp: detailed reading with a file-based series
- 7_extended_write_serial.cpp: particle writing with patches and constant records
- 8_benchmark_parallel.cpp: a MPI-parallel IO-benchmark

Python

- 2_read_serial.py: reading a mesh
- 3_write_serial.py: writing a mesh
- 7_extended_write_serial.py: particle writing with patches and constant records
- 9_particle_write_serial.py: writing particles

Unit Tests

Our unit tests in the `test/` folder might also be informative for advanced developers.

2.3 Development

2.3.1 Contribution Guide

GitHub

The best starting point is the GitHub issue tracker.

For existing tasks, the labels good first issue and help wanted are great for contributions. In case you want to start working on one of those, just *comment* in it first so no work is duplicated.

New contributions in form of pull requests always need to go in the `dev` (development) branch. The `master` branch contains the last stable release and receives updates only when a new version is drafted.

Maintainers organize prioritites and progress in the projects tab.

Style Guide

For coding style, please try to follow the guides in ComputationalRadiationPhysics/contributing for new code.

2.3.2 Repository Structure

Branches

- `master`: the latest stable release, always tagged with a version
- `dev`: the development branch where all features start from and are merged to
- `release-X.Y.Z`: release candiate for version `X.Y.Z` with an upcoming release, receives updates for bug fixes and documentation such as change logs but usually no new features

Directory Structure

- `include/`
  - C++ header files
  - set `-I` here
  - prefixed with project name
- `src/`
  - C++ source files
- `lib/`
  - `python/`
    - modules, e.g. additional python interfaces and helpers
    - set `PYTHONPATH` here
- `examples/`
  - read and write examples
- `samples/`
  - example files; need to be added manually with: `.travis/download_samples.sh`
- `share/openPMD/`
  - `cmake/`
    - cmake scripts
  - `thirdParty/`
    - included third party software
- `test/`
  - unit tests which are run with `ctest` (`make test`)
- `.travis/`

  - setup scripts for our continuous integration systems

- `docs/`
  - documentation files

2.3.3 How to Write a Backend

Adding support for additional types of file storage or data transportation is possible by creating a backend. Backend design has been kept independent of the openPMD-specific logic that maintains all constraints within a file. This should allow easy introduction of new file formats with only little knowledge about the rest of the system.

File Formats

To get started, you should create a new file format in `include/openPMD/IO/Format.hpp` representing the new backend. Note that this enumeration value will never be seen by users of openPMD-api, but should be kept short and concise to improve readability.

```
enum class Format
{
     JSON
};
```

In order to use the file format through the API, you need to provide unique and characteristic filename extensions that are associated with it. This happens in `src/Series.cpp`:

```
Format
determineFormat(std::string const& filename)
{
    if( auxiliary::ends_with(filename, ".json") )
        return Format::JSON;
}
```

```
std::string
cleanFilename(std::string const& filename, Format f)
{
    switch( f )
    {
        case Format::JSON:
            return auxiliary::replace_last(filename, ".json", "");
    }
}
```

```
std::function< bool(std::string const&) >
matcher(std::string const& name, Format f)
{
    switch( f )
    {
        case Format::JSON:
        {
            std::regex pattern(auxiliary::replace_last(name + ".json$", "%T",
↪"[[:digit:]]+"));
            return [pattern](std::string const& filename) -> bool { return␣
↪std::regex_search(filename, pattern); };
        }
    }
}
```

Unless your file format imposes additional restrictions to the openPMD constraints, this is all you have to do in the frontend section of the API.

IO Handler

Now that the user can specify that the new backend is to be used, a concrete mechanism for handling IO interactions is required. We call this an `IOHandler`. It is not concerned with any logic or constraints enforced by openPMD, but merely offers a small set of elementary IO operations.

On the very basic level, you will need to derive a class from `AbstractIOHandler`:

```
/* file: include/openPMD/IO/JSON/JSONIOHandler.hpp */
#include "openPMD/IO/AbstractIOHandler.hpp"

namespace openPMD
{
class JSONIOHandler : public AbstractIOHandler
{
public:
    JSONIOHandler(std::string const& path, AccessType);
    virtual ~JSONIOHandler();

    std::future< void > flush() override;
}
} // openPMD
```

```
/* file: src/IO/JSON/JSONIOHandler.cpp */
#include "openPMD/IO/JSON/JSONIOHandler.hpp"

namespace openPMD
{
JSONIOHandler::JSONIOHandler(std::string const& path, AccessType at)
        : AbstractIOHandler(path, at)
{ }

JSONIOHandler::~JSONIOHandler()
{ }

std::future< void >
JSONIOHandler::flush()
{ return std::future< void >(); }
} // openPMD
```

Familiarizing your backend with the rest of the API happens in just one place in `src/IO/AbstractIOHandlerHelper.cpp`:

```
#if openPMD_HAVE_MPI
std::shared_ptr< AbstractIOHandler >
createIOHandler(
    std::string const& path,
    AccessType at,
    Format f,
    MPI_Comm comm
)
{
    switch( f )
    {
        case Format::JSON:
            std::cerr << "No MPI-aware JSON backend available. "
                         "Falling back to the serial backend! "
                         "Possible failure and degraded performance!" << std::endl;
            return std::make_shared< JSONIOHandler >(path, at);
    }
}
#endif
```

(continues on next page)

(continued from previous page)

```
std::shared_ptr< AbstractIOHandler >
createIOHandler(
    std::string const& path,
    AccessType at,
    Format f
)
{
    switch( f )
    {
        case Format::JSON:
            return std::make_shared< JSONIOHandler >(path, at);
    }
}
```

In this state, the backend will do no IO operations and just act as a dummy that ignores all queries.

IO Task Queue

Operations between the logical representation in this API and physical storage are funneled through a queue `m_work` that is contained in the newly created IOHandler. Contained in this queue are `IOTask` s that have to be processed in FIFO order (unless you can prove sequential execution guarantees for out-of-order execution) when `AbstractIOHandler::flush()` is called. A **recommended** skeleton is provided in `AbstractIOHandlerImpl`. Note that emptying the queue this way is not required and might not fit your IO scheme.

Using the provided skeleton involves

- deriving an IOHandlerImpl for your IOHandler and
- delegating all flush calls to the IOHandlerImpl:

```
/* file: include/openPMD/IO/JSON/JSONIOHandlerImpl.hpp */
#include "openPMD/IO/AbstractIOHandlerImpl.hpp"

namespace openPMD
{
class JSONIOHandlerImpl : public AbstractIOHandlerImpl
{
public:
    JSONIOHandlerImpl(AbstractIOHandler*);
    virtual ~JSONIOHandlerImpl();

    void createFile(Writable*, Parameter< Operation::CREATE_FILE > const&)␣
↪override;
    void createPath(Writable*, Parameter< Operation::CREATE_PATH > const&)␣
↪override;
    void createDataset(Writable*, Parameter< Operation::CREATE_DATASET > const&)␣
↪override;
    void extendDataset(Writable*, Parameter< Operation::EXTEND_DATASET > const&)␣
↪override;
    void openFile(Writable*, Parameter< Operation::OPEN_FILE > const&) override;
    void openPath(Writable*, Parameter< Operation::OPEN_PATH > const&) override;
    void openDataset(Writable*, Parameter< Operation::OPEN_DATASET > &) override;
    void deleteFile(Writable*, Parameter< Operation::DELETE_FILE > const&)␣
↪override;
    void deletePath(Writable*, Parameter< Operation::DELETE_PATH > const&)␣
↪override;
    void deleteDataset(Writable*, Parameter< Operation::DELETE_DATASET > const&)␣
↪override;
    void deleteAttribute(Writable*, Parameter< Operation::DELETE_ATT > const&)␣
↪override;
```

(continues on next page)

(continued from previous page)

```
    void writeDataset(Writable*, Parameter< Operation::WRITE_DATASET > const&)␣
↪override;
    void writeAttribute(Writable*, Parameter< Operation::WRITE_ATT > const&)␣
↪override;
    void readDataset(Writable*, Parameter< Operation::READ_DATASET > &) override;
    void readAttribute(Writable*, Parameter< Operation::READ_ATT > &) override;
    void listPaths(Writable*, Parameter< Operation::LIST_PATHS > &) override;
    void listDatasets(Writable*, Parameter< Operation::LIST_DATASETS > &) override;
    void listAttributes(Writable*, Parameter< Operation::LIST_ATTS > &) override;
}
} // openPMD
```

```
/* file: include/openPMD/IO/JSON/JSONIOHandler.hpp */
#include "openPMD/IO/AbstractIOHandler.hpp"
#include "openPMD/IO/JSON/JSONIOHandlerImpl.hpp"

namespace openPMD
{
class JSONIOHandler : public AbstractIOHandler
{
public:
    /* ... */
private:
    JSONIOHandlerImpl m_impl;
}
} // openPMD
```

```
/* file: src/IO/JSON/JSONIOHandler.cpp */
#include "openPMD/IO/JSON/JSONIOHandler.hpp"

namespace openPMD
{
/*...*/
std::future< void >
JSONIOHandler::flush()
{
    return m_impl->flush();
}
} // openPMD
```

Each IOTask contains a pointer to a `Writable` that corresponds to one object in the openPMD hierarchy. This object may be a group or a dataset. When processing certain types of IOTasks in the queue, you will have to assign unique FilePositions to these objects to identify the logical object in your physical storage. For this, you need to derive a concrete FilePosition for your backend from `AbstractFilePosition`. There is no requirement on how to identify your objects, but ids from your IO library and positional strings are good candidates.

```
/* file: include/openPMD/IO/JSON/JSONFilePosition.hpp */
#include "openPMD/IO/AbstractFilePosition.hpp"

namespace openPMD
{
struct JSONFilePosition : public AbstractFilePosition
{
    JSONFilePosition(uint64_t id)
        : id{id}
    { }

    uint64_t id;
};
} // openPMD
```

From this point, all that is left to do is implement the elementary IO operations provided in the IOHandlerImpl. The `Parameter` structs contain both input parameters (from storage to API) and output parameters (from API to storage). The easy way to distinguish between the two parameter sets is their C++ type: Input parameters are `std::shared_ptr` s that allow you to pass the requested data to their destination. Output parameters are all objects that are *not* `std::shared_ptr` s. The contract of each function call is outlined in `include/openPMD/IO/AbstractIOHandlerImpl`.

```
/* file: src/IO/JSON/JSONIOHandlerImpl.cpp */
#include "openPMD/IO/JSONIOHandlerImpl.hpp"

namespace openPMD
{
void
JSONIOHandlerImpl::createFile(Writable* writable,
                              Parameter< Operation::CREATE_FILE > const&␣
↪parameters)
{
    if( !writable->written )
    {
        path dir(m_handler->directory);
        if( !exists(dir) )
            create_directories(dir);

        std::string name = m_handler->directory + parameters.name;
        if( !auxiliary::ends_with(name, ".json") )
            name += ".json";

        uint64_t id = /*...*/
        VERIFY(id >= 0, "Internal error: Failed to create JSON file");

        writable->written = true;
        writable->abstractFilePosition = std::make_shared< JSONFilePosition >(id);
    }
}
/*...*/
} // openPMD
```

Note that you might have to keep track of open file handles if they have to be closed explicitly during destruction of the IOHandlerImpl (prominent in C-style frameworks).

2.3.4 Build Dependencies

Section author: Axel Huebl

`openPMD-api` depends on a series of third-party projects. These are currently:

Required

- CMake 3.11.0+
- C++11 capable compiler, e.g. g++ 4.8+, clang 3.9+, VS 2015+

Shipped internally

The following libraries are shipped internally in `share/openPMD/thirdParty/` for convenience:

- MPark.Variant 1.4.0+ (BSL-1.0)
- Catch2 2.6.1+ (BSL-1.0)
- pybind11 2.2.4+ (new BSD)

- NLohmann-JSON 3.5.0+ (MIT)

Optional: I/O backends

- JSON
- HDF5 1.8.13+
- ADIOS1 1.13.1+
- ADIOS2 2.1+ (*not yet implemented*)

while those can be build either with or without:

- MPI 2.1+, e.g. OpenMPI 1.6.5+ or MPICH2

Optional: language bindings

- Python:
  - Python 3.5 - 3.7
  - pybind11 2.2.4+
  - numpy 1.15+
  - mpi4py 2.1+

2.3.5 Build Options

Section author: Axel Huebl

Variants

The following options can be added to the `cmake` call to control features. CMake controls options with prefixed `-D`, e.g. `-DopenPMD_USE_MPI=OFF`:

CMake Option	Values	Description
`openPMD_USE_MPI`	**AUTO**/ON/OFF	Parallel, Multi-Node I/O for clusters
`openPMD_USE_JSON`	**AUTO**/ON/OFF	JSON backend (`.json` files)
`openPMD_USE_HDF5`	**AUTO**/ON/OFF	HDF5 backend (`.h5` files)
`openPMD_USE_ADIOS1`	**AUTO**/ON/OFF	ADIOS1 backend (`.bp` files)
`openPMD_USE_ADIOS2`	AUTO/ON/**OFF**	ADIOS2 backend (`.bp` files) [1]
`openPMD_USE_PYTHON`	**AUTO**/ON/OFF	Enable Python bindings
`openPMD_USE_INVASIVE_TESTS`	ON/**OFF**	Enable unit tests that modify source code [2]
`openPMD_USE_VERIFY`	**ON**/OFF	Enable internal VERIFY (assert) macro independent of build type [3]
`PYTHON_EXECUTABLE`	(first found)	Path to Python executable

[1] *not yet implemented*

[2] e.g. changes C++ visibility keywords, breaks MSVC

[3] this includes most pre-/post-condition checks, disabling without specific cause is highly discouraged

Shared or Static

By default, we will build as a static library and install also its headers. You can only build a static (`libopenPMD.a` or `openPMD.lib`) or a shared library (`libopenPMD.so` or `openPMD.dll`) at a time.

The following options can be tried to switch between static and shared builds and control if dependencies are linked dynamically or statically:

CMake Option	Values	Description
`BUILD_SHARED_LIBS`	ON/**OFF**	Build the C++ API as shared library
`HDF5_USE_STATIC_LIBRARIES`	ON/**OFF**	Require static HDF5 library
`ADIOS_USE_STATIC_LIBS`	ON/**OFF**	Require static ADIOS1 library

Note that python modules (`openpmd_api.cpython.[...].so` or `openpmd_api.pyd`) are always dynamic libraries. Therefore, if you want to build the python module and prefer static dependencies, make sure to provide all of dependencies build with position independent code (`-fPIC`). The same requirement is true if you want to build a *shared* C++ API library with *static* dependencies.

Debug

By default, the `Release` version is built. In order to build with debug symbols, pass `-DCMAKE_BUILD_TYPE=Debug` to your `cmake` command.

Shipped Dependencies

Additionally, the following libraries are shipped internally for convenience. These might get installed in your *CMAKE_INSTALL_PREFIX* if the option is `ON`.

The following options allow to switch to external installs of dependencies:

CMake Option	Values	Installs	Library	Version
`openPMD_USE_INTERNAL_VARIANT`	**ON**/OFF	Yes	MPark.Variant	1.4.0+
`openPMD_USE_INTERNAL_CATCH`	**ON**/OFF	No	Catch2	2.6.1+
`openPMD_USE_INTERNAL_PYBIND11`	**ON**/OFF	No	pybind11	2.2.4+
`openPMD_USE_INTERNAL_JSON`	**ON**/OFF	No	NLohmann-JSON	3.5.0+

Tests

By default, tests and examples are built. In order to skip building those, pass `-DBUILD_TESTING=OFF` or `-DBUILD_EXAMPLES=OFF` to your `cmake` command.

2.3.6 Sphinx

Section author: Axel Huebl

In the following section we explain how to contribute to this documentation.

If you are reading the HTML version on http://openPMD-api.readthedocs.io and want to improve or correct existing pages, check the "Edit on GitHub" link on the right upper corner of each document.

Alternatively, go to *docs/source* in our source code and follow the directory structure of reStructuredText (`.rst`) files there. For intrusive changes, like structural changes to chapters, please open an issue to discuss them beforehand.

Build Locally

This document is build based on free open-source software, namely Sphinx, Doxygen (C++ APIs as XML) and Breathe (to include doxygen XML in Sphinx). A web-version is hosted on ReadTheDocs.

The following requirements need to be installed (once) to build our documentation successfully:

```
cd docs/

# doxygen is not shipped via pip, install it externally,
# from the homepage, your package manager, conda, etc.
# example:
sudo apt-get install doxygen

# python tools & style theme
pip install -r requirements.txt # --user
```

With all documentation-related software successfully installed, just run the following commands to build your docs locally. Please check your documentation build is successful and renders as you expected before opening a pull request!

```
# skip this if you are still in docs/
cd docs/

# parse the C++ API documentation,
#   enjoy the doxygen warnings!
doxygen
# render the `.rst` files and replace their macros within
#   enjoy the breathe errors on things it does not understand from doxygen :)
make html

# open it, e.g. with firefox :)
firefox build/html/index.html

# now again for the pdf :)
make latexpdf

# open it, e.g. with okular
build/latex/openPMD-api.pdf
```

Useful Links

- A primer on writing restFUL files for sphinx
- Why You Shouldn't Use "Markdown" for Documentation
- Markdown Limitations in Sphinx

2.3.7 Doxygen

Section author: Axel Huebl

An online version of our Doxygen build can be found at

http://www.openPMD.org/openPMD-api/

We regularly update it via

```
git checkout gh-pages

# optional argument: branch or tag name
```

(continues on next page)

(continued from previous page)

```
./update.sh

git commit -a
git push
```

This section explains what is done when this script is run to build it manually.

Requirements

First, install Doxygen and its dependencies for graph generation.

```
# install requirements (Debian/Ubuntu)
sudo apt-get install doxygen graphviz

# enable HTML output in our Doxyfile
sed -i 's/GENERATE_HTML.*=.*NO/GENERATE_HTML     = YES/' docs/Doxyfile
```

Build

Now run the following commands to build the Doxygen HTML documentation locally.

```
cd docs/

# build the doxygen HTML documentation
doxygen

# open the generated HTML pages, e.g. with firefox
firefox html/index.html
```

2.3.8 Release Channels

Section author: Axel Huebl

Spack

Our recommended HPC release channel when in need for MPI. Also very useful for Linux and OSX desktop releases.

Example workflow for a new release:

https://github.com/spack/spack/pull/9178

[TODO: show how to add a tag as version; please CC @ax3l on updates]

Conda-Forge

Our primary release channel for desktops, fully automated binary distribution. Supports Windows, OSX and Linux. Packages are built without MPI.

Example workflow for a new release:

https://github.com/conda-forge/openpmd-api-feedstock/pull/7

PyPI

On PyPI, we only upload a source page with all settings to default / `AUTO` and proper `RPATH` settings for internal libraries.

PyPI releases are experimental and not highly recommended for the average user. They do come handy to test pre-releases quickly with power-users.

```
# 1. check out the git tag you want to release
# 2. verify the version in setup.py is correct (PEP-0440),
#    e.g. `<v>.dev[N]`, `<v>a[N]`, `<v>b[N]`, `<v>rc[N]`, or `<v>`

# prepare source distribution
python setup.py sdist

# GPG sign and upload
#   note: have up-to-date tools!
#   https://packaging.python.org/guides/making-a-pypi-friendly-readme/
twine upload -s dist/*

# verify everything is updated as expected on
# https://pypi.org/project/openPMD-api/
```

ReadTheDocs

Before a new version can be tagged in our manual, at least one commit must go to the mainline repo. (For some reason, pushing the tag alone does not trigger a webhook update on RTD.)

Then, activate the new version in Projects - openPMD-api - Versions which triggers its build.

And after the new version was built, and if this version was not a backport to an older release series, set the new default version in Admin - Versions.

Doxygen

In order to update the Doxygen C++ API docs, do:

```
# assuming a clean source tree
git checkout gh-pages

# stash anything that the regular branches have in `.gitignore`
git stash --include-untracked

# optional first argument is branch/tag on mainline repo, default: dev
./update.sh
git commit -a
git push

# go back
git checkout -
git stash pop
```

2.4 Backends

2.4.1 JSON Backend

openPMD supports writing to and reading from JSON files. For this, the installed copy of openPMD must have been built with support for the JSON backend. To build openPMD with support for JSON, use the CMake option

`-DopenPMD_USE_JSON=ON`. For further information, check out the *installation guide*, *build dependencies* and the *build options*.

JSON File Format

A JSON file uses the file ending `.json`. The JSON backend is chosen by creating a `Series` object with a filename that has this file ending.

The top-level JSON object is a group representing the openPMD root group `"/"`. Any **openPMD group** is represented in JSON as a JSON object with two reserved keys:

- `attributes`: Attributes associated with the group. This key may be null or not be present at all, thus indicating a group without attributes.
- `platform_byte_widths` (root group only): Byte widths specific to the writing platform. Will be overwritten every time that a JSON value is stored to disk, hence this information is only available about the last platform writing the JSON value.

All datasets and subgroups contained in this group are represented as a further key of the group object. `attributes` and `platform_byte_widths` have hence the character of reserved keywords and cannot be used for group and dataset names when working with the JSON backend. Datasets and groups have the same namespace, meaning that there may not be a subgroup and a dataset with the same name contained in one group.

Any **openPMD dataset** is a JSON object with three keys:

- `attributes`: Attributes associated with the dataset. May be `null` or not present if no attributes are associated with the dataset.
- `datatype`: A string describing the type of the stored data.
- `data` A nested array storing the actual data in row-major manner. The data needs to be consistent with the fields `datatype` and `extent`. Checking whether this key points to an array can be (and is internally) used to distinguish groups from datasets.

Attributes are stored as a JSON object with a key for each attribute. Every such attribute is itself a JSON object with two keys:

- `datatype`: A string describing the type of the value.
- `value`: The actual value of type `datatype`.

Restrictions

For creation of JSON serializations (i.e. writing), the restrictions of the JSON backend are equivalent to those of the JSON library by Niels Lohmann used by the openPMD backend.

Numerical values, integral as well as floating point, are supported up to a length of 64 bits. Since JSON does not support special floating point values (i.e. NaN, Infinity, -Infinity), those values are rendered as `null`.

Instructing openPMD to write values of a datatype that is too wide for the JSON backend does *not* result in an error:

- If casting the value to the widest supported datatype of the same category (integer or floating point) is possible without data loss, the cast is performed and the value is written. As an example, on a platform with `sizeof(double) == 8`, writing the value `static_cast<long double>(std::numeric_limits<double>::max())` will work as expected since it can be cast back to `double`.
- Otherwise, a `null` value is written.

Upon reading `null` when expecting a floating point number, a NaN value will be returned. Take notice that a NaN value returned from the deserialization process may have originally been +/-Infinity or beyond the supported value range.

Upon reading `null` when expecting any other datatype, the JSON backend will propagate the exception thrown by Niels Lohmann's library.

The (keys) names `"attributes"`, `"data"` and `"datatype"` are reserved and must not be used for base/mesh/particles path, records and their components.

A parallel (i.e. MPI) implementation is *not* available.

Example

The example code in the *usage section* will produce the following JSON serialization when picking the JSON backend:

```
{
  "attributes": {
    "basePath": {
      "datatype": "STRING",
      "value": "/data/%T/"
    },
    "iterationEncoding": {
      "datatype": "STRING",
      "value": "groupBased"
    },
    "iterationFormat": {
      "datatype": "STRING",
      "value": "/data/%T/"
    },
    "meshesPath": {
      "datatype": "STRING",
      "value": "meshes/"
    },
    "openPMD": {
      "datatype": "STRING",
      "value": "1.1.0"
    },
    "openPMDextension": {
      "datatype": "UINT",
      "value": 0
    }
  },
  "data": {
    "1": {
      "attributes": {
        "dt": {
          "datatype": "DOUBLE",
          "value": 1
        },
        "time": {
          "datatype": "DOUBLE",
          "value": 0
        },
        "timeUnitSI": {
          "datatype": "DOUBLE",
          "value": 1
        }
      },
      "meshes": {
        "rho": {
          "attributes": {
            "axisLabels": {
              "datatype": "VEC_STRING",
              "value": [
```

(continues on next page)

(continued from previous page)


```
            "x"
          ]
        },
        "dataOrder": {
          "datatype": "STRING",
          "value": "C"
        },
        "geometry": {
          "datatype": "STRING",
          "value": "cartesian"
        },
        "gridGlobalOffset": {
          "datatype": "VEC_DOUBLE",
          "value": [
            0
          ]
        },
        "gridSpacing": {
          "datatype": "VEC_DOUBLE",
          "value": [
            1
          ]
        },
        "gridUnitSI": {
          "datatype": "DOUBLE",
          "value": 1
        },
        "position": {
          "datatype": "VEC_DOUBLE",
          "value": [
            0
          ]
        },
        "timeOffset": {
          "datatype": "FLOAT",
          "value": 0
        },
        "unitDimension": {
          "datatype": "ARR_DBL_7",
          "value": [
            0,
            0,
            0,
            0,
            0,
            0,
            0
          ]
        },
        "unitSI": {
          "datatype": "DOUBLE",
          "value": 1
        }
      },
      "data": [
        [
          0,
          1,
          2
        ],
        [
```


(continues on next page)

(continued from previous page)

```
                3,
                4,
                5
              ],
              [
                6,
                7,
                8
              ]
            ],
            "datatype": "DOUBLE"
          }
        }
      }
    },
    "platform_byte_widths": {
      "BOOL": 1,
      "CHAR": 1,
      "DOUBLE": 8,
      "FLOAT": 4,
      "INT": 4,
      "LONG": 8,
      "LONGLONG": 8,
      "LONG_DOUBLE": 16,
      "SHORT": 2,
      "UCHAR": 1,
      "UINT": 4,
      "ULONG": 8,
      "ULONGLONG": 8,
      "USHORT": 2
    }
}
```

2.4.2 ADIOS1 Backend

openPMD supports writing to and reading from ADIOS1 `.bp` files. For this, the installed copy of openPMD must have been built with support for the ADIOS1 backend. To build openPMD with support for ADIOS, use the CMake option `-DopenPMD_USE_ADIOS1=ON`. For further information, check out the *installation guide*, *build dependencies* and the *build options*.

I/O Method

ADIOS1 has several staging methods for alternative file formats, yet natively writes to `.bp` files. We currently implement the `MPI_AGGREGATE` transport method for MPI-parallel write (`POSIX` for serial write) and `ADIOS_READ_METHOD_BP` for read.

Backend-Specific Controls

The following environment variables control ADIOS1 I/O behavior at runtime. Fine-tuning these is especially useful when running at large scale.

environment variable	default	description
`OPENPMD_ADIOS_NUM_AGGREGATORS`		Number of I/O aggregator nodes for ADIOS1 `MPI_AGGREGATE` transport method.
`OPENPMD_ADIOS_NUM_OST`	`0`	Number of I/O OSTs for ADIOS1 `MPI_AGGREGATE` transport method.
`OPENPMD_ADIOS_HAVE_METADATA_FILE`	`1`	Online creation of the adios journal file (`1`: yes, `0`: no).

Please refer to the ADIOS1 manual, section 6.1.5 for details.

Best Practice at Large Scale

A good practice at scale is to disable the online creation of the metadata file. After writing the data, run `bpmeta` on the (to-be-created) filename to generate the metadata file offline (repeat per iteration for file-based encoding). This metadata file is needed for reading, while the actual heavy data resides in `<metadata filename>.dir/` directories.

Further options depend heavily on filesystem type, specific file striping, network infrastructure and available RAM on the aggregator nodes. If your filesystem exposes explicit object-storage-targets (OSTs), such as Lustre, try to set the number of OSTs to the maximum number available and allowed per job (e.g. non-full), assuming the number of writing MPI ranks is larger. A good number for aggregators is usually the number of contributing nodes divided by four.

For fine-tuning at extreme scale or for exotic systems, please refer to the ADIOS1 manual and talk to your filesystem admins and the ADIOS1 authors. Be aware that extreme-sale I/O is a research topic after all.

Selected References

- Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. *Datastager: scalable data staging services for petascale applications,* Cluster Computing, 13(3):277–290, 2010. DOI:10.1007/s10586-010-0135-6
- Ciprian Docan, Manish Parashar, and Scott Klasky. *DataSpaces: An interaction and coordination framework or coupled simulation workflows,* In Proc. of 19th International Symposium on High Performance and Distributed Computing (HPDC'10), June 2010. DOI:10.1007/s10586-011-0162-y
- Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. *Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks,* Concurrency and Computation: Practice and Experience, 26(7):1453–1473, 2014. DOI:10.1002/cpe.3125
- Robert McLay, Doug James, Si Liu, John Cazes, and William Barth. *A user-friendly approach for tuning parallel file operations,* In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14, pages 229–236, IEEE Press, 2014. DOI:10.1109/SC.2014.24
- Axel Huebl, Rene Widera, Felix Schmitt, Alexander Matthes, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, and Michael Bussmann. *On the Scalability of Data Reduction Techniques in Current and Upcoming HPC Systems from an Application Perspective,* ISC High Performance 2017: High Performance Computing, pp. 15-29, 2017. arXiv:1706.00522, DOI:10.1007/978-3-319-67630-2_2

2.5 Utilities

2.5.1 Benchmark

The openPMD API provides utilities to quickly configure and run benchmarks in a flexible fashion. The starting point for configuring and running benchmarks is the class template

`Benchmark<DatasetFillerProvider>`.

```
#include "openPMD/benchmark/mpi/Benchmark.hpp"
```

An object of this class template allows to preconfigure a number of benchmark runs to execute, each run specified by:

- The compression configuration, consisting itself of the compression string and the compression level.
- The backend to use, specified by the filename extension (e.g. "h5", "bp", "json", ...).
- The type of data to write, specified by the openPMD datatype.
- The number of ranks to use, not greater than the MPI size. An overloaded version of `addConfiguration()` exists that picks the MPI size.
- The number *n* of iterations. The benchmark will effectively be repeated *n* times.

The benchmark object is globally (i.e. by its constructor) specified by:

- The base path to use. This will be extended with the chosen backend's filename extension. Benchmarks might overwrite each others' files.
- The total extent of the dataset across all MPI ranks.
- The `BlockSlicer`, i.e. an object telling each rank which portion of the dataset to write to and read from. Most users will be content with the implementation provided by `OneDimensionalBlockSlicer` that will simply divide the dataset into hyperslabs along one dimension, default = 0. This implementation can also deal with odd dimensions that are not divisible by the MPI size.
- A `DatasetFillerProvider`. `DatasetFiller<T>` is an abstract class template whose job is to create the write data of type `T` for one run of the benchmark. Since one Benchmark object allows to use several datatypes, a `DatasetFillerProvider` is needed to create such objects. `DatasetFillerProvider` is a template parameter of the benchmark class template and should be a templated functor whose `operator()<T>()` returns a `shared_ptr<DatasetFiller<T>>` (or a value that can be dynamically casted to it). For users seeking to only run the benchmark with one datatype, the class template `SimpleDatasetFillerProvider<DF>` will lift a `DatasetFiller<T>` to a `DatasetFillerProvider` whose `operator()<T'>()` will only successfully return if `T` and `T'` are the same type.
- The MPI Communicator.

The class template `RandomDatasetFiller<Distr, T>` (where by default `T = typename Distr::result_type`) provides an implementation of the `DatasetFiller<T>` that lifts a random distribution to a `DatasetFiller`. The general interface of a `DatasetFiller<T>` is kept simple, but an implementation should make sure that every call to `DatasetFiller<T>::produceData()` takes roughly the same amount of time, thus allowing to deduct from the benchmark results the time needed for producing data.

The configured benchmarks are run one after another by calling the method `Benchmark<...>::runBenchmark<Clock>(int rootThread)`. The Clock template parameter should meet the requirements of a trivial clock. Although every rank will return a `BenchmarkReport<typename Clock::rep>`, only the report of the previously specified root rank will be populated with data, i.e. all ranks' data will be collected into one report.

Example Usage

```
#include <openPMD/openPMD.hpp>
#include <openPMD/benchmark/mpi/MPIBenchmark.hpp>
#include <openPMD/benchmark/mpi/RandomDatasetFiller.hpp>
#include <openPMD/benchmark/mpi/OneDimensionalBlockSlicer.hpp>

#if openPMD_HAVE_MPI
#   include <mpi.h>
#endif
```

(continues on next page)

(continued from previous page)

```
#include <iostream>


#if openPMD_HAVE_MPI
int main(
    int argc,
    char *argv[]
)
{
    using namespace std;
    MPI_Init(
        &argc,
        &argv
    );

    // For simplicity, use only one datatype in this benchmark.
    // Note that a single Benchmark object can be used to configure
    // multiple different benchmark runs with different datatypes,
    // given that you provide it with an appropriate DatasetFillerProvider
    // (template parameter of the Benchmark class).
    using type = long int;
#if openPMD_HAVE_ADIOS1 || openPMD_HAVE_HDF5
    openPMD::Datatype dt = openPMD::determineDatatype<type>();
#endif

    // Total (in this case 4D) dataset across all MPI ranks.
    // Will be the same for all configured benchmarks.
    openPMD::Extent total{
        100,
        100,
        100,
        10
    };

    // The blockslicer assigns to each rank its part of the dataset. The rank will
    // write to and read from that part. OneDimensionalBlockSlicer is a simple
    // implementation of the BlockSlicer abstract class that will divide the
    // dataset into hyperslab along one given dimension.
    // If you wish to partition your dataset in a different manner, you can
    // replace this with your own implementation of BlockSlicer.
    auto blockSlicer = std::make_shared<openPMD::OneDimensionalBlockSlicer>(0);

    // Set up the DatasetFiller. The benchmarks will later inquire the␣
↪DatasetFiller
    // to get data for writing.
    std::uniform_int_distribution<type> distr(
        0,
        200000000
    );
    openPMD::RandomDatasetFiller<decltype(distr)> df{distr};

    // The Benchmark class will in principle allow a user to configure
    // runs that write and read different datatypes.
    // For this, the class is templated with a type called DatasetFillerProvider.
    // This class serves as a factory for DatasetFillers for concrete types and
    // should have a templated operator()<T>() returning a value
    // that can be dynamically casted to a std::shared_ptr<openPMD::DatasetFiller
↪<T>>
    // The openPMD API provides only one implementation of a DatasetFillerProvider,
    // namely the SimpleDatasetFillerProvider being used in this example.
```

(continues on next page)

(continued from previous page)

```
    // Its purpose is to leverage a DatasetFiller for a concrete type (df in this␣
↪example)
    // to a DatasetFillerProvider whose operator()<T>() will fail during runtime␣
↪if T does
    // not correspond with the underlying DatasetFiller.
    // Use this implementation if you only wish to run the benchmark for one␣
↪Datatype,
    // otherwise provide your own implementation of DatasetFillerProvider.
    openPMD::SimpleDatasetFillerProvider<decltype(df)> dfp{df};

    // Create the Benchmark object. The file name (first argument) will be extended
    // with the backends' file extensions.
    openPMD::MPIBenchmark<decltype(dfp)> benchmark{
        "../benchmarks/benchmark",
        total,
        std::dynamic_pointer_cast<openPMD::BlockSlicer>(blockSlicer),
        dfp,
    };

    // Add benchmark runs to be executed. This will only store the configuration␣
↪and not
    // run the benchmark yet. Each run is configured by:
    // * The compression scheme to use (first two parameters). The first parameter␣
↪chooses
    //   the compression scheme, the second parameter is the compression level.
    // * The backend (by file extension).
    // * The datatype to use for this run.
    // * The number of iterations. Effectively, the benchmark will be repeated for␣
↪this many
    //   times.
#if openPMD_HAVE_ADIOS1
    benchmark.addConfiguration("", 0, "bp", dt, 10);
#endif
#if openPMD_HAVE_HDF5
    benchmark.addConfiguration("", 0, "h5", dt, 10);
#endif

    // Execute all previously configured benchmarks. Will return a␣
↪MPIBenchmarkReport object
    // with write and read times for each configured run.
    // Take notice that results will be collected into the root rank's report␣
↪object, the other
    // ranks' reports will be empty. The root rank is specified by the first␣
↪parameter of runBenchmark,
    // the default being 0.
    auto res =
        benchmark.runBenchmark<std::chrono::high_resolution_clock>();

    int rank;
    MPI_Comm_rank(
        MPI_COMM_WORLD,
        &rank
    );
    if( rank == 0 )
    {
        for( auto it = res.durations.begin();
             it != res.durations.end();
             it++ )
        {
            auto time = it->second;
            std::cout << "on rank " << std::get<res.RANK>(it->first)
```

(continues on next page)

(continued from previous page)

```
                        << "\t with backend "
                        << std::get<res.BACKEND>(it->first)
                        << "\twrite time: "
                        << std::chrono::duration_cast<std::chrono::milliseconds>(
                            time.first
                        ).count() << "\tread time: "
                        << std::chrono::duration_cast<std::chrono::milliseconds>(
                            time.second
                        ).count() << std::endl;
        }
    }

    MPI_Finalize();
}
#else
int main(void)
{
    return 0;
}
#endif
```