



openPMD-api Documentation

Release 0.13.0

The openPMD Community

Jan 04, 2021

CONTENTS

1	Supported openPMD Standard Versions	3
1.1	Code of Conduct	3
1.1.1	Our Pledge	3
1.1.2	Our Standards	3
1.1.3	Our Responsibilities	4
1.1.4	Scope	4
1.1.5	Enforcement	4
1.1.6	Attribution	4
1.2	Citation	4
1.2.1	openPMD-standard	4
1.2.2	openPMD-api	5
2	Installation	7
2.1	Installation	7
2.1.1	Using the Spack Package	7
2.1.2	Using the Conda Package	7
2.1.3	Using the Brew Package	7
2.1.4	Using the PyPI Package	8
2.1.5	From Source with CMake	8
2.2	Changelog	9
2.2.1	0.13.0	9
2.2.2	0.12.0-alpha	12
2.2.3	0.11.1-alpha	13
2.2.4	0.11.0-alpha	14
2.2.5	0.10.3-alpha	15
2.2.6	0.10.2-alpha	16
2.2.7	0.10.1-alpha	16
2.2.8	0.10.0-alpha	17
2.2.9	0.9.0-alpha	18
2.2.10	0.8.0-alpha	20
2.2.11	0.7.1-alpha	21
2.2.12	0.7.0-alpha	21
2.2.13	0.6.3-alpha	22
2.2.14	0.6.2-alpha	23
2.2.15	0.6.1-alpha	23
2.2.16	0.6.0-alpha	23
2.2.17	0.5.0-alpha	24
2.2.18	0.4.0-alpha	25
2.2.19	0.3.1-alpha	26
2.2.20	0.3.0-alpha	27
2.2.21	0.2.0-alpha	28
2.2.22	0.1.1-alpha	28
2.2.23	0.1.0-alpha	29
2.3	Upgrade Guide	29

2.3.1	0.13.0	29
2.3.2	0.12.0-alpha	30
2.3.3	0.11.0-alpha	30
2.3.4	0.10.0-alpha	31
2.3.5	0.9.0-alpha	31
2.3.6	0.7.0-alpha	31
3	Usage	35
3.1	First Write	35
3.1.1	Include / Import	35
3.1.2	Open	35
3.1.3	Iteration	36
3.1.4	Attributes	36
3.1.5	Data	37
3.1.6	Record	37
3.1.7	Units	38
3.1.8	Register Chunk	39
3.1.9	Flush Chunk	39
3.1.10	Close	40
3.2	First Read	40
3.2.1	Include / Import	40
3.2.2	Open	41
3.2.3	Iteration	41
3.2.4	Attributes	41
3.2.5	Record	42
3.2.6	Units	42
3.2.7	Register Chunk	43
3.2.8	Flush Chunk	44
3.2.9	Data	44
3.2.10	Close	45
3.3	Serial Examples	45
3.3.1	Reading	45
3.3.2	Writing	48
3.4	Parallel Examples	50
3.4.1	Reading	50
3.4.2	Writing	52
3.5	Streaming	55
3.5.1	Reading	55
3.5.2	Writing	58
3.6	Benchmarks	60
3.6.1	Parallel benchmarks 8a & 8b	60
3.6.2	Benchmark Utilities	62
3.7	All Examples	62
3.7.1	C++	62
3.7.2	Python	62
3.7.3	Unit Tests	63
4	API Details	65
4.1	C++	65
4.1.1	Public Headers	65
4.1.2	External Documentation	65
4.2	Python	65
4.2.1	Public Headers	65
4.3	MPI	66
4.3.1	Collective Behavior	66
4.3.2	Efficient Parallel I/O Patterns	66
4.4	Backend-Specific Configuration	67
4.4.1	Configuration Structure per Backend	67

5	Utilities	69
5.1	Command Line Tools	69
5.1.1	openpmd-ls	69
5.2	Benchmark	69
5.2.1	Example Usage	70
6	Backends	73
6.1	Overview	73
6.1.1	Selected References	73
6.2	JSON	73
6.2.1	JSON File Format	74
6.2.2	Restrictions	74
6.2.3	Example	75
6.3	ADIOS1	77
6.3.1	I/O Method	77
6.3.2	Backend-Specific Controls	77
6.3.3	Best Practice at Large Scale	78
6.3.4	Limitations	78
6.3.5	Selected References	79
6.4	ADIOS2	79
6.4.1	I/O Method	79
6.4.2	Steps	79
6.4.3	Backend-Specific Controls	80
6.4.4	Best Practice at Large Scale	80
6.4.5	Selected References	80
6.5	HDF5	81
6.5.1	I/O Method	81
6.5.2	Backend-Specific Controls	81
6.5.3	Selected References	82
7	Development	83
7.1	Contribution Guide	83
7.1.1	GitHub	83
7.1.2	Style Guide	83
7.2	Repository Structure	83
7.2.1	Branches	83
7.2.2	Directory Structure	83
7.3	Design Overview	84
7.3.1	Backend	85
7.3.2	I/O-Queue	85
7.3.3	Frontend	86
7.4	How to Write a Backend	87
7.4.1	File Formats	87
7.4.2	IO Handler	88
7.4.3	IO Task Queue	89
7.5	Build Dependencies	91
7.5.1	Required	92
7.5.2	Shipped internally	92
7.5.3	Optional: I/O backends	92
7.5.4	Optional: language bindings	92
7.5.5	Quick Install with Spack	92
7.6	Build Options	93
7.6.1	Variants	93
7.6.2	Shared or Static	93
7.6.3	Debug	93
7.6.4	Shipped Dependencies	94
7.6.5	Tests, Examples and Command Line Tools	94
7.7	Sphinx	94

7.7.1	Build Locally	94
7.7.2	Useful Links	95
8	Maintenance	97
8.1	Release Channels	97
8.1.1	Spack	97
8.1.2	Conda-Forge	97
8.1.3	Brew	97
8.1.4	PyPI	97
8.1.5	ReadTheDocs	98
8.1.6	Doxygen	98

openPMD is an open meta-data schema that provides meaning and self-description for data sets in science and engineering. See [the openPMD standard](#) for details of this schema.

This library provides a reference API for openPMD data handling. Since openPMD is a schema (or markup) on top of portable, hierarchical file formats, this library implements various backends such as HDF5, ADIOS1, ADIOS2 and JSON. Writing & reading through those backends and their associated files is supported for serial and [MPI-parallel](#) workflows.

SUPPORTED OPENPMD STANDARD VERSIONS

openPMD-api is a library using [semantic versioning](#) for its public API. Please see [this link](#) for ABI-compatibility. The version number of openPMD-api is not related to the version of [the openPMD standard](#).

The supported version of the [openPMD standard](#) are reflected as follows: `standardMAJOR.apiMAJOR.apiMINOR`.

openPMD-api version	supported openPMD standard versions
2.0.0+	2.0.0+ (not released yet)
1.0.0+	1.0.1–1.1.0 (not released yet)
0.13.0 (beta)	1.0.0–1.1.0
0.1.0–0.12.0 (alpha)	1.0.0–1.1.0

1.1 Code of Conduct

1.1.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

1.1.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

1.1.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

1.1.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

1.1.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at openpmd@plasma.ninja. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

1.1.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant](https://www.contributor-covenant.org/version/1/4/code-of-conduct.html), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

For answers to common questions about this code of conduct, see <https://www.contributor-covenant.org/faq>

1.2 Citation

openPMD (Open Standard for Particle-Mesh Data Files) is a community project with many people contributing to it. If you use openPMD and/or openPMD related software in your work, please credit it when publishing and/or presenting work performed with it in order to give back to the community.

1.2.1 openPMD-standard

The central definition of **openPMD is the meta data schema** defined in [openPMD/openPMD-standard](#). The most general reference to openPMD is:

Tip: Axel Huebl, Remi Lehe, Jean-Luc Vay, David P. Grote, Ivo F. Sbalzarini, Stephan Kuschel, David Sagan, Christopher Mayes, Frederic Perez, Fabian Koller, and Michael Bussmann. “*openPMD: A meta data standard for particle and mesh based data*,” DOI:10.5281/zenodo.591699 (2015)

Since the openPMD-standard is an actively evolving meta data schema, a specific version of the openPMD standard might be used in your work. You can select a version-specific DOI from the [release page](#) and add the version number to the cited title, e.g.

Note: [author list as above] ... “*openPMD 1.1.0: A meta data standard for particle and mesh based data,*” DOI:10.5281/zenodo.1167843 (2018)

1.2.2 openPMD-api

openPMD-api is a **software library** that provides a reference implementation of the openPMD-standard for popular data formats. It targets both desktop as well as high-performance computing environments.

It is good scientific practice to document all used software, including transient dependencies, with versions in, e.g. a methods section of a publication. As a software citation, you almost always want to refer to a *specific version* of openPMD-api in your work, as illustrated for version 0.10.3:

Tip: Fabian Koller, Franz Poeschel, Junmin Gu, and Axel Huebl. “*openPMD-api 0.10.3: C++ & Python API for Scientific I/O with openPMD,*” DOI:10.14278/rodare.209 (2019)

A list of all releases and DOIs can be found [on the release page](#).

We also provide a DOI that refers to all releases of openPMD-api:

Note: [author list as above] ... “*openPMD-api: C++ & Python API for Scientific I/O with openPMD*” DOI:10.14278/rodare.27 (2018)

Dependent Software

The good way to control complex software environments is to install software through a *package manager* (*see installation*). Furthermore, openPMD-api provides functionality to simplify the documentation of its version and enabled backends:

C++14

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

namespace io = openPMD;

// ...
std::cout << "openPMD-api: "
          << io::getVersion() << std::endl;
std::cout << "openPMD-standard: "
          << io::getStandard() << std::endl;

std::cout << "openPMD-api backend variants: " << std::endl;
for( auto const & v : io::getVariants() )
    std::cout << " " << v.first << ": "
              << v.second << std::endl;
```

Python

```
import openpmd_api as io

print("openPMD-api: {}".format(io.__version__))
print("openPMD-api backend variants: {}".format(io.variants))
```

INSTALLATION

2.1 Installation

Our community loves to help each other. Please [report installation problems](#) in case you should get stuck.

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

2.1.1 Using the Spack Package

A package for openPMD-api is available via the [Spack](#) package manager.

```
# optional:          +python +adios1 -adios2 -hdf5 -mpi
spack install openpmd-api
spack load -r openpmd-api
```

2.1.2 Using the Conda Package

A package for openPMD-api is available via the [Conda](#) package manager.

```
# optional:          OpenMPI support  ==mpi_openmpi*
# optional:          MPICH support    ==mpi_mpich*
conda create -n openpmd -c conda-forge openpmd-api
conda activate openpmd
```

2.1.3 Using the Brew Package

A package for openPMD-api is available via the [Homebrew/Linuxbrew](#) package manager.

```
brew tap openpmd/openpmd
brew install openpmd-api
```

Brew ship only the latest release, includes (Open)MPI support and lacks the ADIOS1 backend.

2.1.4 Using the PyPI Package

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

On very old macOS versions (<10.9) or on exotic processor architectures, this install method *compiles from source* against the found installations of HDF5, ADIOS1, ADIOS2, and/or MPI (in system paths, from other package managers, or loaded via a module system, ...).

```
# we need pip 19 or newer
# optional:                                --user
python3 -m pip install -U pip

# optional:                                --user
python3 -m pip install openpmd-api
```

If MPI-support shall be enabled, we always have to recompile:

```
# optional:                                --user
python3 -m pip install -U pip setuptools wheel
python3 -m pip install -U cmake

# optional:                                --
↪user
openPMD_USE_MPI=ON python3 -m pip install openpmd-api --no-binary openpmd-api
```

2.1.5 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/share/openPMD/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
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\share\openPMD\download_samples.ps1` to download sample files for tests (optional). Replace the last three commands with

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

# optional
ctest -C Release

# 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
# Note that on some systems, /lib might need to be replaced with /lib64.

#               change path to your python MAJOR.MINOR version
export PYTHONPATH=$HOME/somepath/lib/python3.8/site-packages:$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%
```

2.2 Changelog

2.2.1 0.13.0

Date: 2021-01-03

Streaming Support, Python, Benchmarks

This release adds first support for streaming I/O via ADIOS2’s SST engine. More I/O benchmarks have been added with realistic application load patterns. Many Python properties for openPMD attributes have been modernized, with slight breaking changes in Iteration and Mesh data order. This release requires C++14 and adds support for Python 3.9. With this release, we leave the “alpha” phase of the software and declare “beta” status.

Changes to “0.12.0-alpha”

Features

- ADIOS2: streaming support (via ADIOS SST) #570
- add `::availableChunks` call to record component types #802 #835 #847
- HDF5: control alignment via `OPENPMD_HDF5_ALIGNMENT` #830
- JSON configuration on the dataset level #818
- Python
 - attributes as properties in `Series`, `Mesh`, `Iteration`, ... #859
 - add missing python interface (read/write) for machine #796
 - add `Record_Component.make_empty()` #538
- added tests 8a & 8b to do 1D/2D mesh writing and reading #803 #816 #834
- PyPI: support for Windows wheels on x86-64 #853

Bug Fixes

- fix `Series` attributes: read defaults #812
- allow reading a file-based series with many iterations without crashing the number of file handles #822 #837
- Python: Fix & replace `Data_Order` semantics #850
- ADIOS1:
 - add missing `CLOSE_FILE` IO task to parallel backend #785
- ADIOS2:
 - fix engine destruction order, anticipating release 2.7.0 #838
- HDF5:
 - support alternate form of empty records (FBPIC) #849
- Intel ICC (`icpc`):
 - fix export #788
 - fix segfault in `Iteration` #789
- fix & support ClangCL on Windows #832
- CMake:
 - Warnings: ICC & root project only #791
 - Warnings: `FindADIOS(1).cmake` 2.8.12+ #841
 - Warnings: less verbose on Windows #851

Other

- switched to “beta” status: dropping the version `-suffix`
- switch to C++14 #825 #826 #836
- CMake:
 - require version 3.15.0+ #857
 - re-order dependency checks #810
- Python: support 3.6 - 3.9 #828
- Nlohmann-JSON dependency updated to 3.9.1+ #839
- pybind11 dependency updated 2.6.1+ #857
- ADIOS2:
 - less verbose about missing boolean helper attributes #801
 - turn off statistics (Min/Max) #831
- HDF5: better status checks & error messages #795
- Docs:
 - release cibuildwheel example #775
 - `Iteration::close()` is MPI-collective #779
 - overview compression ADIOS2 #781
 - add comment on `lib64/` #793
 - typo in description for ADIOS1 #797
 - conda: recommend fresh environment #799
 - Sphinx/rst: fix warnings #809
 - first read: slice example #819
- CI:
 - Travis -> GH Action #823 #827
 - remove Cygwin #820
 - sanitize only project (temporarily disabled) #800
 - update LGTM environment #844
 - clang-tidy updates #843
 - set oldest supported macOS #854
- Tests:
 - add HiPACE parallel I/O pattern #842 #848
 - cover FBPIC empty HDF5 #849
- Internal: add `Optional` based on `variantSrc::variant` #806

2.2.2 0.12.0-alpha

Date: 2020-09-07

Complex Numbers, Close & Backend Options

This release adds data type support for complex numbers, allows to close iterations and adds first support for backend configuration options (via JSON), which are currently implemented for ADIOS2. Further installation options have been added (homebrew and CLI tool support with pip). New free standing functions and macro defines are provided for version checks.

Changes to “0.11.1-alpha”

Features

- `Record(Component): scalar(), constant(), empty()` #711
- Advanced backend configuration via JSON #569 #733
- Support for complex floating point types #639
- Functionality to close an iteration (and associated files) #746
- Python:
 - `__init__.py` facade #720
 - add `Mesh_Record_Component.position` read-write property #713
 - add `openpmd-ls` tool in pip installs and as module #721 #724
 - more idiomatic unit properties #735
 - add `file_extensions` property #768
- CD:
 - homebrew: add Formula (OSX/Linux) #724 #725
 - PyPI: autodeploy wheels (OSX/Linux) #716 #719
- version compare macro #747
- `getFileExtensions` function #768
- Spack environment file `spack.yaml` added to repo #737
- `openpmd-ls`: add `-v, --version` option #771

Bug Fixes

- `flush()` exceptions in `~Series/~..IOHandler` do not abort anymore #709
- `Iteration/Attributable` assignment operator left object in invalid state #769
- `Datatype.hpp`: add missing include #764
- readme: python example syntax was broken and outdated #722
- examples:
 - fix “weighting” record attribute (ED-PIC) #728
 - fix & validate all created test/example files #738 #739
- warnings:
 - `listSeries`: unused params in try-catch #707
 - fix Doxygen 1.18.8 and 1.18.20 warnings #766

- extended write example: remove MSVC warning #752

Other

- CMake: require version 3.12.0+ #755
- ADIOS2: require version 2.6.0+ #754
- separate header for export macros #704
- rename `AccessType/Access_Type` to `Access` #740 #743 #744
- CI & tests:
 - migration to travis-ci.com / GitHub app #703
 - migrate to GitHub checkout action v2 #712
 - fix OSX numpy install #714
 - move `.travis/` to `.github/ci/` #715
 - move example file download scripts to `share/openPMD/` #715
 - add GCC 9.3 builds #723
 - add Cygwin builds #727
 - add Clang 10.0 builds #759
 - migrate Spack to use AppleClang #758
 - style check scripts: `eval-uable` #757
 - new Spack external package syntax #760
 - python tests: `testAttributes` JSON backend coverage #767
- `listSeries`: remove unused parameters in try-catch #706
- safer internal `*dynamic_cast` of pointers #745
- CMake: subproject inclusion cleanup #751
- Python: remove redundant move in container #753
- read example: show particle load #706
- Record component: fix formatting #763
- add `.editorconfig` file #762
- MPI benchmark: doxygen params #653

2.2.3 0.11.1-alpha

Date: 2020-03-24

HDF5-1.12, Azimuthal Examples & Tagfile

This release adds support for the latest HDF5 release. Also, we add versioned Doxygen and a tagfile for external docs to our online manual.

Changes to “0.11.0-alpha”

Features

- HDF5: Support 1.12 release #696
- Doxygen: per-version index in Sphinx pages #697

Other

- Examples:
 - document azimuthal decomposition read/write #678
 - better example namespace alias (io) #698
- Docs: update API detail pages #699

2.2.4 0.11.0-alpha

Date: 2020-03-05

Robust Independent I/O

This release improves MPI-parallel I/O with HDF5 and ADIOS. ADIOS2 is now the default backend for handling `.bp` files.

Changes to “0.10.3-alpha”

Features

- ADIOS2:
 - new default for `.bp` files (over ADIOS1) #676
 - expose engine #656
- HDF5: `OPENPMD_HDF5_INDEPENDENT=ON` is now default in parallel I/O #677
- defaults for date and software base attributes #657
- `Series::setSoftware()` add second argument for version #657
- free standing functions to query the API version and feature variants at runtime #665
- expose `determineFormat` and `suffix` functions #684
- CLI: add `openpmd-ls` tool #574

Bug Fixes

- `std::ostream& operator<<` overloads are not declared in namespace `std` anymore #662
- ADIOS1:
 - ensure creation of files that only contain attributes #674
 - deprecated in favor of ADIOS2 backend #676
 - allow non-collective `storeChunk()` calls with multiple iterations #679
- Pip: work-around `setuptools`/`CMake` bootstrap issues on some systems #689

Other

- deprecated `Series::setSoftwareVersion:` set the version with the second argument of `setSoftware()` #657
- ADIOS2: require version 2.5.0+ #656
- nvcc:
 - warning missing `erase` overload of `Container` child classes #648
 - warning on unreachable code #659
 - `MPark.Variant`: update C++14 hotfix #618 to upstream version #650
- docs:
 - typo in Python example for first read #649
 - remove all Doxygen warnings and add to CI #654
 - backend feature matrix #661
 - document CMake's `FetchContent` feature for developers #667
 - more notes on HDF5 & ADIOS1 #685
- migrate static checks for python code to GitHub actions #660
- add MPICH tests to CI #670
- `Attribute` constructor: move argument into place #663
- Spack: ADIOS2 backend now enabled by default #664 #676
- add independent HDF5 write test to CI #669
- add test of multiple active `Series` #686

2.2.5 0.10.3-alpha

Date: 2019-12-22

Improved HDF5 Handling

More robust HDF5 file handling and fixes of local includes for more isolated builds.

Changes to “0.10.2-alpha”

Bug Fixes

- Source files: fix includes #640
- HDF5: gracefully handle already open files #643

Other

- Better handling of legacy libSplash HDF5 files #641
- new contributors #644

2.2.6 0.10.2-alpha

Date: 2019-12-17

Improved Error Messages

Thrown errors are now prefixed by the backend in use and ADIOS1 series reads are more robust.

Changes to “0.10.1-alpha”

Bug Fixes

- Implement assignment operators for: `IOTask`, `Mesh`, `Iteration`, `BaseRecord`, `Record` #628
- Missing `virtual` destructors added #632

Other

- Backends: Prefix Error Messages #634
- ADIOS1: Skip Invalid Scalar Particle Records #635

2.2.7 0.10.1-alpha

Date: 2019-12-06

ADIOS2 Open Speed and NVCC Fixes

This releases improves the initial time spend when parsing data series with the ADIOS2 backend. Compile problems when using the CUDA NVCC compiler in downstream projects have been fixed. We adopted a Code of Conduct in openPMD.

Changes to “0.10.0-alpha”

Features

- C++: add `Container::contains` method #622

Bug Fixes

- ADIOS2:
 - fix C++17 build #614
 - improve initial open speed of series #613
- nvcc:
 - ignore export of `enum class Operation` #617
 - fix C++14 build #618

Other

- community:
 - code of conduct added #619
 - all contributors listed in README #621
- manylinux2010 build automation updated for Python 3.8 #615

2.2.8 0.10.0-alpha

Date: 2019-11-14

ADIOS2 Preview, Python & MPI Improved

This release adds a first (preview) implementation of ADIOS2 (BP4). Python 3.8 support as well as improved pip builds on macOS and Windows have been added. ADIOS1 and HDF5 now support non-collective (independent) store and load operations with MPI. More HPC compilers, such as IBM XL, ICC and PGI have been tested. The manual has been improved with more details on APIs, examples, installation and backends.

Changes to “0.9.0-alpha”

Features

- ADIOS2: support added (v2.4.0+) #482 #513 #530 #568 #572 #573 #588 #605
- HDF5: add `OPENPMD_HDF5_INDEPENDENT` for non-collective parallel I/O #576
- Python:
 - Python 3.8 support #581
 - support empty datasets via `Record_Component.make_empty` #538
- pkg-config: add `static` variable (`true/false`) to `openPMD.pc` package #580

Bug Fixes

- Clang: fix pybind11 compile on older releases, such as AppleClang 7.3-9.0, Clang 3.9 #543
- Python:
 - OSX: fix `dlopen` issues due to missing `@loader_path` with `pip/setup.py` #595
 - Windows: fix a missing DLL issue by building static with `pip/setup.py` #602
 - import `mpi4py` first (MPICH on OSX issue) #596
 - skip examples using HDF5 if backend is missing #544
 - fix a variable shadowing in `Mesh` #582
 - add missing `.unit_dimension` for records #611
- ADIOS1: fix deadlock in MPI-parallel, non-collective calls to `storeChunk()` #554
- xLC 16.1: work-around C-array initializer parsing issue #547
- icc 19.0.0 and PGI 19.5: fix compiler ID identification #548
- CMake: fix false-positives in `FindADIOS.cmake` module #609
- Series: throws an error message if no file ending is specified #610

Other

- Python: improve `pip` install instructions #594 #600
- PGI 19.5: fix warning `static constexpr: storage class first` #546
- JSON:
 - the backend is now always enabled #564 #587
 - NLOhmann-JSON dependency updated to 3.7.0+ #556
- gitignore: generalize CLion, more build dirs #549 #552
- fix clang-tidy warnings: `strcmp` and modernize `auto`, `const` correctness #551 #560
- `ParallelIOTest`: less code duplication #553
- Sphinx manual:
 - PDF Chapters #557
 - draft for the API architecture design #186
 - draft for MPI data and collective contract in API usage #583
 - fix tables & missing examples #579
 - “first write” explains `unitDimension` #592
 - link to datasets used in examples #598
 - fix minor formatting and include problems #608
- README:
 - add authors and acknowledgements #566
 - correct a typo #584
 - use `$(which python3)` for CMake Python option #599
 - update ADIOS homepage & CMake #604
- Travis CI:
 - speedup dependency build #558
 - `-Werror` only in build phase #565

2.2.9 0.9.0-alpha

Date: 2019-07-25

Improved Builds and Packages

This release improves PyPI releases with proper declaration of build dependencies (use `pip 19.0+`). For Makefile-based projects, an `openPMD.pc` file to be used with `pkg-config` is added on install. `RecordComponent` now supports a `makeEmpty` method to write a zero-extent, yet multi-dimensional record component. We are now building as shared library by default.

Changes to “0.8.0-alpha”

Features

- C++: support empty datasets via `RecordComponent::makeEmpty` #528 #529
- CMake:
 - build a shared library by default #506
 - generate `pkg-config .pc` file #532 #535 #537
- Python:
 - manylinux2010 wheels for PyPI #523
 - add `pyproject.toml` for build dependencies (PEP-518) #527

Bug Fixes

- `MPark.Variant`: work-around missing version bump #504
- linker error concerning `Mesh::setTimeOffset` method template #511
- remove dummy dataset writing from `RecordComponent::flush()` #528
- remove dummy dataset writing from `PatchRecordComponent::flush` #512
- allow flushing before defining `position` and `positionOffset` components of particle species #518 #519
- CMake:
 - make install paths cacheable on Windows #521
 - HDF5 linkage is private #533
- warnings:
 - unused variable in JSON backend #507
 - MSVC: Warning DLL Interface `STDlib` #508

Other

- increase `pybind11` dependency to 2.3.0+ #525
- GitHub:
 - auto-add labels #515
 - issue template for install issues #526
 - update badges #522
- docs:
 - link parallel python examples in manual #499
 - improved Doxygen parsing for all backends #500
 - fix typos #517

2.2.10 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 `Access` #483
 - improved MPI-parallel write example #496

2.2.11 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

2.2.12 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

2.2.13 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

2.2.14 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

2.2.15 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

2.2.16 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

2.2.17 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

2.2.18 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

2.2.19 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\dt 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

2.2.20 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

2.2.21 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

2.2.22 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

2.2.23 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.

2.3 Upgrade Guide

2.3.1 0.13.0

Building openPMD-api now requires a compiler that supports C++14 or newer. Supported Python version are now 3.6 to 3.9. CMake 3.15.0 is now the minimally supported version for CMake.

Python

Reading the `data_order` of a mesh was broken. The old setter function (`set_data_order`) and read-only property (`data_order`) are now unified in a single, writable property:

```
import openpmd_api as io

series = io.Series("data%T.h5", io.Access.read_only)
rho = series.iterations[0].meshes["rho"]
rho.data_order = 'C' # or 'F'

print(rho.data_order == 'C') # True
```

Note: we recommend using 'C' order since version 2 of the openPMD-standard will simplify this option to 'C', too. For Fortran-ordered indices, please just invert the attributes `axis_labels`, `grid_spacing` and `grid_global_offset` accordingly.

The Iteration functions `time`, `dt` and `time_unit_SI` have been replaced with read-write properties of the same name, essentially without the `()`-access. `set_time`, `set_dt` and `set_time_unit_SI` are now deprecated and will be removed in future versions of the library.

The already existing read-only Series properties `openPMD`, `openPMD_extension`, `base_path`, `meshes_path`, `particles_path`, `particles_path`, `author`, `date`, `iteration_encoding`, `iteration_format` and `name` are now declared as read-write properties. `set_openPMD`, `set_openPMD_extension`, `set_base_path`, `set_meshes_path`, `set_particles_path`, `set_author`, `set_date`, `set_iteration_encoding`, `set_iteration_format` and `set_name` are now deprecated and will be removed in future versions of the library.

The already existing read-only Mesh properties `geometry`, `geometry_parameters`, `axis_labels`, `grid_spacing`, `grid_global_offset` and `grid_unit_SI` are now declared as read-write properties. `set_geometry`, `set_geometry_parameters`, `set_axis_labels`, `set_grid_spacing`, `set_grid_global_offset` and `set_grid_unit_SI` are now deprecated and will be removed in future versions of the library.

The already existing read-only Attributable property `comment` is now declared as read-write properties. `set_comment` is now deprecated and will be removed in future versions of the library.

2.3.2 0.12.0-alpha

CMake 3.12.0 is now the minimally supported version for CMake. ADIOS 2.6.0 is now the minimally supported version for ADIOS2 support.

Python

The already existing read-only properties `unit_dimension`, `unit_SI`, and `time_offset` are now declared as read-write properties. `set_unit_dimension`, `set_unit_SI`, and `set_time_offset` are now deprecated and will be removed in future versions of the library.

`Access_Type` is now called `Access`. Using it by the old name is deprecated and will be removed in future versions of the library.

C++

`AccessType` is now called `Access`. Using it by the old name is deprecated and will be removed in future versions of the library.

2.3.3 0.11.0-alpha

ADIOS2 is now the default backend for `.bp` files. As soon as the ADIOS2 backend is enabled it will take precedence over a potentially also enabled ADIOS1 backend. In order to prefer the legacy ADIOS1 backend in such a situation, set an environment variable: `export OPENPMD_BP_BACKEND="ADIOS1"`. Support for ADIOS1 is now deprecated.

Independent MPI-I/O is now the default in parallel HDF5. For the old default, collective parallel I/O, set the environment variable `export OPENPMD_HDF5_INDEPENDENT="OFF"`. Collective parallel I/O makes more functionality, such as `storeChunk` and `loadChunk`, MPI-collective. HDF5 attribute writes are MPI-collective in either case, due to HDF5 restrictions.

Our `Spack` packages build the ADIOS2 backend now by default. Pass `-adios2` to the Spack spec to disable it: `spack install openpmd-api -adios2` (same for `spack load -r`).

The `Series::setSoftwareVersion` method is now deprecated and will be removed in future versions of the library. Use `Series::setSoftware(name, version)` instead. Similarly for the Python API, use `Series.set_software` instead of `Series.set_software_version`.

The automated example-download scripts have been moved from `.travis/download_samples.sh` (and `.ps1`) to `share/openPMD/`.

2.3.4 0.10.0-alpha

We added preliminary support for ADIOS2 in this release. As long as also the ADIOS1 backend is enabled it will take precedence for `.bp` files over the newer ADIOS2 backend. In order to enforce using the new ADIOS2 backend in such a situation, set an environment variable: `export OPENPMD_BP_BACKEND="ADIOS2"`. We will change this default in upcoming releases to prefer ADIOS2.

The JSON backend is now always enabled. The CMake option `-DopenPMD_USE_JSON` has been removed (as it is always ON now).

Previously, omitting a file ending in the `Series` constructor chose a “dummy” no-operation file backend. This was confusing and instead a runtime error is now thrown.

2.3.5 0.9.0-alpha

We are now building a shared library by default. In order to keep build the old default, a static library, append `-DBUILD_SHARED_LIBS=OFF` to the `cmake` command.

2.3.6 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();
```


3.1 First Write

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

3.1.1 Include / Import

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

C++14

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

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

namespace io = openPMD;
```

Python

```
import openpmd_api as io

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

3.1.2 Open

Write into a new openPMD series in `myOutput/data_<00...N>.h5`. Further file formats than `.h5` ([HDF5](#)) are supported: `.bp` ([ADIOS1/ADIOS2](#)) or `.json` ([JSON](#)).

C++14

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

Python

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

3.1.3 Iteration

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

C++14

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

Python

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

3.1.4 Attributes

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

C++14

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

Python

```
series.author = \  
    "Axel Huebl <axelhuebl@lbl.gov>"  
series.machine = "Hall Probe 5000, Model 3"  
series.set_attribute(  
    "dinner", "Pizza and Coke")  
i.set_attribute(  
    "vacuum", True)
```

3.1.5 Data

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

C++14

```
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.
```

3.1.6 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++14

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

// record components
auto B_x = B["x"];
auto B_y = B["y"];
auto B_z = B["z"];

auto dataset = io::Dataset(
    io::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 = io.Dataset(
    x_data.dtype,
    x_data.shape)
B_x.reset_dataset(dataset)
B_y.reset_dataset(dataset)
B_z.reset_dataset(dataset)
```

3.1.7 Units

Let's describe this magnetic field \vec{B} in more detail. Independent of the absolute unit system, a magnetic field has the **physical dimension** of $[\text{mass (M)}^1 \cdot \text{electric current (I)}^{-1} \cdot \text{time (T)}^{-2}]$.

Ouch, our magnetic field was measured in **cgs units**! Quick, let's also store the conversion factor 10^{-4} from **Gauss** (cgs) to **Tesla** (SI).

C++14

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

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

Python

```
# unit system agnostic dimension
B.unit_dimension = {
    io.Unit_Dimension.M: 1,
    io.Unit_Dimension.I: -1,
    io.Unit_Dimension.T: -2
}

# conversion to SI
B_x.unit_SI = 1.e-4
B_y.unit_SI = 1.e-4
B_z.unit_SI = 1.e-4
```

Tip: Annotating the *physical dimension* (`unitDimension`) of a record allows us to read data sets with *arbitrary names* and understand their purpose simply by **dimensional analysis**. The dimensional **base quantities** in openPMD are length (L), mass (M), time (T), electric current (I), thermodynamic temperature (`theta`), amount of substance

(N), luminous intensity (J) after the international system of quantities (ISQ). The *factor to SI* (`unitSI`) on the other hand allows us to convert values between absolute unit systems.

3.1.8 Register Chunk

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

C++14

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

3.1.9 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++14

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

Python

```
series.flush()
```

3.1.10 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++14

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

Python

```
del series
```

3.2 First Read

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

3.2.1 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 io = openPMD;
```

Python

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

3.2.2 Open

Open an existing openPMD series in `data<N>.h5`. Further file formats than `.h5` ([HDF5](#)) are supported: `.bp` ([ADIOS1/ADIOS2](#)) or `.json` ([JSON](#)).

C++11

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

Python

```
series = io.Series(
    "data%T.h5",
    io.Access.read_only)
```

3.2.3 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]
```

3.2.4 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)
```

3.2.5 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"]
```

Tip: You can check via `i.meshes.contains("E")` (C++) or `"E" in i.meshes` (Python) if an entry exists.

3.2.6 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();

// ...
// io::UnitDimension::M

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


Python

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

# ...
# io.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.

3.2.7 Register Chunk

We can load record components partially and in parallel or at once. Reading small data one by one 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
# invalid 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` !**

One can also request to load a slice of data:

C++11

```
Extent extent = E_x.getExtent();
extent.at(2) = 1;
std::shared_ptr< double > x_slice_data =
    E_x.loadChunk< double >(
        io::Offset{0, 0, 4}, extent);
```

Python

```
# we support slice syntax, too
x_slice_data = E_x[:, :, 4]
```

Don't forget that we still need to `flush()`.

3.2.8 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 variables `x_data` and `x_slice_data` can be read, manipulated and/or deleted.

C++11

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

Python

```
series.flush()
```

3.2.9 Data

We can now work with the newly loaded data in `x_data` (or `x_slice_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";

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])
```

3.2.10 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
```

3.3 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.

3.3.1 Reading

C++

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

#include <iostream>
#include <memory>
#include <cstdint>

using std::cout;
using namespace openPMD;

int main()
{
    Series series = Series(
        "../samples/git-sample/data%T.h5",
        Access::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:"
    << '\n';
```

(continues on next page)

(continued from previous page)

```

    for( auto const& ps : i.particles ) {
        cout << "\n\t" << ps.first;
        for( auto const& r : ps.second ) {
            cout << "\n\t" << r.first;
            cout << '\n';
        }
    }

    openPMD::ParticleSpecies electrons = i.particles["electrons"];
    std::shared_ptr<double> charge = electrons["charge
↪"] [openPMD::RecordComponent::SCALAR].loadChunk<double>();
    series.flush();
    cout << "And the first electron particle has a charge = " << charge.get()[0];
    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";

    /* 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 as io

if __name__ == "__main__":
    series = io.Series("../samples/git-sample/data%T.h5",
                      io.Access.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("With records:")
        for r in i.particles[ps]:
            print("\t {0}".format(r))

    # printing a scalar value
    electrons = i.particles["electrons"]
    charge = electrons["charge"][io.Mesh_Record_Component.SCALAR]
    series.flush()
    print("And the first electron particle has a charge {}".format(
        charge[0]))
    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))
    print(all_data[0, 0, :5])

```

(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.
# One can delete the object explicitly (or let it run out of scope) to
# trigger this.
del series

```

3.3.2 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",
        Access::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();
    cout << "File structure and required attributes have been written\n";
}

```

(continues on next page)

(continued from previous page)

```

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 as io
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 = io.Series(
        "../samples/3_write_serial_py.h5",
        io.Access.create
    )

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

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

    dataset = io.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

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

```

(continues on next page)

(continued from previous page)

```
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
```

3.4 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.

3.4.1 Reading

C++

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

#include <mpi.h>

#include <iostream>
#include <memory>
#include <cstdint>

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",
            Access::READ_ONLY,
            MPI_COMM_WORLD
        );
        if( 0 == mpi_rank )
            cout << "Read a series in parallel with " << mpi_size << " MPI ranks\n"
                << "\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;
}

```

Python

```

# IMPORTANT: include mpi4py FIRST
# https://mpi4py.readthedocs.io/en/stable/mpi4py.run.html
# on import: calls MPI_Init_thread()
# exit hook: calls MPI_Finalize()
from mpi4py import MPI

import openpmd_api as io

if __name__ == "__main__":
    # also works with any other MPI communicator

```

(continues on next page)

(continued from previous page)

```

comm = MPI.COMM_WORLD

series = io.Series(
    "../samples/git-sample/data%T.h5",
    io.Access.read_only,
    comm
)
if 0 == comm.rank:
    print("Read a series in parallel with {} MPI ranks".format(
        comm.size))

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

chunk_offset = [comm.rank + 1, 1, 1]
chunk_extent = [2, 2, 1]

chunk_data = E_x.load_chunk(chunk_offset, chunk_extent)

if 0 == comm.rank:
    print("Queued the loading of a single chunk per MPI rank from disk, "
        "ready to execute")
series.flush()

if 0 == comm.rank:
    print("Chunks have been read from disk")

for i in range(comm.size):
    if i == comm.rank:
        print("Rank {} - Read chunk contains:".format(i))
        for row in range(chunk_extent[0]):
            for col in range(chunk_extent[1]):
                print("\t({}|{}|1)\t{:e}".format(
                    row + chunk_offset[0],
                    col + chunk_offset[1],
                    chunk_data[row, col, 0]
                ), end='')
            print("")

        # this barrier is not necessary but structures the example output
        comm.Barrier()

# 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.
# In any case, this must happen before MPI_Finalize() is called
# (usually in the mpi4py exit hook).
del series

```

3.4.2 Writing

C++

```

#include <openPMD/openPMD.hpp>

#include <mpi.h>

#include <iostream>
#include <memory>

```

(continues on next page)

(continued from previous page)

```

#include <vector>    // std::vector

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();
     */
    {
        // global data set to write: [MPI_Size * 10, 300]
        // each rank writes a 10x300 slice with its MPI rank as values
        auto 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",
            Access::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
    }
}

```

(continues on next page)

(continued from previous page)

```

    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, "
                "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;
}

```

Python

```

# IMPORTANT: include mpi4py FIRST
# https://mpi4py.readthedocs.io/en/stable/mpi4py.run.html
# on import: calls MPI_Init_thread()
# exit hook: calls MPI_Finalize()
from mpi4py import MPI

import openpmd_api as io
import numpy as np

if __name__ == "__main__":
    # also works with any other MPI communicator
    comm = MPI.COMM_WORLD

    # global data set to write: [MPI_Size * 10, 300]
    # each rank writes a 10x300 slice with its MPI rank as values
    local_value = comm.size
    local_data = np.ones(10 * 300,
                          dtype=np.double).reshape(10, 300) * local_value

    if 0 == comm.rank:
        print("Set up a 2D array with 10x300 elements per MPI rank ({}x) "
              "that will be written to disk".format(comm.size))

    # open file for writing
    series = io.Series(
        "../samples/5_parallel_write_py.h5",
        io.Access.create,
        comm
    )
    if 0 == comm.rank:
        print("Created an empty series in parallel with {} MPI ranks".format(
            comm.size))

    mymesh = series.iterations[1]. \
        meshes["mymesh"][io.Mesh_Record_Component.SCALAR]

    # example 1D domain decomposition in first index
    global_extent = [comm.size * 10, 300]

```

(continues on next page)

(continued from previous page)

```

dataset = io.Dataset(local_data.dtype, global_extent)

if 0 == comm.rank:
    print("Prepared a Dataset of size {} and Datatype {}".format(
        dataset.extent, dataset.dtype))

mymesh.reset_dataset(dataset)
if 0 == comm.rank:
    print("Set the global Dataset properties for the scalar field "
          "mymesh in iteration 1")

# example shows a 1D domain decomposition in first index
mymesh[comm.rank*10:(comm.rank+1)*10, :] = local_data
if 0 == comm.rank:
    print("Registered a single chunk per MPI rank containing its "
          "contribution, ready to write content to disk")

series.flush()
if 0 == comm.rank:
    print("Dataset content has been fully written to disk")

# 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

```

3.5 Streaming

Note: Data streaming is a novel backend and under active development. At the moment, the internal data format is still changing rapidly and is likely not compatible between releases of the openPMD-api.

The openPMD API includes a streaming-aware API as well as streaming-enabled backends (currently: ADIOS2).

Unlike in file-based backends, the order in which data is put out becomes relevant in streaming-based backends. Each iteration will be published as one atomical step by the streaming API (compare the concept of [steps in ADIOS2](#)).

3.5.1 Reading

The reading end of the streaming API enforces further restrictions that become necessary through the nature of streaming. It can be used to read any kind of openPMD-compatible dataset, stream-based and filesystem-based alike.

C++

The reading end of the streaming API is activated through use of `Series::readIterations()` instead of accessing the field `Series::iterations` directly. The returned object of type `ReadIterations` can be used in a C++11 range-based for loop to iterate over objects of type `IndexedIteration`. This class extends the `Iteration` class with a field `IndexedIteration::iterationIndex`, denoting this iteration's index.

Users are encouraged to explicitly `.close()` the iteration after reading from it. Closing the iteration will flush all pending operations on that iteration. If an iteration is not closed until the beginning of the next iteration, it will be closed automatically.

Note that a closed iteration cannot be reopened. This pays tribute to the fact that in streaming mode, an iteration may be dropped by the data source once the data sink has finished reading from it.

```
#define BUILD_STREAMING_EXAMPLE false
#if BUILD_STREAMING_EXAMPLE
#include <openPMD/openPMD.hpp>

#include <array>
#include <iostream>
#include <memory>

using std::cout;
using namespace openPMD;

int
main()
{
    #if openPMD_HAVE_ADIOS2
    using position_t = double;
    Series series = Series( "electrons.sst", Access::READ_ONLY );

    for( IndexedIteration iteration : series.readIterations() )
    {
        std::cout << "Current iteration: " << iteration.iterationIndex
                    << std::endl;
        Record electronPositions = iteration.particles[ "e" ][ "position" ];
        std::array< std::shared_ptr< position_t >, 3 > loadedChunks;
        std::array< Extent, 3 > extents;
        std::array< std::string, 3 > const dimensions{ { "x", "y", "z" } };

        for( size_t i = 0; i < 3; ++i )
        {
            std::string dim = dimensions[ i ];
            RecordComponent rc = electronPositions[ dim ];
            loadedChunks[ i ] = rc.loadChunk< position_t >(
                Offset( rc.getDimensionality(), 0 ), rc.getExtent() );
            extents[ i ] = rc.getExtent();
        }

        iteration.close();

        for( size_t i = 0; i < 3; ++i )
        {
            std::string dim = dimensions[ i ];
            Extent const & extent = extents[ i ];
            std::cout << "\ndim: " << dim << "\n" << std::endl;
            auto chunk = loadedChunks[ i ];
            for( size_t j = 0; j < extent[ 0 ]; ++j )
            {
                std::cout << chunk.get()[ j ] << ", ";
            }
            std::cout << "\n-----\n" << std::endl;
        }
    }
    #endif
}
```

(continues on next page)

(continued from previous page)

```

    }
}

return 0;
#else
    std::cout << "The streaming example requires that openPMD has been built "
                "with ADIOS2."
                << std::endl;
    return 0;
#endif
}
#else
int main(){ return 0; }
#endif

```

Python

The reading end of the streaming API is activated through use of `Series.read_iterations()` instead of accessing the field `Series.iterations` directly. The returned object of type `ReadIterations` can be used in a Python range-based for loop to iterate over objects of type `IndexedIteration`. This class extends the `Iteration` class with a field `IndexedIteration.iteration_index`, denoting this iteration's index.

Users are encouraged to explicitly `.close()` the iteration after reading from it. Closing the iteration will flush all pending operations on that iteration. If an iteration is not closed until the beginning of the next iteration, it will be closed automatically.

Note that a closed iteration cannot be reopened. This pays tribute to the fact that in streaming mode, an iteration may be dropped by the data source once the data sink has finished reading from it.

```

#!/usr/bin/env python
import openpmd_api as io

run_streaming_example = False

if not run_streaming_example:
    exit(0)

if __name__ == "__main__":
    if 'adios2' not in io.variants or not io.variants['adios2']:
        print('This example requires ADIOS2')
        exit(0)

    series = io.Series("stream.sst", io.Access_Type.read_only)

    for iteration in series.read_iterations():
        print("Current iteration {}".format(iteration.iteration_index))
        electronPositions = iteration.particles["e"]["position"]
        loadedChunks = []
        shapes = []
        dimensions = ["x", "y", "z"]

        for i in range(3):
            dim = dimensions[i]
            rc = electronPositions[dim]
            loadedChunks.append(rc.load_chunk([0], rc.shape))
            shapes.append(rc.shape)
        iteration.close()

        for i in range(3):
            dim = dimensions[i]

```

(continues on next page)

(continued from previous page)

```
shape = shapes[i]
print("dim: {}".format(dim))
chunk = loadedChunks[i]
print(chunk)
```

3.5.2 Writing

The writing end of the streaming API enforces further restrictions that become necessary through the nature of streaming. It can be used to write any kind of openPMD-compatible dataset, stream-based and filesystem-based alike.

C++

The writing end of the streaming API is activated through use of `Series::writeIterations()` instead of accessing the field `Series::iterations` directly. The returned object of type `WriteIterations` wraps the field `Series::iterations`, but exposes only a restricted subset of functionality. Using `WriteIterations::operator[](uint64_t)` will automatically open a streaming step for the corresponding iteration.

Users are encouraged to explicitly `.close()` the iteration after writing to it. Closing the iteration will flush all pending operations on that iteration. If an iteration is not closed until the next iteration is accessed via `WriteIterations::operator[](uint64_t)`, it will be closed automatically.

Note that a closed iteration cannot be reopened. This pays tribute to the fact that in streaming mode, an iteration is sent to the sink upon closing it and the data source can no longer modify it.

```
#define BUILD_STREAMING_EXAMPLE false
#if BUILD_STREAMING_EXAMPLE
#include <openPMD/openPMD.hpp>

#include <iostream>
#include <memory>
#include <numeric> // std::iota

using std::cout;
using namespace openPMD;

int
main()
{
    #if openPMD_HAVE_ADIOS2
        using position_t = double;

        // open file for writing
        Series series = Series( "electrons.sst", Access::CREATE );

        Datatype datatype = determineDatatype< position_t >();
        constexpr unsigned long length = 10ul;
        Extent global_extent = { length };
        Dataset dataset = Dataset( datatype, global_extent );
        std::shared_ptr< position_t > local_data(
            new position_t[ length ],
            []( position_t const * ptr ) { delete[] ptr; } );

        WriteIterations iterations = series.writeIterations();
        for( size_t i = 0; i < 100; ++i )
        {
            Iteration iteration = iterations[ i ];
```

(continues on next page)

(continued from previous page)

```

Record electronPositions = iteration.particles[ "e" ][ "position" ];

std::iota( local_data.get(), local_data.get() + length, i * length );
for( auto const & dim : { "x", "y", "z" } )
{
    RecordComponent pos = electronPositions[ dim ];
    pos.resetDataset( dataset );
    pos.storeChunk( local_data, Offset{ 0 }, global_extent );
}
iteration.close();
}

return 0;
#else
std::cout << "The streaming example requires that openPMD has been built "
            "with ADIOS2."
            << std::endl;
return 0;
#endif
}
#else
int main(){ return 0; }
#endif

```

Python

The writing end of the streaming API is activated through use of `Series.write_iterations()` instead of accessing the field `Series.iterations` directly. The returned object of type `WriteIterations` wraps the field `Series.iterations`, but exposes only a restricted subset of functionality. Using `WriteIterations.__getitem__(index)` (i.e. the index operator `series.writeIterations()[index]`) will automatically open a streaming step for the corresponding iteration.

Users are encouraged to explicitly `.close()` the iteration after writing to it. Closing the iteration will flush all pending operations on that iteration. If an iteration is not closed until the next iteration is accessed via `WriteIterations.__getitem__(index)`, it will be closed automatically.

Note that a closed iteration cannot be reopened. This pays tribute to the fact that in streaming mode, an iteration is sent to the sink upon closing it and the data source can no longer modify it.

```

#!/usr/bin/env python
import openpmd_api as io
import numpy as np

run_streaming_example = False

if not run_streaming_example:
    exit(0)

if __name__ == "__main__":
    if 'adios2' not in io.variants or not io.variants['adios2']:
        print('This example requires ADIOS2')
        exit(0)

    series = io.Series("stream.sst", io.Access_Type.create)
    datatype = np.dtype("double")
    length = 10
    global_extent = [10]
    dataset = io.Dataset(datatype, global_extent)

    iterations = series.write_iterations()

```

(continues on next page)

(continued from previous page)

```
for i in range(100):
    iteration = iterations[i]
    electronPositions = iteration.particles["e"]["position"]

    local_data = np.arange(i * length, (i + 1) * length, dtype=datatype)
    for dim in ["x", "y", "z"]:
        pos = electronPositions[dim]
        pos.reset_dataset(dataset)
        pos[()] = local_data
    iteration.close()
```

3.6 Benchmarks

3.6.1 Parallel benchmarks 8a & 8b

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.

Writing

Source: `examples/8a_benchmark_write_parallel.cpp`

This benchmark writes a few meshes and particles, either 1D, 2D or 3D.

The meshes are viewed as grid of mini blocks. As an example, we assume the mini blocks dimension are [16, 32, 32].

Next we define the grid based on the mini block. say, [32, 32, 16]. Then our actual mesh size is [16x32, 32x32, 32x16].

Here is a sample input file ("w.input"):

```
dim=3
balanced=true
ratio=1
steps=10
minBlock=16 32 32
grid=32 32 16
```

With the above input file, we will create an openPMD file with the above mesh using

- 3D data
- balanced load
- particle to mesh ratio = 1
- 10 iteration steps

Note: All files generated are group based. i.e. One file per iteration.

To run:

```
./8a_benchmark_write_parallel w.input
```

then the file generated are: `../samples/8a_parallel_3Db_*`

Reading

Source: `examples/8b_benchmark_read_parallel.cpp`

This benchmark is to read from the files written by 8a.

The options are: a file prefix, and a read pattern

For example, if the files are in the format of `/path/8a_parallel_3Db_%07T.bp` the input will be: `/path/8a_parallel_3Db <options>`

The Read options intent to measure overall processing time in the following categories:

- Metadata only (option = m)
- or data retrieval (after metadata loaded)

The data retrieval is further divided into:

- **slice the “rho” mesh** (options = `sx/sy/sz` depends on which direction. e.g. `sx` implies `x=0`.)
- **slice on the 3D magnetic field(e.g. find values for “Bx”, “By” and “Bz”)** (options = `fx/fy/fz` depends on which direction. e.g. `fx` implies `x=0`.)

So here are the options one can use to read a file:

- m
- sx
- sy
- sz
- fx
- fy
- fz

For example, To read files generated by the above write command, metadata only:

```
./8b_benchmark_read_parallel ../samples/8a_parallel_3Db m
```

More complicated Writing options (Applies to ADIOS BP)

The ADIOS BP files uses subfiles to store data from each rank. We have an option to provide hint on how data should be divided per rank in the command line: the order of options are:

- grid of minimal blocks|balance|particle2mesh ratio
- minial blocks
- use multiple blocks
- num of timesteps,
- dimensions
- hint on work load arrangement.

Example: `“mpirun -n 4 ./8a_benchmark_write_parallel 400801 16016 1 5 3 4004002 “`

Here 4 ranks are used to write a 3D mesh, minimal block is [16,16,16], grid of minimal block is [8,4,4], so the actual mesh = [16x8, 16x4, 16x4]. Number of timestep = 5.

The hint is asking each rank to work on a [16x2, 16x4, 16x4] block. It precisely cover the mesh with 4 ranks, so will be applied.

3.6.2 Benchmark Utilities

Further benchmarks are found in *utilities*.

3.7 All Examples

The full list of example scripts shown below is also contained in our `examples/` folder.

Example data sets can be downloaded from: github.com/openPMD/openPMD-example-datasets. The following command will automatically install those into `samples/` on Linux and OSX: `curl -sSL https://git.io/JewVw | bash`

3.7.1 C++

- `1_structure.cpp`: creating a first series
- `2_read_serial.cpp`: reading a mesh
- `2a_read_thetaMode_serial.cpp`: read an azimuthally decomposed mesh (and reconstruct it)
- `3_write_serial.cpp`: writing a mesh
- `3a_write_thetaMode_serial.cpp`: write an azimuthally decomposed 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

Benchmarks

- `8_benchmark_parallel.cpp`: a MPI-parallel IO-benchmark
- `8a_benchmark_write_parallel.cpp`: creates 1D/2D/3D arrays, with each rank having a few blocks to write to
- `8b_benchmark_read_parallel.cpp`: read slices of meshes and particles

3.7.2 Python

- `2_read_serial.py`: reading a mesh
- `2a_read_thetaMode_serial.py`: reading an azimuthally decomposed mesh (and reconstruct it)
- `3_write_serial.py`: writing a mesh
- `3a_write_thetaMode_serial.py`: write an azimuthally decomposed mesh
- `4_read_parallel.py`: MPI-parallel mesh read
- `5_write_parallel.py`: MPI-parallel mesh write
- `7_extended_write_serial.py`: particle writing with patches and constant records
- `9_particle_write_serial.py`: writing particles

3.7.3 Unit Tests

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

API DETAILS

4.1 C++

Our Doxygen page [provides an index of all C++ functionality](#).

4.1.1 Public Headers

`#include` ... the following headers to use openPMD-api:

Include	Description
<code><openPMD/openPMD.hpp></code>	Public facade header (serial and MPI)
<code><openPMD/benchmark/...></code>	Optional <i>benchmark</i> helpers

4.1.2 External Documentation

If you want to link to the openPMD-api doxygen index [from an external documentation](#), you can find the [Doxygen tag file here](#).

If you want to use this tag file with e.g. `xeus-cling`, add the following in its configuration directory:

```
{
  "url": "https://openpmd-api.readthedocs.io/en/<adjust-version-of-tag-file-here>
↔/_static/doxyhtml/",
  "tagfile": "openpmd-api-doxygen-web.tag.xml"
}
```

4.2 Python

4.2.1 Public Headers

`import` ... the following python module to use openPMD-api:

Import	Description
<code>openpmd_api</code>	Public facade import (serial and MPI)

Note: As demonstrated in our [python examples](#), MPI-parallel scripts must import from `mpi4py` `import MPI` prior to importing `openpmd_api` in order to [initialize MPI first](#).

Otherwise, errors of the following kind will occur:

The `MPI_Comm_test_inter()` function was called before `MPI_INIT` was invoked. This **is** disallowed by the MPI standard. Your MPI job will now abort.

4.3 MPI

4.3.1 Collective Behavior

openPMD-api is designed to support both serial as well as parallel I/O. The latter is implemented through the [Message Passing Interface \(MPI\)](#).

A **collective** operation needs to be executed by *all* MPI ranks of the MPI communicator that was passed to `openPMD::Series`. Contrarily, **independent** operations can also be called by a subset of these MPI ranks. For more information, please see the [MPI standard documents](#), for example MPI-3.1 in “[Section 2.4 - Semantic Terms](#)”.

Functionality	Behavior	Description
<code>Series</code>	collective	open and close
<code>::flush()</code>	collective	read and write
<code>Iteration</code> ¹	independent	declare and open
<code>::open()</code> ³	collective	explicit open
<code>Mesh</code> ¹	independent	declare, open, write
<code>ParticleSpecies</code> ¹	independent	declare, open, write
<code>::setAttribute</code> ²	<i>backend-specific</i>	declare, write
<code>::getAttribute</code>	independent	open, reading
<code>::storeChunk</code> ¹	independent	write
<code>::loadChunk</code>	independent	read

Tip: Just because an operation is independent does not mean it is allowed to be inconsistent. For example, undefined behavior will occur if ranks pass differing values to `::setAttribute` or try to use differing names to describe the same mesh.

4.3.2 Efficient Parallel I/O Patterns

Note: This section is a stub. We will improve it in future versions.

Write as large data set chunks as possible in `::storeChunk` operations.

Read in large, non-overlapping subsets of the stored data (`::loadChunk`). Ideally, read the same chunk extents as were written, e.g. through `ParticlePatches` (example to-do).

See the *implemented I/O backends* for individual tuning options.

¹ Individual backends, e.g. *HDF5*, will only support independent operations if the default, non-collective behavior is kept. (Otherwise these operations are collective.)

³ We usually open iterations delayed on first access. This first access is usually the `flush()` call after a `storeChunk/loadChunk` operation. If the first access is non-collective, an explicit, collective `Iteration::open()` can be used to have the files already open.

² *HDF5* only supports collective attribute definitions/writes; *ADIOS1* and *ADIOS2* attributes can be written independently. If you want to support all backends equally, treat as a collective operation.

4.4 Backend-Specific Configuration

While the openPMD API intends to be a backend-*independent* implementation of the openPMD standard, it is sometimes useful to pass configuration parameters to the specific backend in use. *For each backend*, configuration options can be passed via a JSON-formatted string or via environment variables. A JSON option always takes precedence over an environment variable.

The fundamental structure of this JSON configuration string is given as follows:

```
{
  "adios": "put ADIOS config here",
  "adios2": "put ADIOS2 config here",
  "hdf5": "put HDF5 config here",
  "json": "put JSON config here"
}
```

This structure allows keeping one configuration string for several backends at once, with the concrete backend configuration being chosen upon choosing the backend itself.

The configuration is read in a case-sensitive manner. Generally, keys of the configuration are *lower case*. Parameters that are directly passed through to an external library and not interpreted within openPMD API (e.g. `adios2.engine.parameters`) are unaffected by this and follow the respective library's conventions.

The configuration string may refer to the complete `openPMD::Series` or may additionally be specified per `openPMD::Dataset`, passed in the respective constructors. This reflects the fact that certain backend-specific parameters may refer to the whole Series (such as storage engines and their parameters) and others refer to actual datasets (such as compression).

For a consistent user interface, backends shall follow the following rules:

- The configuration structures for the Series and for each dataset should be defined equivalently.
- Any setting referring to single datasets should also be applicable globally, affecting all datasets.
- If a setting is defined globally, but also for a concrete dataset, the dataset-specific setting should override the global one.
- If a setting is passed to a dataset that only makes sense globally (such as the storage engine), the setting should be ignored except for printing a warning. Backends should define clearly which keys are applicable to datasets and which are not.

4.4.1 Configuration Structure per Backend

ADIOS2

A full configuration of the ADIOS2 backend:

```
{
  "adios2": {
    "engine": {
      "type": "sst",
      "parameters": {
        "BufferGrowthFactor": "2.0",
        "QueueLimit": "2"
      }
    },
    "dataset": {
      "operators": [
        {
          "type": "blosc",
          "parameters": {
            "clevel": "1",

```

(continues on next page)

(continued from previous page)

```
        "doshuffle": "BLOSC_BITSHUFFLE"
    }
}
]
```

All keys found under `adios2.dataset` are applicable globally as well as per dataset, keys found under `adios2.engine` only globally. Explanation of the single keys:

- `adios2.engine.type`: A string that is passed directly to `adios2::IO::SetEngine` for choosing the ADIOS2 engine to be used. Please refer to the [official ADIOS2 documentation](#) for a list of available engines.
- `adios2.engine.parameters`: An associative array of string-formatted engine parameters, passed directly through to `adios2::IO::SetParameters`. Please refer to the official ADIOS2 documentation for the allowable engine parameters.
- `adios2.engine.usesteps`: Described more closely in the documentation for the *ADIOS2 backend*.
- `adios2.dataset.operators`: This key contains a list of ADIOS2 [operators](#), used to enable compression or dataset transformations. Each object in the list has two keys:
 - `type` supported ADIOS operator type, e.g. `zfp`, `sz`
 - `parameters` is an associative map of string parameters for the operator (e.g. compression levels)

Any setting specified under `adios2.dataset` is applicable globally as well as on a per-dataset level. Any setting under `adios2.engine` is applicable globally only.

Other backends

Do currently not read the configuration string. Please refer to the respective backends' documentations for further information on their configuration.

UTILITIES

5.1 Command Line Tools

openPMD-api installs command line tools alongside the main library. These terminal-focused tools help to quickly explore, manage or manipulate openPMD data series.

5.1.1 openpmd-ls

List information about an openPMD data series.

The syntax of the command line tool is printed via:

```
openpmd-ls --help
```

With some pip-based python installations, you might have to run this as a module:

```
python -m openpmd_api.ls --help
```

5.2 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.

5.2.1 Example Usage

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

#ifdef openPMD_HAVE_MPI
#   include <mpi.h>
#endif

#include <iostream>

#ifdef 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;
```

(continues on next page)

(continued from previous page)

```

#if openPMD_HAVE_ADIOS1 || openPMD_HAVE_ADIOS2 || 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,
        2000000000
    );
    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.
    // 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:

```

(continues on next page)

(continued from previous page)

```

    // * 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 || openPMD_HAVE_ADIOS2
        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)
                << "\t with backend "
                << std::get<res.BACKEND>(it->first)
                << "\t write time: "
                << std::chrono::duration_cast<std::chrono::milliseconds>(
                    time.first
                ).count() << "\t read time: "
                << std::chrono::duration_cast<std::chrono::milliseconds>(
                    time.second
                ).count() << std::endl;
        }
    }

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

```

BACKENDS

6.1 Overview

This section provides an overview of features in I/O backends.

Feature	ADIOS1	ADIOS2	HDF5	JSON
Operating Systems	Linux, OSX	Linux, OSX, Windows		
Serial	supported	supported	supported	supported
MPI-parallel	supported	supported	supported	no
Dataset deletion	no	no	supported	supported
Compression	upcoming	supported	upcoming	no
Streaming/Staging	not exposed	upcoming	no	no
Portable Files	limited	awaiting	yes	yes
PByte-scalable	yes	yes	no	no
Performance	A	TBD	B	C
Native File Format	.bp (BP3)	.bp (BP4)	.h5	.json

- supported/yes: implemented and accessible for users of openPMD-api
- upcoming: planned for upcoming releases of openPMD-api
- limited: for example, limited to certain datatypes
- awaiting: planned for upcoming releases of a dependency
- TBD: to be determined (e.g. with upcoming benchmarks)

6.1.1 Selected References

- 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](https://arxiv.org/abs/1706.00522), DOI:10.1007/978-3-319-67630-2_2

6.2 JSON

openPMD supports writing to and reading from JSON files. The JSON backend is always available.

6.2.1 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`.

6.2.2 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.

6.2.3 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": [
                "x"
              ]
            }
          },
          "dataOrder": {
            "datatype": "STRING",
```

(continues on next page)

(continued from previous page)

```

        "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
    ],
    [
        3,
        4,
        5
    ],
    [

```

(continues on next page)

(continued from previous page)

```

        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
}
}

```

6.3 ADIOS1

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](#).

Note: This backend is deprecated, please use ADIOS2 instead.

6.3.1 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.

6.3.2 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	de- fault	description
OPENPMD_ADIOS_NUM_AGGREGATORS	ADIOS1	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).
OPENPMD_BP_BACKEND	ADIOS2	Chose preferred .bp file backend if ADIOS1 and ADIOS2 are available.

Please refer to the [ADIOS1 manual](#), section 6.1.5 for details on I/O tuning.

In case both the ADIOS1 backend and the *ADIOS2 backend* are enabled, set `OPENPMD_BP_BACKEND` to `ADIOS1` to enforce using ADIOS1. If only the ADIOS1 backend was compiled but not the *ADIOS2 backend*, the default of `OPENPMD_BP_BACKEND` is automatically switched to `ADIOS1`. Be advised that ADIOS1 only supports .bp files up to the internal version BP3, while ADIOS2 supports BP3, BP4 and later formats.

6.3.3 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-scale I/O is a research topic after all.

6.3.4 Limitations

Note: You cannot initialize and use more than one `openPMD::Series` with ADIOS1 backend at the same time in a process, even if both Series operate on totally independent data. This is an upstream bug in ADIOS1 that we cannot control: ADIOS1 cannot be initialized more than once, probably because it shares some internal state.

Note: The way we currently implement ADIOS1 in openPMD-api is sub-ideal and we close/re-open file handles way too often. Consequently, this can lead to severe performance degradation unless fixed. Mea culpa, we did better in the past (in PIconGPU). Please consider using our ADIOS2 backend instead, on which we focus our developments these days.

Note: ADIOS1 does not support attributes that are arrays of complex types.

6.3.5 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

6.4 ADIOS2

openPMD supports writing to and reading from ADIOS2 `.bp` files. For this, the installed copy of openPMD must have been built with support for the ADIOS2 backend. To build openPMD with support for ADIOS2, use the CMake option `-DopenPMD_USE_ADIOS2=ON`. For further information, check out the *installation guide*, *build dependencies* and the *build options*.

6.4.1 I/O Method

ADIOS2 has several engines for alternative file formats and other kinds of backends, yet natively writes to `.bp` files. The openPMD API uses the BP4 engine as the default file engine and the SST engine for streaming support. We currently leverage the default ADIOS2 transport parameters, i.e. `POSIX` on Unix systems and `FStream` on Windows.

6.4.2 Steps

ADIOS2 is optimized towards organizing the process of reading/writing data into IO steps. In order to activate steps, it is imperative to use the *Streaming API* (which can be used for either file-based or streaming-based workflows). With ADIOS2 release 2.6.0 containing a bug (fixed in development versions, see [PR #2348](#)) that disallows random-accessing steps in file-based engines, step-based processing must currently be opted in to via use of the *JSON parameter* `adios2.engine.usesteps = true` when using a file-based engine such as BP3 or BP4. With these ADIOS2 releases, files written in such a way may only be read using the streaming API. Upon reading a file, the ADIOS2 backend will automatically recognize whether it has been written with or without steps, ignoring the JSON option mentioned above. Steps are mandatory for streaming-based engines and trying to switch them off will result in a runtime error.

Note: ADIOS2 will in general dump data to disk/transport only upon closing a file/engine or a step. If not using steps, users are hence strongly encouraged to use file-based iteration layout (by creating a Series with a filename pattern such as `simData_%06T.bp`) and enforce dumping to disk by `Iteration::close()` -ing an iteration after writing to it. Otherwise, out-of-memory errors are likely to occur.

6.4.3 Backend-Specific Controls

The ADIOS2 SST engine for streaming can be picked by specifying the ending `.sst` instead of `.bp`.

The following environment variables control ADIOS2 I/O behavior at runtime. Fine-tuning these is especially useful when running at large scale.

environment variable	de- fault	description
OPENPMD_ADIOS2_HAVE_PROFILING	1	Turns on/off profiling information right after a run.
OPENPMD_ADIOS2_HAVE_METADATA_FILE	1	Online creation of the adios journal file (1: yes, 0: no).
OPENPMD_ADIOS2_NUM_SUBSTREAMS	0	Number of files to be created, 0 indicates maximum number possible.
OPENPMD_ADIOS2_ENGINE	File	ADIOS2 engine
OPENPMD_BP_BACKEND	ADIOS2	Chose preferred <code>.bp</code> file backend if ADIOS1 and ADIOS2 are available.

Please refer to the [ADIOS2 documentation](#) for details on I/O tuning.

In case the ADIOS2 backend was not compiled but only the deprecated [ADIOS1 backend](#), the default of `OPENPMD_BP_BACKEND` will fall back to ADIOS1. Be advised that ADIOS1 only supports `.bp` files up to the internal version BP3, while ADIOS2 supports BP3, BP4 and later formats.

Notice that the ADIOS2 backend is alternatively configurable via [JSON parameters](#).

Due to performance considerations, the ADIOS2 backend configures ADIOS2 not to compute any dataset statistics (Min/Max) by default. Statistics may be activated by setting the [JSON parameter](#) `adios2.engine.parameters.StatsLevel = "1"`.

6.4.4 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. Note that such a tool is not yet available for ADIOS2, but the `bpmeta` utility provided by ADIOS1 is capable of processing files written by ADIOS2.

Further options depend heavily on filesystem type, specific file striping, network infrastructure and available RAM on the aggregator nodes. A good number for substreams is usually the number of contributing nodes divided by four.

For fine-tuning at extreme scale or for exotic systems, please refer to the ADIOS2 manual and talk to your filesystem admins and the ADIOS2 authors. Be aware that extreme-scale I/O is a research topic after all.

6.4.5 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

6.5 HDF5

openPMD supports writing to and reading from HDF5 `.h5` files. For this, the installed copy of openPMD must have been built with support for the HDF5 backend. To build openPMD with support for HDF5, use the CMake option `-DopenPMD_USE_HDF5=ON`. For further information, check out the [installation guide](#), [build dependencies](#) and the [build options](#).

6.5.1 I/O Method

HDF5 internally either writes serially, via `POSIX` on Unix systems, or parallel to a single logical file via `MPI-I/O`.

6.5.2 Backend-Specific Controls

The following environment variables control HDF5 I/O behavior at runtime.

environment variable	de- fault	description
<code>OPENPMD_HDF5_INDEPENDENT</code>	<code>OFF</code>	Sets the <code>MPI</code> -parallel transfer mode to collective (<code>OFF</code>) or independent (<code>ON</code>).
<code>OPENPMD_HDF5_ALIGNMENT</code>		Tuning parameter for parallel I/O, choose an alignment which is a multiple of the disk block size.
<code>H5_COLL_API_SANITY_CHECK</code>	<code>OFF</code>	Set to 1 to perform an <code>MPI_Barrier</code> inside each meta-data operation.

`OPENPMD_HDF5_INDEPENDENT`: by default, we implement `MPI`-parallel data `storeChunk` (write) and `loadChunk` (read) calls as [none-collective MPI operations](#). Attribute writes are always collective in parallel HDF5. Although we choose the default to be non-collective (independent) for ease of use, be advised that performance penalties may occur, although this depends heavily on the use-case. For independent parallel I/O, potentially prefer using a modern version of the `MPICH` implementation (especially, use `ROMIO` instead of `OpenMPI`'s `ompio` implementation). Please refer to the [HDF5 manual](#), [function `H5Pset_dxpl_mpio`](#) for more details.

`OPENPMD_HDF5_ALIGNMENT` This sets the alignment in Bytes for writes via the `H5Pset_alignment` function. According to the [HDF5 documentation](#): *For MPI IO and other parallel systems, choose an alignment which is a multiple of the disk block size*. On Lustre filesystems, according to the [NERSC documentation](#), it is advised to set this to the Lustre stripe size. In addition, ORNL Summit GPFS users are recommended to set the alignment value to 16777216(16MB).

`H5_COLL_API_SANITY_CHECK`: this is a HDF5 control option for debugging parallel I/O logic (API calls). Debugging a parallel program with that option enabled can help to spot bugs such as collective `MPI`-calls that are not called by all participating `MPI` ranks. Do not use in production, this will slow parallel I/O operations down.

6.5.3 Selected References

- [GitHub issue #554](#)
- 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

DEVELOPMENT

7.1 Contribution Guide

7.1.1 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 priorities and progress in the [projects tab](#).

7.1.2 Style Guide

For coding style, please try to follow the guides in [ComputationalRadiationPhysics/contributing](#) for new code.

7.2 Repository Structure

7.2.1 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 candidate 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

7.2.2 Directory Structure

- `include/`
 - C++ header files
 - `set -I` here
 - prefixed with project name
 - `auxiliary/`
 - * internal auxiliary functionality
 - `helper/, benchmark/`
 - * user-facing helper functionality

- `src/`
 - C++ source files
 - `cli/`
 - * user-facing command line tools
- `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: `share/openPMD/download_samples.sh` (or `.ps1`)
- `share/openPMD/`
 - download scripts for example files
 - `cmake/`
 - * `cmake` scripts
 - `thirdParty/`
 - * included third party software
- `test/`
 - unit tests which are run with `ctest` (`make test`)
- `.github/`
 - GitHub issue/pull request templates
 - `workflows/`
 - * GitHub Action scripts for continuous integration checks
 - `ci/`
 - * service-agnostic configurations for continuous integration
- `docs/`
 - documentation files

7.3 Design Overview

Note: This section is a stub. Please open a pull-request to improve it or open an issue with open questions.

This library consists of three conceptual parts:

- The backend, concerned with elementary low-level I/O operations.
- The I/O-Queue, acting as a communication mechanism and buffer between the other two parts.
- The user-facing frontend, enforcing the openPMD standard, keeping a logical state of the data, and synchronizing that state with persistent data by scheduling I/O-Tasks.

7.3.1 Backend

One of the main goals of this library is to provide a high-level common interface to synchronize persistent data with a volatile representation in memory. This includes handling data in any number of supported file formats transparently. Therefore, enabling users to handle hierarchical, self-describing file formats while disregarding the actual nitty-gritty details of just those file formats, required the reduction of possible operations reduced to a common set of [IOTasks](#):

```
CLOSE_PATH,
OPEN_PATH,
DELETE_PATH,
LIST_PATHS,

CREATE_DATASET,
EXTEND_DATASET,
OPEN_DATASET,
DELETE_DATASET,
WRITE_DATASET,
READ_DATASET,
LIST_DATASETS,

DELETE_ATT,
WRITE_ATT,
READ_ATT,
LIST_ATTS,

ADVANCE,
AVAILABLE_CHUNKS ///  
Query chunks that can be loaded in a dataset
}; // Operation

struct OPENPMDAPI_EXPORT AbstractParameter
```

Every task is designed to be a fully self-contained description of one such atomic operation. By describing a required minimal step of work (without any side-effect), these operations are the foundation of the unified handling mechanism across suitable file formats. The actual low-level exchange of data is implemented in `IOHandlers`, one per file format (possibly two if handling MPI-parallel work is possible and requires different behaviour). The only task of these `IOHandlers` is to execute one atomic `IOTask` at a time. Ideally, additional logic is contained to improve performance by keeping track of open file handles, deferring and coalescing parts of work, avoiding redundant operations. It should be noted that while this is desirable, sequential consistency must be guaranteed (see *I/O-Queue*.)

Note this paragraph is a stub: `AbstractParameter` and subclasses as typesafe descriptions of task parameters, `Writable` as unique identification in task, corresponding to node in frontend hierarchy (tree-like structure), subclass of `AbstractIOHandler` to ensure simple extensibility, and only two public interface methods (`enqueue()` and `flush()`) to hide separate behaviour & state `AbstractFilePosition` as a format-dependent location inside persistent data (e.g. node-id / path string) should be entirely agnostic to openPMD and just treat transferred data as raw bytes without *knowledge*

7.3.2 I/O-Queue

To keep coupling between openPMD logic and actual low-level I/O to a minimum, a sequence of atomic I/O-Tasks is used to transfer data between logical and physical representation. Individual tasks are scheduled by frontend application logic and stored in a data structure that allows for FIFO order processing (in future releases, this order might be relaxed). Tasks are not executed during their creation, but are instead buffered in this queue. Disk accesses can be coalesced and high access latencies can be amortized by performing multiple tasks bunched together. At appropriate points in time, the used backend processes all pending tasks (strict, single-threaded, synchronous FIFO is currently used in all backends, but is not mandatory as long as consistency with that order can be guaranteed).

A typical sequence of tasks that are scheduled during the read of an existing file *could* look something like this:

```
1. OPEN_FILE
2. READ_ATT      // 'openPMD'
3. READ_ATT      // 'openPMDextension'
4. READ_ATT      // 'basePath'
### PROCESS ELEMENTS ###
5. LIST_ATTS     // in '/'
### PROCESS ELEMENTS ###
5.1 READ_ATT     // 'meshesPath', if in 5.
5.2 READ_ATT     // 'particlesPath', if in 5.
### PROCESS ELEMENTS ###
6. OPEN_PATH     // 'basePath'
7. LIST_ATTS     // in 'basePath'
### PROCESS ELEMENTS ###
7.X READ_ATT     // every 'att' in 7.
8. LIST_PATHS    // in 'basePath'
### PROCESS ELEMENTS ###
9.X OPEN_PATH    // every 'path' in 8.
...
```

Note that (especially for reading), pending tasks might have to be processed between any two steps to guarantee data consistency. That is because action might have to be taken conditionally on read or written values, openPMD conformity checked to fail fast, or a processing of the tasks be requested by the user explicitly.

As such, FIFO-equivalence with the scheduling order must be satisfied. A task that is not located at the head of the queue (i.e. does not have the earliest schedule time of all pending tasks) is not guaranteed to succeed in isolation. Currently, this can only be guaranteed by sequentially performing all tasks scheduled prior to it in chronological order. To give two examples where this matters:

- Reading value chunks from a dataset only works after the dataset has been opened. Due to limitations in some of the backends and the atomic nature of the I/O-tasks in this API (i.e. operations without side effects), datatype and extent of a dataset are only obtained by opening the dataset. For some backends this information is required for chunk reading and thus must be known prior to performing the read.
- **Consecutive chunk writing and reading (to the same dataset) mirrors classical RAW data dependence.**
The two chunks might overlap, in which case the read has to reflect the value changes introduced by the write.

Atomic operations contained in this queue are ...

7.3.3 Frontend

While the other two components are primarily concerned with actual I/O, this one is the glue and constraint logic that lets a user build the in-memory view of the hierarchical file structure. Public interfaces should be limited to this part (exceptions may arise, e.g. format-dependent dataset parameters). Where the other parts contain virtually zero knowledge about openPMD, this one contains all of it and none of the low-level I/O.

`Writable` (mixin) base class of every front-end class, used to tree structure used in backend

`Attributable` (mixin) class that allows attaching meta-data to tree nodes (openPMD attributes)

`Attribute` a variadic datastore for attributes supported across backends

Container serves two purposes

- python-esque access inside hierarchy groups (`foo["bar"]["baz"]`)
- only way for user to construct objects (private constructors), forces them into the correct hierarchy (no dangling objects)

all meta-data access stores in the `Attributable` part of an object and follows the syntax

```
Object& setFoo(Foo foo);
Foo foo() const;
```

(future work: use [CRTP](#))

Series as root of every hierarchy, supporting `groupBased` and `fileBased` transparently ...

7.4 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.

7.4.1 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.

7.4.2 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, Access);
    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, Access 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,
    Access 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,
    Access 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.

7.4.3 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;

```

(continues on next page)

(continued from previous page)

```

    void deleteAttribute(Writable*, Parameter< Operation::DELETE_ATT > const&) override;
    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_ATT > &) 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;
}
}

```

(continues on next page)

(continued from previous page)

```
};
} // 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).

7.5 Build Dependencies

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

7.5.1 Required

- CMake 3.15.0+
- C++14 capable compiler, e.g. g++ 5.0+, clang 5.0+, VS 2017+

7.5.2 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](#) 2.6.1+ (new BSD)
- [NLOhmann-JSON](#) 3.9.1+ (MIT)

7.5.3 Optional: I/O backends

- [JSON](#)
- [HDF5](#) 1.8.13+
- [ADIOS1](#) 1.13.1+
- [ADIOS2](#) 2.6.0+

while those can be build either with or without:

- [MPI](#) 2.1+, e.g. [OpenMPI](#) 1.6.5+ or [MPICH2](#)

7.5.4 Optional: language bindings

- Python:
 - Python 3.6 - 3.9
 - [pybind11](#) 2.6.1+
 - [numpy](#) 1.15+
 - [mpi4py](#) 2.1+

7.5.5 Quick Install with Spack

Quickly install all dependencies with a [Spack anonymous environment](#). Go in the base directory and type:

```
spack env activate -d .
spack install
```

7.6 Build Options

7.6.1 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
<code>openPMD_USE_MPI</code>	AUTO/ON/OFF	Parallel, Multi-Node I/O for clusters
<code>openPMD_USE_HDF5</code>	AUTO/ON/OFF	HDF5 backend (<code>.h5</code> files)
<code>openPMD_USE_ADIOS1</code>	AUTO/ON/OFF	ADIOS1 backend (<code>.bp</code> files up to version BP3)
<code>openPMD_USE_ADIOS2</code>	AUTO/ON/OFF	ADIOS2 backend (<code>.bp</code> files in BP3, BP4 or higher)
<code>openPMD_USE_PYTHON</code>	AUTO/ON/OFF	Enable Python bindings
<code>openPMD_USE_INVASIVE_TESTS</code>	ON/OFF	Enable unit tests that modify source code ¹
<code>openPMD_USE_VERIFY</code>	ON/OFF	Enable internal <code>VERIFY</code> (assert) macro independent of build type ²
<code>openPMD_INSTALL</code>	ON/OFF	Add installation targets
<code>Python_EXECUTABLE</code>	(newest found)	Path to Python executable

¹ e.g. changes C++ visibility keywords, breaks MSVC

² this includes most pre-/post-condition checks, disabling without specific cause is highly discouraged

7.6.2 Shared or Static

By default, we will build as a shared 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.dylib` or `openPMD.dll`) at a time.

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

CMake Option	Values	Description
<code>BUILD_SHARED_LIBS</code>	ON/OFF	Build the C++ API as shared library
<code>HDF5_USE_STATIC_LIBRARIES</code>	ON/OFF	Require static HDF5 library
<code>ADIOS_USE_STATIC_LIBS</code>	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.

7.6.3 Debug

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

7.6.4 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
<code>openPMD_USE_INTERNAL_VARIANT</code>	ON/OFF	Yes	MPark.Variant	1.4.0+
<code>openPMD_USE_INTERNAL_CATCH</code>	ON/OFF	No	Catch2	2.6.1+
<code>openPMD_USE_INTERNAL_PYBIND11</code>	ON/OFF	No	pybind11	2.6.1+
<code>openPMD_USE_INTERNAL_JSON</code>	ON/OFF	No	NLohmann-JSON	3.9.1+

7.6.5 Tests, Examples and Command Line Tools

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

7.7 Sphinx

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.

7.7.1 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 graphviz

# python tools & style theme
python -m 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/

# render the `.rst` files and replace their macros within
# enjoy the breathe errors on things it does not understand from doxygen :)
make html
```

(continues on next page)

(continued from previous page)

```
# 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
```

7.7.2 Useful Links

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

MAINTENANCE

8.1 Release Channels

8.1.1 Spack

Our recommended HPC release channel when in need for MPI. Also very useful for Linux and OSX desktop releases. Supports all variants of openPMD-api via flexible user-level controls. The same install workflow used to bundle this release also comes in handy to test a development version quickly with power-users.

Example workflow for a new release:

- <https://github.com/spack/spack/pull/14018>

Please ping @ax3l in your pull-request.

8.1.2 Conda-Forge

Our primary release channel for desktops via a fully automated binary distribution. Provides the C++ and Python API for users. Supports Windows, OSX, and Linux. Packages are built with and without MPI, the latter is the default variant.

Example workflow for a new release:

- <https://github.com/conda-forge/openpmd-api-feedstock/pull/41>

8.1.3 Brew

We maintain a [homebrew tap](#) for openPMD. Provides the C++ and Python API for users. Supports OSX and Linux. Its source-only Formula for the latest release includes (Open)MPI support and lacks the ADIOS1 backend.

Example workflow for a new release:

- <https://github.com/openPMD/homebrew-openPMD/commit/839c458f1e8fa2a40ad0b4fd7d0d239d1062f867>

8.1.4 PyPI

Our PyPI release provides our Python bindings in a self-contained way, without providing access to the C++ API. On PyPI, we upload a source package with all build-variants to default (AUTO), but MPI (OFF) unless activated. Furthermore, we build portable, serial (non-MPI) binary wheels for Linux ([manylinux2010](#)) and macOS (10.9+).

The deployment of our binary wheels is automated via [cibuildwheel](#). Update the version number with a new git tag in the [wheels branch](#) to trigger an automated deployment to [pypi.org/project/openPMD-api](#). A push (merge) to this branch will build and upload all wheels together with the source distribution through [twine](#).

The same `pip` install workflow used to bundle this release also comes in handy to [test a development version quickly with power-users](#).

Example workflow for a new release:

- <https://github.com/openPMD/openPMD-api/pull/774>

8.1.5 ReadTheDocs

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 - Advanced Settings](#).

8.1.6 Doxygen

In order to update the *latest* Doxygen C++ API docs, located under <http://www.openPMD.org/openPMD-api/>, 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 add .
git commit
git push

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

Note that we publish per-release versions of the *Doxygen HTML pages* automatically on ReadTheDocs.