openPMD-api Documentation

Release 0.16.0-dev

The openPMD Community

Apr 22, 2024

CONTENTS

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, ADIOS2 and JSON. Writing & reading through those backends and their associated files is supported for serial and MPI-parallel workflows.

CHAPTER

ONE

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)
`0.13.1-0.15.1` (beta)	`1.0.0-1.1.0`
`0.1.0-0.12.0` (alpha)	`1.0.0-1.1.0`

CHAPTER

TWO

FUNDING ACKNOWLEDGEMENTS

The openPMD-api authors acknowledge support via the following programs. Supported by the CAMPA collaboration, a project of the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of High Energy Physics, Scientific Discovery through Advanced Computing (SciDAC) program. Previously supported by the Consortium for Advanced Modeling of Particles Accelerators (CAMPA), funded by the U.S. DOE Office of Science under Contract No. DE-AC02-05CH11231. Supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration). This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 654220. This work was partially funded by the Center of Advanced Systems Understanding (CASUS), which is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament. Supported by the HElmholtz Laser Plasma Metadata Initiative (HELPMI) project (ZT-I-PF-3-066), funded by the "Initiative and Networking Fund" of the Helmholtz Association in the framework of the "Helmholtz Metadata Collaboration" project call 2022.

2.1 Code of Conduct

2.1.1 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.

2.1.2 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

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, 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

2.2 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.

2.2.1 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, Franz Poeschel, Carsten Fortmann-Grote, Angel Ferran Pousa, Juncheng E, Maxence Thevenet and Michael Bussmann. *"openPMD: A meta data standard for particle and mesh based data,"* DOI:10.5281/zenodo.591699 (2015)

The equivalent BibTeX code is:

```
@misc{openPMDstandard,
  author       = {Huebl, Axel and
                  Lehe, R{\'e}mi and
                  Vay, Jean-Luc and
                  Grote, David P. and
                  Sbalzarini, Ivo and
                  Kuschel, Stephan and
                  Sagan, David and
                  Mayes, Christopher and
                  P{\'e}rez, Fr{\'e}d{\'e}ric and
                  Koller, Fabian and
                  Poeschel, Franz and
                  Fortmann-Grote, Carsten and
                  Ferran Pousa, Angel and
                  E, Juncheng and
                  Th{\'e}venet, Maxence and
                  Bussmann, Michael},
  title        = {{openPMD: A meta data standard for particle and mesh based data}},
  year         = 2015,
  publisher    = {Zenodo},
  doi          = {10.5281/zenodo.591699},
  url          = {https://www.openPMD.org},
  howpublished = {https://github.com/openPMD}
}
```

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)

2.2.2 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 0.14.3:

Tip: Axel Huebl, Franz Poeschel, Fabian Koller, Junmin Gu, Michael Bussmann, Jean-Luc Vay and Kesheng Wu. *"openPMD-api 0.14.3: C++ & Python API for Scientific I/O with openPMD,"* DOI:10.14278/rodare.1234 (2021)

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: Axel Huebl, Franz Poeschel, Fabian Koller, Junmin Gu, Michael Bussmann, Jean-Luc Vay and Kesheng Wu. *"openPMD-api: C++ & Python API for Scientific I/O with openPMD"* DOI:10.14278/rodare.27 (2018)

The equivalent BibTeX code is:

```
@misc{openPMDapi,
  author       = {Huebl, Axel and
                  Poeschel, Franz and
                  Koller, Fabian and
                  Gu, Junmin and
                  Bussmann, Michael and
                  Vay, Jean-Luc and
                  Wu, Kesheng},
  title        = {{openPMD-api: C++ \& Python API for Scientific I/O with openPMD}},
  month        = june,
  year         = 2018,
  doi          = {10.14278/rodare.27},
  url          = {https://github.com/openPMD/openPMD-api}
}
```

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++17

```cpp
#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

```python
import openpmd_api as io

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

CHAPTER

THREE

INSTALLATION

3.1 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:

3.1.1 Using the Spack Package

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

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

3.1.2 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
```

3.1.3 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 and includes (Open)MPI support.

3.1.4 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, 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 packaging setuptools wheel
python3 -m pip install -U cmake

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

For some exotic architectures and compilers, you might need to disable a compiler feature called link-time/interprocedural optimization if you encounter linking problems:

```
export CMAKE_INTERPROCEDURAL_OPTIMIZATION=OFF
# optional:                                                --user
python3 -m pip install openpmd-api --no-binary openpmd-api
```

Additional CMake options can be passed via individual environment variables, which need to be prefixed with `openPMD_CMAKE_`.

3.1.5 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=$HOME/somepath
# for options append:
#   -DopenPMD_USE_...=...
# e.g. for python support add:
#   -DopenPMD_USE_PYTHON=ON -DPython_EXECUTABLE=$(which python3)
cmake ../openPMD-api
```

(continues on next page)

(continued from previous page)

```
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 one 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%
```

3.2 Changelog

3.2.1 0.16.0

Date: TBA

[Title]

[Summary]

Changes to "0.15.0"

Features

- pybind11: require version 2.11.1+ #1220 #1322

Bug Fixes

Other

3.2.2 0.15.1

Date: 2023-04-02

Build Regressions

This release fixes build regressions and minor documentation updates for the 0.15.0 release.

Changes to "0.15.0"

Bug Fixes

- Build issues:
  - CMake: Fix Python Install Directory #1393
  - Work-Around: libc++ shared_ptr array #1409
  - Artifact Placement in Windows Wheels #1400
  - macOS AppleClang12 Fixes #1395
  - ADIOS1:
    - ADIOS1 on macOS #1396
    - If no ADIOS1, then add ADIOS1 sources to main lib #1407
    - Instantiate only parallel ADIOS1 IO Handler in parallel ADIOS1 lib #1411

Other

- Docker: CMake 3.24+: ZLIB_USE_STATIC_LIBS (#1410
- CI:
  - Test on Ubuntu 20.04 #1408
  - clang-format also for `.tpp` and `.hpp.in` files #1403
- docs:
  - update funding #1412
  - HTML5: CSS updates #1397 #1413
  - README: Remove LGTM Batches #1402
  - Docs TOML and ADIOS2 best practices #1404
  - Docs: ADIOS1 EOL in Overview #1398
  - Releases: Nils Schild (IPP) #1394
  - Formatting of lists in 0.15.0 changelog #1399

3.2.3 0.15.0

Date: 2023-03-25

C++17, Error Recovery, ADIOS2 BP5, Append & Read-Linear Modes, Performance & Memory

This release adds error recovery mechanisms, in order to access erroneous datasets, created e.g. by crashing simulations. The BP5 engine of ADIOS2 v2.9 is fully supported by this release, including access to its various features for more fine-grained control of memory usage. Various I/O performance improvements for HDF5 are activated by default. Runtime configuration of openPMD and its backends, e.g. selection of backends and compression, is now consistently done via JSON, and alternatively via TOML for better readability. The data storage/retrieval API now consistently supports all common C++ pointer types (raw and smart pointers), implementing automatic memory optimizations for ADIOS2 BP5 if using unique pointers.

The miminum required C++ version is now C++17. Supported Python versions are Python 3.10 and 3.11.

Changes to "0.14.0"

Features

- Python: support of 3.10 and 3.11, removal of 3.6 #1323 #1139
- include internally shipped toml11 v3.7.1 #1148 #1227
- pybind11: require version 2.10.1+ #1220 #1322
- Switch to C++17 #1103 #1128 #1140 #1157 #1164 #1183 #1185
- Error-recovery during parsing #1150 #1179 #1237
- Extensive update for JSON/TOML configuration #1043
  - TOML as an alternative to JSON #1146
  - compression configuration via JSON# 1043
  - case insensitivity #1043
  - datatype conversion for string values #1043
  - `json::merge` public function #1043 #1333

  - better warnings for unused values #1043
  - new JSON options: `backend` and `iteration_encoding` #1043
  - ADIOS1 compression configuration via JSON #1043 #1162
- New access types:
  - `APPEND`: Add new iterations without reading, supports ADIOS2 Append mode #1007 #1302
  - `READ_LINEAR`: For reading through ADIOS2 steps, for full support of ADIOS2 BP5 #1291 #1379
- ADIOS2:
  - Support for ADIOS 2.8 and newer #1166
  - Support for ADIOS2 BP5 engine #1119 #1215 #1258 #1262 #1291
  - Support for selecting flush targets (buffer/disk) in ADIOS2 BP5 for more fine-grained memory control #1226 #1207
  - Add file extensions for ADIOS2: `.bp4`, `.bp5` and furthers, make them behave more as expected #1218
  - ADIOS2: Support for operator specification at read time #1191
  - ADIOS2: Automatic (de)activation of span API depending on compression configuration #1155
  - Optionally explicitly map ADIOS2 steps to openPMD iterations via modifiable attributes (only supported in experimental ADIOS2 modes) #949
- HDF5:
  - I/O optimizations for HDF5 #1129 #1133 #1192
    - Improve write time by disabling fill #1192
- Miscellaneous API additions:
  - Support for all char types (CHAR SCHAR UCHAR) #1275 #1378
  - Header for openPMD-defined error types #1080 #1355
  - Add `Series::close()` API call #1324
  - Support for array specializations of C++ smart pointer types #1296
  - Direct support for raw pointer types in `store/loadChunk()` API, replacing former `shareRaw()` #1229
  - Support for and backend optimizations (ADIOS2 BP5) based on unique pointer types in `store/loadChunk()` #1294
  - Use C++ `std::optional` types in public Attribute API (`Attribute::getOptional<T>()`) for dynamic attribute type conversion #1278
- Support for empty string attributes #1087 #1223 #1338
- Support for inconsistent and overflowing padding of filenames in file-based encoding #1118 #1173 #1253

Bug Fixes

- HDF5
  - Support attribute reads from HDF5 Vlen Strings #1084
  - Close HFD5 handles in availableChunks task #1386
- ADIOS1
  - Fix use-after-free issue in `ADIOS1IOHandler` #1224
- ADIOS2

  - Don't apply compression operators multiple times #1152
  - Fix logic for associating openPMD objects to files and paths therein (needed for interleaved write and close) #1073
  - Fix precedence of environment variable vs. JSON configuration
  - Detect changing datatypes and warn/fail accordingly #1356
  - Remove deprecated debug parameter in ADIOS2 #1269
- HDF5
  - missing HDF5 include #1236
- CMake:
  - MPI: prefer HDF5 in Config package, too #1340
  - ADIOS1: do not include as `-isystem` #1076
  - Remove caching of global CMake variables #1313
  - Fix Build & Install Option Names #1326
  - Prefer parallel HDF5 in find_package in downstream use #1340
  - CMake: Multi-Config Generator #1384
- Warnings:
  - Avoid copying std::string in for loop #1268
  - SerialIOTest: Fix GCC Pragma Check #1213 #1260
  - Fix `-Wsign-compare` #1202
- Python:
  - Fix `__repr__` (time and Iteration) #1242 #1149
  - Python Tests: Fix `long` Numpy Type #1348
  - use `double` as standard for attributes #1290 #1369kk
  - Fix `dtype_from_numpy` #1357
  - Wheels: Fix macOS arm64 (M1) builds #1233
  - Avoid use-after-free in Python bindings #1225
  - Patch MSVC pybind11 debug bug #1209
  - sign compare warning #1198
- Don't forget closing unmodified files #1083
- Diverse relaxations on attribute type conversions #1085 #1096 #1137
- Performance bug: Don't reread iterations that are already parsed #1089
- Performance bug: Don't flush prematurely #1264
- Avoid object slicing in Series class #1107
- Logical fixes for opening iterations #1239

Breaking Changes

- Deprecations
  - `Iteration::closedByWriter()` attribute #1088
  - `shareRaw` (replaced with raw- and unique-ptr overloads, see features section) #1229
  - ADIOS1 backend (deprecation notice has hints on upgrading to ADIOS2) #1314
- Redesign of public class structure
  - Apply frontend redesign to Container and deriving classes #1115 #1159
- Removal of APIs
  - `Dataset::transform`, `Dataset::compression` and `Dataset::chunksize` #1043

Note: See *NEWS.rst* for a more detailed upgrade guide.

Other

- Catch2: updated to 2.13.10 #1299 #1344
- Tests & Examples:
  - Test: Interleaved Write and Close #1073 #1078
  - Extend and fix examples 8a and 8b (bench write/read parallel) #1131 #1144 #1231 #1359 #1240 - support variable encoding #1131 - block located at top left corner was mistaken to read a block in the center #1131 - GPU support in example 8a #1240
  - Extensive Python example for Streaming API #1141
  - General overhaul of examples to newest API standards #1371
- CI
  - URL Check for broken links #1086
  - CI savings (abort prior push, draft skips most) #1116
  - Appveyor fixes for Python Executable #1127
  - Pre-commit and clang-format #1142 #1175 #1178 #1032 #1222 #1370
  - ADIOS1: Fix Serial Builds, CI: Clang 10->12 #1167
  - Upgrade NVHPC Apt repository #1241
  - Spack upgrade to v0.17.1 and further fixes #1244
  - Update CUDA repository key #1256
  - Switch from Conda to Mamba #1261
  - Remove `-Wno-deprecated-declarations` where possible #1246
  - Expand read-only permission tests #1272
  - Ensure that the CI also build against ADIOS2 v2.7.1 #1271
  - Build(deps): Bump s-weigand/setup-conda from 1.1.0 to 1.1.1 #1284
  - Style w/ Ubuntu 22.04 #1346
  - Add CodeQL workflow for GitHub code scanning #1345
  - Cache Action v3 #1358 #1362

  - Spack: No More `load -r` #1125
- CMake
  - Extra CMake Arg Control in `setup.py` #1199
  - Do not strip Python symbols in Debug #1219
  - Disable in-source builds #1079
  - Fixes for NVCC #1102 #1103 #1184
  - Set RPATHs on installed targets #1105
  - CMake 3.22+: Policy `CMP0127` #1165
  - Warning Flags First in `CXXFLAGS` #1172
- Docs
  - More easily findable documentation for `-DPython_EXECUTABLE` #1104 and lazy parsing #1111
  - HDF5 performance tuning and known issues #1129 #1132
  - HDF5: Document `HDF5_USE_FILE_LOCKING` #1106
  - SST/libfabric installation notes for Cray systems #1134
  - OpenMPI: Document `OMPI_MCA_io` Control #1114
  - Update Citation & Add BibTeX (#1168)
  - Fix CLI Highlighting #1171
  - HDF5 versions that support collective metadata #1250
  - Recommend Static Build for Superbuilds #1325
  - Latest Sphinx, Docutils, RTD #1341
- Tooling
  - `openpmd-pipe`: better optional support for MPI #1186 #1336
  - `openpmd-ls`: use lazy parsing #1111
- Enable use of `Series::setName()` and `Series::setIterationEncoding()` in combination with file-based encoding 1081
- Remove `DATATYPE`, `HIGHEST_DATATYPE` AND `LOWEST_DATATYPE` from Datatype enumeration #1100
- Check for undefined datatypes in dataset definitions #1099
- Include `StringManip` header into public headers #1124
- Add default constructor for `DynamicMemoryView` class #1156
- Helpful error message upon wrong backend specification #1214
- Helpful error message for errors in `loadChunk` API #1373
- No warning when opening a single file of a file-based Series #1368
- Add `IterationIndex_t` type alias #1285

3.2.4 0.14.5

Date: 2022-06-07

Improve Series Parsing, Python & Fix Backend Bugs

This release improves reading back iterations that overflow the specified zero-pattern. ADIOS1, ADIOS2 and HDF5 backend stability and performance were improved. Python bindings got additional wheel platform support and various smaller issues were fixed.

Changes to "0.14.4"

Bug Fixes

- Series and iterations:
  - fix read of overflowing zero patterns #1173 #1253
  - fix for opening an iteration #1239
- ADIOS1:
  - fix use-after-free in `ADIOS1IOHandler` #1224
  - Remove task from IO queue if it fails with exception #1179
- ADIOS2:
  - Remove deprecated debug parameter in ADIOS2 #1269
  - Add memory leak suppression: `ps_make_timer_name_` #1235
  - Don't safeguard empty strings while reading #1223
- HDF5:
  - missing HDF5 include #1236
- Python:
  - Wheels: Fix macOS arm64 (M1) builds #1233
  - Python Iteration: Fix `__repr__` (time) #1242
  - Increase reference count also in other `load_chunk` overload #1225
  - Do Not Strip Symbols In Debug #1219
  - Patch MSVC pybind11 debug bug #1209

Other

- HDF5:
  - Improve write time by disabling fill #1192
  - Update documented HDF5 versions with collective metadata issues #1250
- Print warning if mpi4py is not found in `openpmd-pipe` #1186
- Pass-through flushing parameters #1226
- Clang-Format #1032 #1222
- Warnings:
  - Avoid copying std::string in for loop #1268
  - SerialIOTest: Fix GCC Pragma Check #1213 #1260

  - Fix `-Wsign-compare` #1202
- CI:
  - Fix Conda Build - <3 Mamba #1261
  - Fix Spack #1244
  - Update CUDA repo key #1256
  - NVHPC New Apt Repo #1241
- Python:
  - `setup.py`: Extra CMake Arg Control #1199
  - sign compare warning #1198

3.2.5 0.14.4

Date: 2022-01-21

Increased Compatibility & Python Install Bug

This release fixes various read/parsing bugs and increases compatibility with upcoming versions of ADIOS and old releases of Intel `icpc`. An installation issue for pip-based installs from source in the last release was fixed and Python 3.10 support added. Various documentation and installation warnings have been fixed.

Changes to "0.14.3"

Bug Fixes

- ADIOS2:
  - automatically deactivate `span` based `Put` API when operators are present #1155
  - solve incompatibilities w/ post-`2.7.1` `master`-branch #1166
- ICC 19.1.2: C++17 work-arounds (`variant`) #1157
- Don't apply compression operators multiple times in variable-based iteration encoding #1152
- Reading/parsing:
  - remove invalid records from data structures entirely #1150
  - fix grid spacing with type long double #1137
- Python:
  - fix `Iteration` `__repr__` typo #1149
  - add `cmake/` to `MANIFEST.in` #1140

Other

- add simple `.pre-commit-config.yaml`
- Python:
  - support Python 3.10 #1139
- CMake:
  - warning flags first in `CXXFLAGS` #1172
  - add policy CMP0127 (v3.22+) #1165

- Docs:
  - fix CLI highlighting #1171
  - update citation & add BibTeX #1168
  - fix HDF5 JSON File #1169
  - minor warnings #1170

3.2.6 0.14.3

Date: 2021-11-03

Read Bugs, C++17 Mixing and HDF5 Performance

This release makes reads more robust by fixing small API, file-based parsing and test bugs. Building the library in C++14 and using it in C++17 will not result in incompatible ABIs anymore. HDF5 1.10.1+ performance was improved significantly.

Changes to "0.14.2"

Bug Fixes

- read:
  - allow inconsistent zero pads #1118
  - time/dt also in long double #1096
- test 8b - bench read parallel:
  - support variable encoding #1131
  - block located at top left corner was mistaken to read a block in the center #1131
- CI (AppVeyor): Python executable #1127
- C++17 mixing: remember <`variant`> implementation #1128
- support NVCC + C++17 #1103
- avoid object slicing when deriving from `Series` class #1107
- executables: `CXX_STANDARD/EXTENSIONS` #1102

Other

- HDF5 I/O optimizations #1129 #1132 #1133
- libfabric 1.6+: Document SST Work-Arounds #1134
- OpenMPI: Document `OMPI_MCA_io` Control #1114
- HDF5: Document `HDF5_USE_FILE_LOCKING` #1106
- Lazy parsing: Make findable in docs and use in `openpmd-ls` #1111
- Docs: More Locations `-DPython_EXECUTABLE` #1104
- Spack: No More `load -r` #1125
- `openPMD.hpp`: include auxiliary `StringManip` #1124

3.2.7 0.14.2

Date: 2021-08-17

Various Reader Fixes

This releases fixes regressions in reads, closing files properly, avoiding inefficient parsing and allowing more permissive casts in attribute reads. (Inofficial) support for HDF5 vlen string reads has been fixed.

Changes to "0.14.1"

Bug Fixes

- do not forget to close files #1083
- reading of vector attributes with only one contained value #1085
- do not read iterations if they have already been parsed #1089
- HDF5: fix string vlen attribute reads #1084

Other

- `setAttribute`: reject empty strings #1087

3.2.8 0.14.1

Date: 2021-08-04

ADIOS2 Close Regressions & ADIOS1 Build

Fix a regression with file handling for ADIOS2 when using explicit close logic, especially with interleaved writes to multiple iterations. Also fix an issue with ADIOS1 builds that potentially picked up headers from older, installed openPMD-api versions.

Changes to "0.14.0"

Bug Fixes

- ADIOS2: interleaved writes of iterations with close #1073
- CMake: ADIOS1 includes w/o `SYSTEM` #1076

3.2.9 0.14.0

Date: 2021-07-29

Resize, Dask, openpmd-pipe and new ADIOS2 Iteration Encoding

This release adds support for resizable data sets. For data-processing, support for Dask (parallel) and Pandas (serial) are added and lazy reader parsing of iterations is now supported. ADIOS2 adds an experimental variable-based iteration encoding. An openPMD Series can now be flushed from non-`Series` objects and write buffers can be requested upfront to avoid unnecessary data copies in some situations.

Changes to "0.13.4"

Features

- Resizable datasets #829 #1020 #1060 #1063
- lazy parsing of iterations #938
- Expose internal buffers to writers #901
- `seriesFlush`: Attributable, Writable, Mesh & ParticleSpecies #924 #925
- ADIOS2:
  - Implement new `variableBased` iteration encoding #813 #855 #926 #941 #1008
  - Set a default `QueueLimit` of 2 in the ADIOS2/SST engine #971
  - Add environment control: `OPENPMD_ADIOS2_STATS_LEVEL` #1003
- Conda environment file `conda.yaml` added to repo #1004
- CMake: Expose Python LTO Control #980
- HDF5:
  - HDF5 1.12.0 fallback APIs: no wrappers and more portable #1012
  - Empiric for optimal chunk size #916
- Python:
  - `ParticleSpecies`: Read to `pandas.DataFrame` #923
  - `ParticleSpecies`: Read to `dask.dataframe` #935 #951 #956 #958 #959 #1033
  - Dask: Array #952
  - `pyproject.toml`: build-backend #932
- Tools: add `openpmd-pipe.py` command line tool #904 #1062 #1069
- Support for custom geometries #1011
- Default constructors for `Series` and `SeriesIterator` #955
- Make `WriteIterations::key_type` public #999
- `ParticleSpecies` & `RecordComponent` serialize #963

Bug Fixes

- ADIOS2:
  - `bp4_steps` test: actually use `NullCore` engine #933
  - Always check the return status of `IO::Open()` and `Engine::BeginStep()` in ADIOS2 #1017 #1023
  - More obvious error message if datatype cannot be found #1036
  - Don't implicitly open files #1045
  - fix C++17 compilation #1067
- HDF5:
  - Support Parallel HDF5 built w/ CMake #1027
  - `HDF5Auxiliary`: Check String Sizes #979
- Tests:
  - Check for existence of the correct files in `ParallelIOtests` #944

  - FBPIC example filename #950
  - `CoreTest`: Lambda outside unevaluated context #1057
- `availableChunks`: improve open logic for early chunk reads #1035 #1045
- CMake:
  - custom copy for dependent files #1016
  - library type control #930
- Fix detection of `loadChunk()` calls with wrong type #1022
- Don't flush `Series` a second time after throwing an error #1018
- Use `Series::writeIterations()` without explicit flushing #1030
- `Mesh`: `enable_if` only floating point APIs #1042
- `Datatype`: Fix `std::array` template #1040
- PkgConfig w/ external variant #1050
- warnings: Unused params and unreachable code #1053 #1055

Other

- ADIOS2: require version 2.7.0+ #927
- Catch2: 2.13.4+ #940
- pybind11: require version 2.6.2+ #977
- CI:
  - Update & NVHPC #1052
  - ICC/ICPC & ICX/ICPX #870
  - Reintroduce Clang Sanitizer #947
  - Brew Update #970
  - Source Tools Update #978
  - Use specific commit for downloaded samples #1049
  - `SerialIOTest`: fix CI hang in sanitizer #1054 #1056
- CMake:
  - Require only C-in-CXX MPI component #710
  - Unused setter in `openpmd_option` #1015
- Docs:
  - describe high-level concepts #997
  - meaning of `Writable::written()` #946
  - `Iteration::close/flush` fix typo #988
  - `makeConstant` & parallel HDF5 #1041
  - ADIOS2 memory usage for various encoding schemes #1009
  - `dev`-branch centered development #928
  - limit docutils to 0.16, Sphinx to <4.0 #976
  - Sphinx: rsvg converter for LaTeX #1001
  - Update GitHub issue templates #1034

  - Add `CITATION.cff` #1070
  - Benchmark 8b: "pack" parameter #1066
  - Move quoted lines from `IOTasks` #1061
  - describe iteration encodings #1064
  - describe regexes for showing only attributes or datasets in new ADIOS2 schema #1068
- Tests & Examples:
  - ADIOS2 SST tests: start reader a second after the writer #981
  - ADIOS2 Git sample #1019 #1051
  - Parallel Benchmark (8): 4D is now 3D #1010 #1047
- `RecordComponent`: Remove unimplemented scaling #954
- MSVC: Proper `__cplusplus` macro #919
- Make `switchType` more comfortable to use #931
- Split `Series` into an internal and an external class #886 #936 #1031 #1065
- Series: `fileBased` more consequently throws `no_such_file_error` #1059
- Retrieve paths of objects in the openPMD hierarchy #966
- Remove duplicate function declarations #998
- License Header: Update 2021 #922
- Add Dependabot #929
- Update author order for 0.14.0+ #1005
- Download samples: optional directory #1039

3.2.10 0.13.4

Date: 2021-05-13

Fix AppleClang & DPC++ Build

Fix a missing include that fails builds with Apple's `clang` and Intel's `dpcpp` compilers.

Changes to "0.13.3"

Bug Fixes

- `Variant.hpp`: `size_t` include #972

3.2.11 0.13.3

Date: 2021-04-09

Fix Various Read Issues

This release fixes various bugs related to reading: a chunk fallback for constant components, skip missing patch records, a backend bug in each ADIOS2 & HDF5, and we made the Python `load_chunk` method more robust.

Changes to "0.13.2"

Bug Fixes

- `available_chunks()` for constant components #942
- Particle Patches: Do not emplace patch records if they don't exist in the file being read #945
- ADIOS2: decay `ReadWrite` mode into `adios2::Mode::Read` if the file exists #943
- HDF5: fix segfault with libSplash files #962
- Python: fix `load_chunk` to temporary #913

Other

- Sphinx: limit docutils to 0.16
- CI: remove a failing `find` command

3.2.12 0.13.2

Date: 2021-02-02

Fix Patch Read & Python store_chunk

This release fixes a regression with particle patches, related to `Iteration::open()` and `::close()` functionality. Also, issues with the Python `store_chunk` method are addressed.

Changes to "0.13.1"

Bug Fixes

- Read: check whether particle patches are dirty & handle gracefully #909
- Python `store_chunk`:
  - add support for complex types #915
  - fix a use-after-free with temporary variables #912

Other

- CMake: hint `CMAKE_PREFIX_PATH` as a warning for HDF5 #896

3.2.13 0.13.1

Date: 2021-01-08

Fix openPMD-ls & Iteration open/close

This release fixes regressions in the series "ls" functionality and tools, related to `Iteration::open()` and `::close()` functionality. We also add support to read back complex numbers with JSON.

Changes to "0.13.0"

Bug Fixes

- fix `Iteration::close()` and `helper::listSeries`` / `list_series` / `openPMD-ls` #878 #880 #882 #883 #884
- `setup.py`: stay with `Python_EXECUTABLE` #875
- `FindPython.cmake`: Avoid overspecifying `Development.Module` with CMake 3.18+ #868
- `ChunkInfo`:
  - fix includes #879
  - tests: adapt `sourceID` to handle nondeterministic subfile order #871
- ADIOS1: fix `Iteration::open()` #864
- JSON: support complex datatype reads #885
- Docs: fix formatting of first read/write #892

Other

- bounds check: more readable error message #890
- ADIOS2: add a missing space in an error message #881
- Docs: released pypi wheels include windows #869
- CI:
  - LGTM: fix C++ #873
  - Brew returns non-zero if already installed #877

3.2.14 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

3.2.15 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

3.2.16 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

3.2.17 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 handing `.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

3.2.18 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

3.2.19 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

3.2.20 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

3.2.21 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

3.2.22 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

3.2.23 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

3.2.24 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

3.2.25 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

3.2.26 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

3.2.27 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

3.2.28 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

3.2.29 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

3.2.30 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

3.2.31 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

3.2.32 0.3.1-alpha

Date: 2018-07-07

Refined fileBased Series & Python Data Load

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

Changes to "0.3.0-alpha"

Features

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

Bug Fixes

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

Other

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

3.2.33 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

3.2.34 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

3.2.35 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

3.2.36 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.

3.3 Upgrade Guide

3.3.1 0.16.0

The ADIOS1 library is no longer developed in favor of ADIOS2. Consequently, ADIOS1 support was removed in openPMD-api 0.16.0 and newer. Please transition to ADIOS2.

For reading legacy ADIOS1 BP3 files, either use an older version of openPMD-api or the BP3 backend in ADIOS2. Note that ADIOS2 does not support compression in BP3 files.

pybind11 2.11.1 is now the minimally supported version for Python support.

3.3.2 0.15.0

Building openPMD-api now requires a compiler that supports C++17 or newer. `MPark.Variant` is not a dependency anymore (kudos and thanks for the great time!).

Python 3.10 & 3.11 are now supported, Python 3.6 is removed. openPMD-api now depends on toml11 3.7.1+. pybind11 2.10.1 is now the minimally supported version for Python support. Catch2 2.13.10 is now the minimally supported version for tests.

The following backend-specific members of the `Dataset` class have been removed: `Dataset::setChunkSize()`, `Dataset::setCompression()`, `Dataset::setCustomTransform()`, `Dataset::chunkSize`, `Dataset::compression`, `Dataset::transform`. They are replaced by backend-specific options in the JSON-based backend configuration. This can be passed in `Dataset::options`. The following configuration shows a compression configuration for ADIOS1 and ADIOS2:

```
{
  "adios1": {
    "dataset": {
      "transform": "blosc:compressor=zlib,shuffle=bit,lvl=1;nometa"
    }
  },
  "adios2": {
    "dataset": {
      "operators": [
        {
          "type": "zlib",
          "parameters": {
            "clevel": 9
          }
        }
      ]
    }
  }
}
```

Or alternatively, in TOML:

```
[adios1.dataset]
transform = "blosc:compressor=zlib,shuffle=bit,lvl=1;nometa"

[[adios2.dataset.operators]]
type = "zlib"
parameters.clevel = 9
```

The helper function `shareRaw` of the C++ API has been deprecated. In its stead, there are now new API calls `RecordComponent::storeChunkRaw()` and `RecordComponent::loadChunkRaw`.

The **ADIOS1 backend** is now deprecated, to be replaced fully with ADIOS2. Now is a good time to check if ADIOS2 is able to read old ADIOS1 datasets that you might have. Otherwise, `openpmd-pipe` can be used for conversion:

```
openpmd-pipe --infile adios1_dataset_%T.bp --inconfig 'backend = "adios1"' --outfile␣
↪adios2_dataset_%T.bp --outconfig 'backend = "adios2"'
```

The class structure of `Container` and deriving classes has been reworked. Usage of the API generally stays the same, but code that relies on the concrete class structure might break.

The `Iteration::closedByWriter()` attribute has been deprecated as a leftover from the early streaming implementation.

Old:

```
double const * data;
recordComponent.storeChunk(shareRaw(data), offset, extent);
```

New:

```
double const * data;
recordComponent.storeChunkRaw(data, offset, extent);
```

Additionally, `determineDatatype` now accepts pointer types (raw and smart pointers):

Old:

```
std::vector<double> data;
Datatype dt = determineDatatype(shareRaw(data));
```

New:

```
std::vector<double> data;
Datatype dt = determineDatatype(data.data());
```

Note: `determineDatatype` does not directly accept `determineDatatype(data)`, since it's unclear if the result from that call would be `Datatype::DOUBLE` or `Datatype::VEC_DOUBLE`.

In order to get the direct mapping between C++ type and openPMD datatype, use the template parameter of `determineDatatype`: `determineDatatype<decltype(data)>()` or `determineDatatype<std::vector<double>>()`.

3.3.3 0.14.0

ADIOS 2.7.0 is now the minimally supported version for ADIOS2 support. Catch2 2.13.4 is now the minimally supported version for tests. pybind11 2.6.2 is now the minimally supported version for Python support.

In `RecordComponent::loadChunk`, the optional last argument `targetUnitSI` was removed as it has not been implemented yet and had thus no function.

3.3.4 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:

```python
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.

3.3.5 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.

3.3.6 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/`.

3.3.7 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.

3.3.8 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.

3.3.9 0.7.0-alpha

Python

Module Name

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

```
# old name
import openPMD
```

(continues on next page)

(continued from previous page)

```
# 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);
```

(continues on next page)

(continued from previous page)

```
// 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();
```

CHAPTER

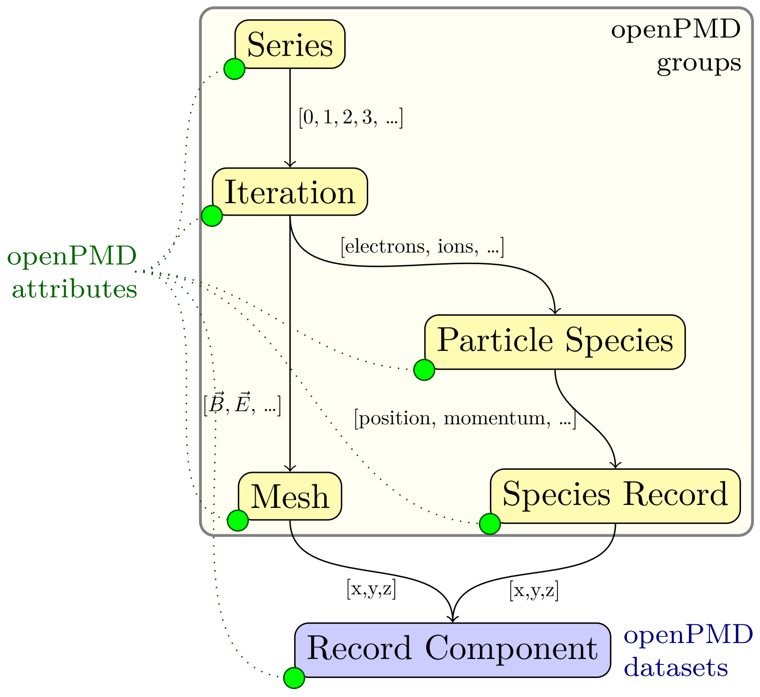
FOUR

USAGE

4.1 Concepts

For a first high-level overview on openPMD, please see www.openPMD.org.

openPMD defines data series of particles and mesh based data in the openPMD-standard. See for example the openPMD base standard definition, version 1.1.0.



4.1.1 Records and Record Components

At the bottom of openPMD is an *Record* that stores data in *record components*. A record is an data set with common properties, e.g. the electric field $\vec{E}$ with three components E_x, E_y, E_z can be a record. A density field could be another record - which is scalar as it only has one component.

In general, openPMD allows records with arbitrary number of components (tensors), as well as vector records and scalar records. In the case of vector records, the single components are stored as datasets within the record. In the case of scalar records, the record and component are equivalent. In the API, the record can be directly used as a component, and in the standard a scalar record is represented by the scalar dataset with attributes.

4.1.2 Meshes and Particles

Records can be either (structured) meshes, e.g. a gridded electric field as mentioned above, or particle records.

Mesh records are logically n-dimensional arrays. The openPMD standard (see above) supports a various mesh geometries, while more can be standardized in the future. Ongoing work also adds support for block-structured mesh-refinement.

Particle species on the other hand group a number of particle records that are themselves stored as (logical) 1d arrays in record components. Conceptually, one could also think as particles *table* or *dataframe*, where each row represents a particle.

4.1.3 Iteration and Series

Updates to records are stored in an *Iteration*. Iterations are numbered by integers, do not need to be consecutive and can for example be used to store a the evolution of records over time.

The collection of iterations is called a *Series*. openPMD-api implements various file-formats (backends) and encoding strategies for openPMD Series, from simple one-file-per-iteration writes over using the backend-provided support for internal updates of records to data streaming techniques.

Iteration encoding: The openPMD-api can encode iterations in different ways. The method `Series::setIterationEncoding()` (C++) or `Series.set_iteration_encoding()` (Python) may be used in writing for selecting one of the following encodings explicitly:

- **group-based iteration encoding:** This encoding is the default. It creates a separate group in the hierarchy of the openPMD standard for each iteration. As an example, all data pertaining to iteration 0 may be found in group `/data/0`, for iteration 100 in `/data/100`.
- **file-based iteration encoding:** A unique file on the filesystem is created for each iteration. The preferred way to create a file-based iteration encoding is by specifying an expansion pattern in the `filepath` argument of the constructor of the `Series` class. Creating a `Series` by the filepath `"series_%T.json"` will create files `series_0.json`, `series_100.json` and `series_200.json` for iterations 0, 100 and 200. A padding may be specified by `"series_%06T.json"` to create files `series_000000.json`, `series_000100.json` and `series_000200.json`. The inner group layout of each file is identical to that of the group-based encoding.
- **variable-based iteration encoding:** This experimental encoding uses a feature of some backends (i.e., ADIOS2) to maintain datasets and attributes in several versions (i.e., iterations are stored inside *variables*). No iteration-specific groups are created and the corresponding layer is dropped from the openPMD hierarchy. In backends that do not support this feature, a series created with this encoding can only contain one iteration.

Spellings for constants in the C++ (`IterationEncoding`) and Python (`Iteration_Encoding`) API: `groupBased`, `variableBased`, `fileBased`, `group_based`, `group_based`, `variable_based`

4.1.4 Attributes

openPMD defines a minimal set of standardized meta-data *Attributes* to for scientific self-description and portability. Such attributes are showcased in the following section and include for example the physical quantities in a record, unit conversions, time and gridding information.

Besides the standardized attributes, arbitrary additional attributes can be added to openPMD data and openPMD-api supports adding use-defined attributes on every object of the herein described hierarchy.

Tip: Does all of this sound a bit too theoretical? Just jump to the next section and see an example in action.

4.2 First Write

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

4.2.1 Include / Import

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

C++17

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

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

namespace io = openPMD;
```

Python

```python
import openpmd_api as io

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

4.2.2 Open

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

C++17

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

Python

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

4.2.3 Iteration

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

C++17

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

Python

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

4.2.4 Attributes

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

C++17

```
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

```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)
```

4.2.5 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 $(B_x, B_y, B_z)^\intercal$ of which the B_y component shall be constant for all (i, j) indices.

C++17

```cpp
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

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



y_data = 4.

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

4.2.6 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++17

```cpp
// 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

```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)
```

4.2.7 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 [mass (M)^{1} · electric current (I)$^{-1}$ · 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++17

```
// 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.

4.2.8 Register Chunk

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

C++17

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

B_y.makeConstant(y_data);
```

Python

```
B_x[:, :] = x_data


B_z[:, :] = 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!

4.2.9 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++17

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

Python

```
series.flush()
```

4.2.10 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++17

```
series.close()
```

Python

```
series.close()
```

4.3 First Read

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

4.3.1 Include / Import

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

C++17

```
#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
```

4.3.2 Open

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

C++17

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

Python

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

Tip: Replace the file ending `.h5` with a wildcard `.%E` to let openPMD autodetect the ending from the file system. Use the wildcard `%T` to match filename encoded iterations.

Tip: More detailed options can be passed via JSON or TOML as a further constructor parameter. Try `{"defer_iteration_parsing": true}` to speed up the first access. (Remember to explicitly `it.open()` iterations in that case.)

4.3.3 Iteration

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

C++17

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

Python

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

4.3.4 Attributes

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

C++17

```
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)
```

4.3.5 Record

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

C++17

```
// 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.

4.3.6 Units

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

C++17

```
// 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.

4.3.7 Register Chunk

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

C++17

```
// 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++17

```
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()`.

4.3.8 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++17

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

Python

```
series.flush()
```

4.3.9 Data

We can now work with the newly loaded data in `x_data` (or `x_slice_data`):

C++17

```
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])
```

4.3.10 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++17

```
series.close()
```

Python

```
series.close()
```

4.4 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.

4.4.1 Reading

C++

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

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

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:";
```

(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"].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};
    // Loading without explicit datatype here
    auto chunk_data = E_x.loadChunkVariant(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";
    std::visit(
        [&chunk_offset, &chunk_extent](auto &shared_ptr) {
            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"
                         << shared_ptr.get()[row * chunk_extent[1] + col];
                cout << '\n';
            }
        },
        chunk_data);

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

    // The iteration can be closed in order to help free up resources.
    // The iteration's content will be flushed automatically.
    // An iteration once closed cannot (yet) be reopened.
    i.close();
    cout << "Full E/x starts with:\n\t{";
    for (size_t col = 0; col < extent[1] && col < 5; ++col)
        cout << all_data.get()[col] << ", ";
    cout << "...}\n";
```

(continues on next page)

(continued from previous page)

```
    /* The files in 'series' are still open until the object is destroyed, on
     * which it cleanly flushes and closes all open file handles.
     * When running out of scope on return, the 'Series' destructor is called.
     * Alternatively, one can call `series.close()` to the same effect as
     * calling the destructor, including the release of file handles.
     */
    series.close();
    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"]
    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))
```

(continues on next page)

(continued from previous page)

```
    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()

    # The iteration can be closed in order to help free up resources.
    # The iteration's content will be flushed automatically.
    # An iteration once closed cannot (yet) be reopened.
    i.close()
    print("Full E/x is of shape {0} and starts with:".format(all_data.shape))
    print(all_data[0, 0, :5])

    # The files in 'series' are still open until the series is closed, at which
    # time it cleanly flushes and closes all open file handles.
    # One can close the object explicitly to trigger this.
    # Alternatively, this will automatically happen once the garbage collector
    # claims (every copy of) the series object.
    series.close()
```

4.4.2 Writing

C++

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

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

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
```

(continues on next page)

(continued from previous page)

```
          << ") 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";

    // `Series::writeIterations()` and `Series::readIterations()` are
    // intentionally restricted APIs that ensure a workflow which also works
    // in streaming setups, e.g. an iteration cannot be opened again once
    // it has been closed.
    // `Series::iterations` can be directly accessed in random-access workflows.
    Mesh rho = series.writeIterations()[1].meshes["rho"];
    cout << "Created a scalar mesh Record with all required openPMD "
            "attributes\n";

    Datatype datatype = determineDatatype(global_data.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";

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

    // The iteration can be closed in order to help free up resources.
    // The iteration's content will be flushed automatically.
    // An iteration once closed cannot (yet) be reopened.
    series.writeIterations()[1].close();

    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.
     * Alternatively, one can call `series.close()` to the same effect as
     * calling the destructor, including the release of file handles.
     */
    series.close();
    return 0;
}
```

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

Python

```python
import numpy as np
import openpmd_api as io

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))
    # `Series.write_iterations()` and `Series.read_iterations()` are
    # intentionally restricted APIs that ensure a workflow which also works
    # in streaming setups, e.g. an iteration cannot be opened again once
    # it has been closed.
    # `Series.iterations` can be directly accessed in random-access workflows.
    rho = series.write_iterations()[1]. \
        meshes["rho"]

    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")

    # The iteration can be closed in order to help free up resources.
    # The iteration's content will be flushed automatically.
    # An iteration once closed cannot (yet) be reopened.
    series.write_iterations()[1].close()
    print("Dataset content has been fully written")

    # The files in 'series' are still open until the series is closed, at which
    # time it cleanly flushes and closes all open file handles.
    # One can close the object explicitly to trigger this.
```

(continues on next page)

(continued from previous page)

```
    # Alternatively, this will automatically happen once the garbage collector
    # claims (every copy of) the series object.
    series.close()
```

4.5 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.

4.5.1 Reading

C++

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

#include <mpi.h>

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

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);

    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";

    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};

    // If you know the datatype, use `loadChunk<double>(...)` instead.
    auto chunk_data = E_x.loadChunkVariant(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";
```

(continues on next page)

(continued from previous page)

```
    // The iteration can be closed in order to help free up resources.
    // The iteration's content will be flushed automatically.
    // An iteration once closed cannot (yet) be reopened.
    series.iterations[100].close();

    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";
                    /*
                     * For hot loops, the std::visit(...) call should be moved
                     * further up.
                     */
                    std::visit(
                        [row, col, &chunk_extent](auto &shared_ptr) {
                            cout << shared_ptr
                                        .get()[row * chunk_extent[1] + col];
                        },
                        chunk_data);
                }
                cout << std::endl;
            }
        }

        // this barrier is not necessary but structures the example output
        MPI_Barrier(MPI_COMM_WORLD);
    }
    // The files in 'series' are still open until the series is closed, at which
    // time it cleanly flushes and closes all open file handles.
    // One can close the object explicitly to trigger this.
    // Alternatively, this will automatically happen once the garbage collector
    // claims (every copy of) the series object.
    // In any case, this must happen before MPI_Finalize() is called
    series.close();

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

    return 0;
}
```

Python

```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
    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")

    # The iteration can be closed in order to help free up resources.
    # The iteration's content will be flushed automatically.
    # An iteration once closed cannot (yet) be reopened.
    series.iterations[100].close()

    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 series is closed, at which
```

(continues on next page)

(continued from previous page)

```
    # time it cleanly flushes and closes all open file handles.
    # One can close the object explicitly to trigger this.
    # Alternatively, this will automatically happen once the garbage collector
    # claims (every copy of) the series object.
    # In any case, this must happen before MPI_Finalize() is called
    # (usually in the mpi4py exit hook).
    series.close()
```

4.5.2 Writing

C++

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

#include <mpi.h>

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

using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    int mpi_size;
    int mpi_rank;

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

    // 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";

    std::string subfiling_config = R"(
[hdf5.vfd]
type = "subfiling"
ioc_selection = "every_nth_rank"
stripe_size = 33554432
stripe_count = -1

[hdf5.dataset]
chunks = "auto"
        )";

    // open file for writing
```

(continues on next page)

(continued from previous page)

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

    // In parallel contexts, it's important to explicitly open iterations.
    // You can either explicitly access Series::iterations and use
    // Iteration::open() afterwards, or use `Series::writeIterations()`,
    // or in read mode `Series::readIterations()` where iterations are opened
    // automatically.
    // `Series::writeIterations()` and `Series::readIterations()` are
    // intentionally restricted APIs that ensure a workflow which also works
    // in streaming setups, e.g. an iteration cannot be opened again once
    // it has been closed.
    series.iterations[1].open();
    Mesh mymesh = series.iterations[1].meshes["mymesh"];

    // example 1D domain decomposition in first index
    Datatype datatype = determineDatatype<float>();
    Extent global_extent = {10ul * mpi_size, 300};
    Dataset dataset = Dataset(datatype, global_extent, R"(
[hdf5.dataset]
chunks = [10, 100]
    )");

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

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

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

    // The iteration can be closed in order to help free up resources.
    // The iteration's content will be flushed automatically.
    // An iteration once closed cannot (yet) be reopened.
    series.iterations[100].close();

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

    /* The files in 'series' are still open until the object is destroyed, on
```

(continues on next page)

(continued from previous page)

```
     * which it cleanly flushes and closes all open file handles.
     * When running out of scope on return, the 'Series' destructor is called.
     * Alternatively, one can call `series.close()` to the same effect as
     * calling the destructor, including the release of file handles.
     */
    series.close();

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

    return 0;
}
```

Python

```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 numpy as np
import openpmd_api as io

try:
    import adios2
    from packaging import version
    USE_JOINED_DIMENSION = \
        version.parse(adios2.__version__) >= version.parse('2.9.0')
except ImportError:
    USE_JOINED_DIMENSION = False

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.rank
    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.bp"
        if USE_JOINED_DIMENSION
        else "../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))
```

(continues on next page)

(continued from previous page)

```
    # In parallel contexts, it's important to explicitly open iterations.
    # This is done automatically when using `Series.write_iterations()`,
    # or in read mode `Series.read_iterations()`.
    #
    # `Series.write_iterations()` and `Series.read_iterations()` are
    # intentionally restricted APIs that ensure a workflow which also works
    # in streaming setups, e.g. an iteration cannot be opened again once
    # it has been closed.
    # `Series.iterations` can be directly accessed in random-access workflows.
    series.iterations[1].open()
    mymesh = series.iterations[1]. \
        meshes["mymesh"]

    # example 1D domain decomposition in first index
    global_extent = [io.Dataset.JOINED_DIMENSION, 300] \
        if USE_JOINED_DIMENSION else [comm.size * 10, 300]

    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

    if USE_JOINED_DIMENSION:
        # explicit API
        # mymesh.store_chunk(local_data, [], [10, 300])
        mymesh[:, :] = local_data
        # or short:
        # mymesh[()] = local_data
    else:
        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")

    # The iteration can be closed in order to help free up resources.
    # The iteration's content will be flushed automatically.
    # An iteration once closed cannot (yet) be reopened.
    series.iterations[1].close()

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

    # The files in 'series' are still open until the series is closed, at which
    # time it cleanly flushes and closes all open file handles.
    # One can close the object explicitly to trigger this.
    # Alternatively, this will automatically happen once the garbage collector
    # claims (every copy of) the series object.
```

(continues on next page)

(continued from previous page)

```
    series.close()
```

4.6 Workflow

4.6.1 Storing and reading chunks

1. **Chunks within an n-dimensional dataset**

   Most commonly, chunks within an n-dimensional dataset are identified by their offset and extent. The extent is the size of the chunk in each dimension, NOT the absolute coordinate within the entire dataset.

   In the Python API, this is modeled to conform to the conventional `__setitem__`/`__getitem__` protocol.

2. **Joined arrays (write only)**

   (Currently) only supported in ADIOS2 no older than v2.9.0 under the conditions listed in the ADIOS2 documentation on joined arrays.

   In some cases, the concrete chunk within a dataset does not matter and the computation of indexes is a needless computational and mental overhead. This commonly occurs for particle data which the openPMD-standard models as a list of particles. The order of particles does not matter greatly, and making different parallel processes agree on indexing is error-prone boilerplate.

   In such a case, at most one *joined dimension* can be specified in the Dataset, e.g. `{Dataset::JOINED_DIMENSION, 128, 128}` (3D for the sake of explanation, particle data would normally be 1D). The chunk is then stored by specifying an empty offset vector `{}`. The chunk extent vector must be equivalent to the global extent in all non-joined dimensions (i.e. joined arrays allow no further sub-chunking other than concatenation along the joined dimension). The joined dimension of the extent vector specifies the extent that this piece should have along the joined dimension. In the Python API, the slice-based setter syntax can be used as an abbreviation since the necessary information is determined from the passed array, e.g. `record_component[()] = local_data`. The global extent of the dataset along the joined dimension will then be the sum of all local chunk extents along the joined dimension.

   Since openPMD follows a struct-of-array layout of data, it is important not to lose correlation of data between components. E.g., joining an array must take care that `particles/e/position/x` and `particles/e/position/y` are joined in uniform way.

   The openPMD-api makes the **following guarantee**:

   Consider a Series written from `N` parallel processes between two (collective) flush points. For each parallel process `n` and dataset `D`, let:

   - `chunk(D, n, i)` be the `i`'th chunk written to dataset `D` on process `n`
   - `num_chunks(D, n)` be the count of chunks written by `n` to `D`
   - `joined_index(D, c)` be the index of chunk `c` in the joining order of `D`

   Then for any two datasets `x` and `y`:

   - If for any parallel process `n` the condition holds that `num_chunks(x, n) = num_chunks(y, n)` (between the two flush points!)…
   - …then for any parallel process `n` and chunk index `i` less than `num_chunks(x, n)`: `joined_index(x, chunk(x, n, i)) = joined_index(y, chunk(y, n, i))`.

   TLDR: Writing chunks to two joined arrays in synchronous way (**1.** same order of store operations and **2.** between the same flush operations) will result in the same joining order in both arrays.

4.6.2 Access modes

The openPMD-api distinguishes between a number of different access modes:

- **Create mode**: Used for creating a new Series from scratch. Any file possibly existing in the specified location will be overwritten.
- Two distinct read modes: **Read-random-access mode** and **Read-linear mode**. (Specification of **Read-only mode** is equivalent to read-random-access mode.) Both modes are used for reading from an existing Series. No modifications will be made.

  Differences between both modes:

  - When intending to use `Series::readIterations()` (i.e. step-by-step reading of iterations, e.g. in streaming), then **linear read mode** is preferred and always supported. Data is parsed inside `Series::readIterations()`, no data is available right after opening the Series. Global attributes are available directly after calling `Series::readIterations()`, Iterations and all their corresponding data become available by use of the returned Iterator, e.g. in a foreach loop.
  - Otherwise (i.e. for random-access workflows), **random-access read mode** is required, but works only in backends that support random access. Data is parsed and available right after opening the Series.

  In both modes, parsing of iterations can be deferred with the JSON/TOML option `defer_iteration_parsing`.

  Detailed rules:

  1. In backends that have no notion of IO steps (all except ADIOS2), *random-access read mode* can always be used.
  2. In backends that can be accessed either in random-access or step-by-step, the chosen access mode decides which approach is used. Examples are the BP4 and BP5 engines of ADIOS2.
  3. In streaming backends, random-access is not possible. When using such a backend, the access mode will be coerced automatically to *linear read mode*. Use of Series::readIterations() is mandatory for access.
  4. Reading a variable-based Series is only fully supported with *linear access mode*. If using *random-access read mode*, the dataset will be considered to only have one single step. If the dataset only has one single step, this is guaranteed to work as expected. Otherwise, it is undefined which step's data is returned.

- **Read/Write mode**: Creates a new Series if not existing, otherwise opens an existing Series for reading and writing. New datasets and iterations will be inserted as needed. Not fully supported by all backends:

  - ADIOS2: Automatically coerced to *Create* mode if the file does not exist yet and to *Read-only* mode if it exists.

  Since this happens on a per-file level, this mode allows to read from existing iterations and write to new iterations at the same time in file-based iteration encoding.

- **Append mode**: Restricted mode for appending new iterations to an existing Series that is supported by all backends in all encodings. The API is equivalent to that of the *Create* mode, meaning that no reading is supported whatsoever. If the Series does not exist yet, this behaves equivalently to the *Create* mode. Existing iterations will not be deleted, newly-written iterations will be inserted.

  Warning: When writing an iteration that already exists, the behavior is implementation-defined and depends on the chosen backend and iteration encoding:

  - The new iteration might fully replace the old one.
  - The new iteration might be merged into the old one.
  - (To be removed in a future update) The old and new iteration might coexist in the resulting dataset.

  We suggest to fully define iterations when using Append mode (i.e. as if using Create mode) to avoid implementation-specific behavior. Appending to an openPMD Series is only supported on a per-iteration level.

Warning: There is no reading involved in using Append mode. It is a user's responsibility to ensure that the appended dataset and the appended-to dataset are compatible with each other. The results of using incompatible backend configurations are undefined.

4.6.3 Deferred Data API Contract

IO operations are in general not performed by the openPMD API immediately after calling the corresponding API function. Rather, operations are enqueued internally and performed at so-called *flush points*. A flush point is a point within an application's sequential control flow where the openPMD API must uphold the following guarantees:

- In write mode, any change made to a user buffer whose data shall be stored in a dataset up to the flush point must be found in the written dataset.
- In write mode, no change made to a user buffer whose data shall be stored in a dataset after the flush point must be found in the written dataset.
- In read mode, a buffer into which data from a dataset should be filled, must not be altered by the openPMD API before the flush point.
- In read mode, a buffer into which data from a dataset should be filled, must have been filled with the requested data after the flush point.

In short: operations requested by `storeChunk()` and `loadChunk()` must happen exactly at flush points.

Flush points are triggered by:

- Calling `Series::flush()`.
- Calling `Iteration::close( flush=true )`. Flush point guarantees affect only the corresponding iteration.
- Calling `Writable::seriesFlush()` or `Attributable::seriesFlush()`.
- The streaming API (i.e. `Series.readIterations()` and `Series.writeIteration()`) automatically before accessing the next iteration.

Attributes are (currently) unaffected by this:

- In writing, attributes are stored internally by value and can afterwards not be accessed by the user.
- In reading, attributes are parsed upon opening the Series / an iteration and are available to read right-away.

Attention: Note that the concrete implementation of `Series::flush()` and `Attributable::seriesFlush()` is backend-specific. Using these calls does neither guarantee that data is moved to storage/transport nor that it can be accessed by independent readers at this point.

Some backends (e.g. the BP5 engine of ADIOS2) have multiple implementations for the openPMD-api-level guarantees of flush points. For user-guided selection of such implementations, `Series::flush` and `Attributable::seriesFlush()` take an optional JSON/TOML string as a parameter. See the section on *backend-specific configuration* for details.

4.6.4 Deferred Data API Contract

A verbose debug log can optionally be printed to the standard error output by specifying the environment variable `OPENPMD_VERBOSE=1`. Note that this functionality is at the current time still relatively basic.

4.7 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).

4.7.1 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. Use of `Access::READ_LINEAR` mode is recommended. 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.

Iterations are implicitly opened by the Streaming API and `Iteration::open()` needs not be called explicitly. 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.

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

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

using std::cout;
using namespace openPMD;

int main()
{
#if openPMD_HAVE_ADIOS2
    auto backends = openPMD::getFileExtensions();
    if (std::find(backends.begin(), backends.end(), "sst") == backends.end())
    {
        std::cout << "SST engine not available in ADIOS2." << std::endl;
        return 0;
    }

    Series series = Series("electrons.sst", Access::READ_LINEAR, R"(
{
```

(continues on next page)

(continued from previous page)

```
  "adios2": {
    "engine": {
      "parameters": {
        "DataTransport": "WAN"
      }
    }
  }
})");

    // `Series::writeIterations()` and `Series::readIterations()` are
    // intentionally restricted APIs that ensure a workflow which also works
    // in streaming setups, e.g. an iteration cannot be opened again once
    // it has been closed.
    // `Series::iterations` can be directly accessed in random-access workflows.
    for (IndexedIteration iteration : series.readIterations())
    {
        std::cout << "Current iteration: " << iteration.iterationIndex
                  << std::endl;
        Record electronPositions = iteration.particles["e"]["position"];
        std::array<RecordComponent::shared_ptr_dataset_types, 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 const &dim = dimensions[i];
            RecordComponent rc = electronPositions[dim];
            loadedChunks[i] = rc.loadChunkVariant(
                Offset(rc.getDimensionality(), 0), rc.getExtent());
            extents[i] = rc.getExtent();
        }

        // The iteration can be closed in order to help free up resources.
        // The iteration's content will be flushed automatically.
        // An iteration once closed cannot (yet) be reopened.
        iteration.close();

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

    /* The files in 'series' are still open until the object is destroyed, on
```

(continues on next page)

(continued from previous page)

```
     * which it cleanly flushes and closes all open file handles.
     * When running out of scope on return, the 'Series' destructor is called.
     * Alternatively, one can call `series.close()` to the same effect as
     * calling the destructor, including the release of file handles.
     */
    series.close();

    return 0;
#else
    std::cout << "The streaming example requires that openPMD has been built "
                 "with ADIOS2."
              << std::endl;
    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. Use of `Access.read_linear` mode is recommended. 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.

Iterations are implicitly opened by the Streaming API and `Iteration.open()` needs not be called explicitly. 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 json
import sys

import openpmd_api as io

# pass-through for ADIOS2 engine parameters
# https://adios2.readthedocs.io/en/latest/engines/engines.html
config = {'adios2': {'engine': {}, 'dataset': {}}}
config['adios2']['engine'] = {'parameters':
                              {'Threads': '4', 'DataTransport': 'WAN'}}
config['adios2']['dataset'] = {'operators': [{'type': 'bzip2'}]}

if __name__ == "__main__":
    # this block is for our CI, SST engine is not present on all systems
    backends = io.file_extensions
    if "sst" not in backends:
        print("SST engine not available in ADIOS2.")
        sys.exit(0)

    series = io.Series("simData.sst", io.Access_Type.read_linear,
                       json.dumps(config))

    # Read all available iterations and print electron position data.
```

(continues on next page)

(continued from previous page)

```
    # Direct access to iterations is possible via `series.iterations`.
    # For streaming support, `series.read_iterations()` needs to be used
    # instead of `series.iterations`.
    # `Series.write_iterations()` and `Series.read_iterations()` are
    # intentionally restricted APIs that ensure a workflow which also works
    # in streaming setups, e.g. an iteration cannot be opened again once
    # it has been closed.
    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)

        # Closing the iteration loads all data and releases the current
        # streaming step.
        # If the iteration is not closed, it will be implicitly closed upon
        # opening the next iteration.
        iteration.close()

        # data is now available for printing
        for i in range(3):
            dim = dimensions[i]
            shape = shapes[i]
            print("dim: {}".format(dim))
            chunk = loadedChunks[i]
            print(chunk)

    # 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.
    # Alternatively, one can call `series.close()` to the same effect as
    # calling the destructor, including the release of file handles.
    series.close()
```

4.7.2 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.

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

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

using std::cout;
using namespace openPMD;

int main()
{
#if openPMD_HAVE_ADIOS2
    using position_t = double;
    auto backends = openPMD::getFileExtensions();
    if (std::find(backends.begin(), backends.end(), "sst") == backends.end())
    {
        std::cout << "SST engine not available in ADIOS2." << std::endl;
        return 0;
    }

    // open file for writing
    Series series = Series("electrons.sst", Access::CREATE, R"(
{
  "adios2": {
    "engine": {
      "parameters": {
        "DataTransport": "WAN"
      }
    }
  }
})");

    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; });

    // `Series::writeIterations()` and `Series::readIterations()` are
    // intentionally restricted APIs that ensure a workflow which also works
    // in streaming setups, e.g. an iteration cannot be opened again once
```

(continues on next page)

(continued from previous page)

```
    // it has been closed.
    // `Series::iterations` can be directly accessed in random-access workflows.
    WriteIterations iterations = series.writeIterations();
    for (size_t i = 0; i < 100; ++i)
    {
        Iteration iteration = iterations[i];
        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();
    }

    /* 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.
     * Alternatively, one can call `series.close()` to the same effect as
     * calling the destructor, including the release of file handles.
     */
    series.close();

    return 0;
#else
    std::cout << "The streaming example requires that openPMD has been built "
                 "with ADIOS2."
              << std::endl;
    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 json
import sys

import numpy as np
import openpmd_api as io
```

(continues on next page)

(continued from previous page)

```python
# pass-through for ADIOS2 engine parameters
# https://adios2.readthedocs.io/en/latest/engines/engines.html
config = {'adios2': {'engine': {}, 'dataset': {}}}
config['adios2']['engine'] = {'parameters':
                              {'Threads': '4', 'DataTransport': 'WAN'}}
config['adios2']['dataset'] = {'operators': [{'type': 'bzip2'}]}

if __name__ == "__main__":
    # this block is for our CI, SST engine is not present on all systems
    backends = io.file_extensions
    if "sst" not in backends:
        print("SST engine not available in ADIOS2.")
        sys.exit(0)

    # create a series and specify some global metadata
    # change the file extension to .json, .h5 or .bp for regular file writing
    series = io.Series("simData.sst", io.Access_Type.create,
                       json.dumps(config))
    series.set_author("Franz Poeschel <f.poeschel@hzdr.de>")
    series.set_software("openPMD-api-python-examples")

    # now, write a number of iterations (or: snapshots, time steps)
    for i in range(10):
        # Direct access to iterations is possible via `series.iterations`.
        # For streaming support, `series.write_iterations()` needs to be used
        # instead of `series.iterations`.
        # `Series.write_iterations()` and `Series.read_iterations()` are
        # intentionally restricted APIs that ensure a workflow which also works
        # in streaming setups, e.g. an iteration cannot be opened again once
        # it has been closed.
        iteration = series.write_iterations()[i]

        #######################
        # write electron data #
        #######################

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

        # openPMD attribute
        # (this one would also be set automatically for positions)
        electronPositions.unit_dimension = {io.Unit_Dimension.L: 1.0}
        # custom attribute
        electronPositions.set_attribute("comment", "I'm a comment")

        length = 10
        local_data = np.arange(i * length, (i + 1) * length,
                               dtype=np.dtype("double"))
        for dim in ["x", "y", "z"]:
            pos = electronPositions[dim]
            pos.reset_dataset(io.Dataset(local_data.dtype, [length]))
            pos[()] = local_data

        # optionally: flush now to clear buffers
        iteration.series_flush()  # this is a shortcut for `series.flush()`
```

(continues on next page)

(continued from previous page)

```
        ################################
        # write some temperature data #
        ################################

        temperature = iteration.meshes["temperature"]
        temperature.unit_dimension = {io.Unit_Dimension.theta: 1.0}
        temperature.axis_labels = ["x", "y"]
        temperature.grid_spacing = [1., 1.]
        # temperature has no x,y,z components, so skip the last layer:
        temperature_dataset = temperature
        # let's say we are in a 3x3 mesh
        temperature_dataset.reset_dataset(
            io.Dataset(np.dtype("double"), [3, 3]))
        # temperature is constant
        temperature_dataset.make_constant(273.15)

        # After closing the iteration, the readers can see the iteration.
        # It can no longer be modified.
        # If not closing an iteration explicitly, it will be implicitly closed
        # upon creating the next iteration.
        iteration.close()

    # 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.
    # Alternatively, one can call `series.close()` to the same effect as
    # calling the destructor, including the release of file handles.
    series.close()
```

4.8 Benchmarks

4.8.1 Parallel benchmark 8

Build based on the helper functions in the *benchmark utilities*, this benchmark executes a simple parallel read-write test.

In particular, this test case writes and reads a 3D array of type `uint64_t`, sliced 1D along the first dimension.

```
openPMD::Extent total{
    100 * scale_up, // sliced along this axis over MPI ranks
    100,
    1000
};
```

The benchmark writes 10 iterations, meaning that in the strong-scaling case, always around 3/4 GB of data are produced. In the weak-scaling case, the data scales as $N * 3/4\text{GiB}$ with N as the number of participating MPI ranks.

By default, the benchmarks executes as strong-scaling unless the `-w`/`--weak` option is passed as a command-line argument to the executable.

4.8.2 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_*

Optional input parameter: pack

Often a processor will write out a few small blocks. Using the example above, if the writer side uses 1024 processors, each processor will handle 16 blocks from the 32x32x16 grid. These 16 blocks per processor can be packed from a selection of 1x1x16 blocks, or 2x2x4, etc. The `pack` parameter specifies this selection, e.g., `"pack=1 1 16"` or `"pack=2 2 4"`. Without specifying this parameter, a default will be applied. This parameter does not expected to impact the performance of writing, it will likely make a difference for certain reading patterns if the underlying storage is using subfiles.

Optional input parameter: encoding The supported iteration encodings are either f(ile), g(roup), v(ariable). By default, we use variable encoding.

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 can be simply: `/path/8a_parallel_3Db <options>`

otherwise, please use the full name of the file.

While openPMD-api supports more than one file types, this benchmark intents to read just one type. By default, ADIOS2 file is assumed. If it is not the desired file type, one can hint with an environment variable, e.g. `export OPENPMD_BENCHMARK_USE_BACKEND=HDF5`

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 furthur 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 commmand, 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.

4.8.3 Benchmark Utilities

Further benchmarks are fund in *utilities*.

4.9 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`

4.9.1 C++

- 1_structure.cpp: creating a first series
- 2_read_serial.cpp: reading a mesh & a particle species
- 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
- 3b_write_resizable_particles.cpp: write particles in a resizeable dataset
- 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
- 10_streaming_write.cpp / 10_streaming_read.cpp: ADIOS2 data streaming
- 12_span_write.cpp: using the span-based API to save memory when writing

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

4.9.2 Python

- 2_read_serial.py: reading a mesh & a particle species
- 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
- 3b_write_resizable_particles.py: write particles in a resizeable dataset
- 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
- 10_streaming_write.py / 10_streaming_read.py: ADIOS2 data streaming
- 11_particle_dataframe.py: reading data into Pandas dataframes or Dask for distributed analysis
- 12_span_write.py: using the span-based API to save memory when writing

4.9.3 Unit Tests

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

CHAPTER

FIVE

API DETAILS

5.1 C++

Our Doxygen page provides an index of all C++ functionality.

5.1.1 Public Headers

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

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

5.1.2 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"
}
```

5.2 Python

5.2.1 Public Headers

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

Import	Description
`openpmd_api`	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.
```

5.3 MPI

5.3.1 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
`Series`	**collective**	open and close
`::flush()`	**collective**	read and write
`Iteration`[1]	independent	declare and open
`::open()`[4]	**collective**	explicit open
`Mesh`[1]	independent	declare, open, write
`ParticleSpecies`[1]	independent	declare, open, write
`::setAttribute`[3]	*backend-specific*	declare, write
`::getAttribute`	independent	open, reading
`RecordComponent`[1]	independent	declare, open, write
`::resetDataset`[1][2]	*backend-specific*	declare, write
`::makeConstant`[3]	*backend-specific*	declare, write
`::storeChunk`[1]	independent	write
`::loadChunk`	independent	read
`::availableChunks`[4]	collective	read, immediate result

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.

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

[4] 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. Alternatively, iterations might be accessed for the first time by immediate operations such as `::availableChunks()`.

[3] *HDF5* only supports collective attribute definitions/writes; *ADIOS2* attributes can be written independently. If you want to support all backends equally, treat as a collective operation. Note that openPMD represents constant record components with attributes, thus inheriting this for `::makeConstant`.

When treating attribute definitions as collective, we advise specifying the ADIOS2 *JSON/TOML key* `adios2.attribute_writing_ranks` for metadata aggregation scalabilty, typically as `adios2.attribute_writing_ranks = 0`.

[2] Dataset declarations in *parallel HDF5* are only non-collective if *chunking* is set to `none` (`auto` by default). Otherwise these operations are collective.

5.3.2 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.

5.4 JSON/TOML 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- or TOML-formatted string or via environment variables. A JSON/TOML option always takes precedence over an environment variable.

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

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

Every JSON configuration can alternatively be given by its TOML equivalent:

```
[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.

Options that can be configured via JSON are often also accessible via other means, e.g. environment variables. The following list specifies the priority of these means, beginning with the lowest priority:

1. Default values
2. Automatically detected options, e.g. the backend being detected by inspection of the file extension
3. Environment variables
4. JSON/TOML configuration. For JSON/TOML, a dataset-specific configuration overwrites a global, Series-wide configuration.
5. Explicit API calls such as `setIterationEncoding()`

The configuration is read in a case-insensitive manner, keys as well as values. An exception to this are string values which are forwarded to other libraries such as ADIOS2. Those are read "as-is" and interpreted by the backend library. 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). Dataset-specific configurations are (currently) only available during dataset creation, but not when reading datasets.

Additionally, some backends may provide different implementations to the `Series::flush()` and `Attributable::flushSeries()` calls. JSON/TOML strings may be passed to these calls as optional parameters.

A JSON/TOML configuration may either be specified as an inline string that can be parsed as a JSON/TOML object, or alternatively as a path to a JSON/TOML-formatted text file (only in the constructor of `openPMD::Series`, all other API calls that accept a JSON/TOML specification require in-line datasets):

- File paths are distinguished by prepending them with an at-sign `@`. JSON and TOML are then distinguished by the filename extension `.json` or `.toml`. If no extension can be uniquely identified, JSON is assumed as default.
- If no at-sign `@` is given, an inline string is assumed. If the first non-blank character of the string is a `{`, it will be parsed as a JSON value. Otherwise, it is parsed as a TOML value.

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 (specifying a default).
- 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.
- All dataset-specific options should be passed inside the `dataset` object, e.g.:

```json
{
  "adios2": {
    "dataset": {
      "put dataset options": "here"
    }
  }
}
```

```toml
[adios2.dataset]
# put dataset options here
```

5.4.1 Backend-independent JSON configuration

The openPMD backend can be chosen via the JSON/TOML key `backend` which recognizes the alternatives `["hdf5", "adios2", "json"]`.

The iteration encoding can be chosen via the JSON/TOML key `iteration_encoding` which recognizes the alternatives `["file_based", "group_based", "variable_based"]`. Note that for file-based iteration encoding, specification of the expansion pattern in the file name (e.g. `data_%T.json`) remains mandatory.

The key `defer_iteration_parsing` can be used to optimize the process of opening an openPMD Series (deferred/lazy parsing). By default, a Series is parsed eagerly, i.e. opening a Series implies reading all available iterations. Especially when a Series has many iterations, this can be a costly operation and users may wish to defer parsing of iterations to a later point adding `{"defer_iteration_parsing": true}` to their JSON/TOML configuration.

When parsing non-eagerly, each iteration needs to be explicitly opened with `Iteration::open()` before accessing. (Notice that `Iteration::open()` is generally recommended to be used in parallel contexts to avoid parallel file accessing hazards). Using the Streaming API (i.e. `SeriesInterface::readIteration()`) will do this automatically. Parsing eagerly might be very expensive for a Series with many iterations, but will avoid bugs by forgotten calls to `Iteration::open()`. In complex environments, calling `Iteration::open()` on an already open environment does no harm (and does not incur additional runtime cost for additional `open()` calls).

The key `resizable` can be passed to `Dataset` options. It if set to `{"resizable": true}`, this declares that it shall be allowed to increased the `Extent` of a `Dataset` via `resetDataset()` at a later time, i.e., after it has been first declared (and potentially written). For HDF5, resizable Datasets come with a performance penalty. For JSON and ADIOS2, all datasets are resizable, independent of this option.

5.4.2 Configuration Structure per Backend

ADIOS2

A full configuration of the ADIOS2 backend:

```json
{
  "adios2": {
    "engine": {
      "type": "sst",
      "preferred_flush_target": "disk",
      "parameters": {
        "BufferGrowthFactor": "2.0",
        "QueueLimit": "2"
      }
    },
    "dataset": {
      "operators": [
        {
          "type": "blosc",
          "parameters": {
            "clevel": "1",
            "doshuffle": "BLOSC_BITSHUFFLE"
          }
        }
      ]
    },
    "attribute_writing_ranks": 0
  }
}
```

```toml
[adios2]
# ignore all attribute writes not issued on these ranks
# can also be a list if multiple ranks need to be given
# however rank 0 should be the most common option here
attribute_writing_ranks = 0

[adios2.engine]
type = "sst"
preferred_flush_target = "disk"

[adios2.engine.parameters]
BufferGrowthFactor = "2.0"
QueueLimit = "2"

# use double brackets to indicate lists
[[adios2.dataset.operators]]
```

(continues on next page)

(continued from previous page)

```
type = "blosc"

# specify parameters for the current operator
[adios2.dataset.operators.parameters]
clevel = "1"
doshuffle = "BLOSC_BITSHUFFLE"

# use double brackets a second time to indicate a further entry
[[adios2.dataset.operators]]
# specify a second operator here
type = "some other operator"

# the parameters dictionary can also be specified in-line
parameters.clevel = "1"
parameters.doshuffle = "BLOSC_BITSHUFFLE"
```

All keys found under `adios2.dataset` are applicable globally as well as per dataset, any other keys such as those 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 available engine parameters. The openPMD-api does not interpret these values and instead simply forwards them to ADIOS2.
- `adios2.engine.preferred_flush_target` Only relevant for BP5 engine, possible values are `"disk"` and `"buffer"` (default: `"disk"`).
  - If `"disk"`, data will be moved to disk on every flush.
  - If `"buffer"`, then only upon ending an IO step or closing an engine.

  This behavior can be overridden on a per-flush basis by specifying this JSON/TOML key as an optional parameter to the `Series::flush()` or `Attributable::seriesFlush()` methods.

  Additionally, specifying `"disk_override"` or `"buffer_override"` will take precedence over options specified without the `_override` suffix, allowing to invert the normal precedence order. This way, a data producing code can hardcode the preferred flush target per `flush()` call, but users can e.g. still entirely deactivate flushing to disk in the `Series` constructor by specifying `preferred_flush_target = buffer_override`. This is useful when applying the asynchronous IO capabilities of the BP5 engine.
- `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)
- `adios2.use_span_based_put`: The openPMD-api exposes the span-based Put() API of ADIOS2 via an overload of `RecordComponent::storeChunk()`. This API is incompatible with compression operators as described above. The openPMD-api will automatically use a fallback implementation for the span-based Put() API if any operator is added to a dataset. This workaround is enabled on a per-dataset level. The workaround can be completely deactivated by specifying `{"adios2": {"use_span_based_put": true}}` or it can alternatively be activated indiscriminately for all datasets by specifying `{"adios2": {"use_span_based_put": false}}`.
- `adios2.attribute_writing_ranks`: A list of MPI ranks that define metadata. ADIOS2 attributes will be written only from those ranks, any other ranks will be ignored. Can be either a list of integers or a single integer.

Hint: Specifying `adios2.attribute_writing_ranks` can lead to serious serialization performance improvements at large scale.

Operations specified inside `adios2.dataset.operators` will be applied to ADIOS2 datasets in writing as well as in reading. Beginning with ADIOS2 2.8.0, this can be used to specify decompressor settings:

```
{
  "adios2": {
    "dataset": {
      "operators": [
        {
          "type": "blosc",
          "parameters": {
            "nthreads": 2
          }
        }
      ]
    }
  }
}
```

In older ADIOS2 versions, this specification will be without effect in read mode. Dataset-specific configurations are (currently) only possible when creating datasets, not when reading.

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.

HDF5

A full configuration of the HDF5 backend:

```
{
  "hdf5": {
    "dataset": {
      "chunks": "auto"
    },
    "vfd": {
      "type": "subfiling",
      "ioc_selection": "every_nth_rank",
      "stripe_size": 33554432,
      "stripe_count": -1
    }
  }
}
```

All keys found under `hdf5.dataset` are applicable globally as well as per dataset. Explanation of the single keys:

- `hdf5.dataset.chunks`: This key contains options for data chunking via H5Pset_chunk. The default is `"auto"` for a heuristic. `"none"` can be used to disable chunking.

  An explicit chunk size can be specified as a list of positive integers, e.g. `hdf5.dataset.chunks = [10, 100]`. Note that this specification should only be used per-dataset, e.g. in `resetDataset()/reset_dataset()`.

  Chunking generally improves performance and only needs to be disabled in corner-cases, e.g. when heavily relying on independent, parallel I/O that non-collectively declares data records.

- `hdf5.vfd.type` selects the HDF5 virtual file driver. Currently available are:

- "default": Equivalent to specifying nothing.
- subfiling": Use the subfiling VFD. Note that the subfiling VFD needs to be enabled explicitly when configuring HDF5 and threaded MPI must be used. When using this VFD, the options described below are additionally available. They correspond with the field entries of H5FD_subfiling_params_t, refer to the HDF5 documentation for their detailed meanings.
  * hdf5.vfd.ioc_selection: Must be one of ["one_per_node", "every_nth_rank", "with_config", "total"]
  * hdf5.vfd.stripe_size: Must be an integer
  * hdf5.vfd.stripe_count: Must be an integer

Other backends

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

CHAPTER

SIX

UTILITIES

6.1 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.

6.1.1 `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:

```
python3 -m openpmd_api.ls --help
```

6.1.2 `openpmd-pipe`

Redirect openPMD data from any source to any sink.

Any Python-enabled openPMD-api installation with enabled CLI tools comes with a command-line tool named `openpmd-pipe`. Naming and use are inspired from the piping concept known from UNIX shells.

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

```
python3 -m openpmd_api.pipe --help
```

The fundamental idea is to redirect data from an openPMD data source to another openPMD data sink. This concept becomes useful through the openPMD-api's ability to use different backends in different configurations; `openpmd-pipe` can hence be understood as a translation from one I/O configuration to another one.

Note: `openpmd-pipe` is (currently) optimized for streaming workflows in order to minimize the number of back-and-forth communications between writer and reader. All data load operations are issued in a single `flush()` per iteration. Data is loaded directly loaded into backend-provided buffers of the writer (if supported by the writer), where again only one `flush()` per iteration is used to put data to disk again. This means that the peak memory usage will be roughly equivalent to the data size of each single iteration.

The reader Series is configured by the parameters `--infile` and `--inconfig` which are both forwarded to the `filepath` and `options` parameters of the `Series` constructor. The writer Series is likewise controlled by `--outfile` and `--outconfig`.

Use of MPI is controlled by the `--mpi` and `--no-mpi` switches. If left unspecified, MPI will be used automatically if the MPI size is greater than 1.

Note: Required parameters are `--infile` and `--outfile`. Otherwise also refer to the output of `--openpmd-pipe --help`.

When using MPI, each dataset will be sliced into roughly equally-sized hyperslabs along the dimension with highest item count for load distribution across worker ranks.

If you are interested in further chunk distribution strategies (e.g. node-aware distribution, chunking-aware distribution) that are used/tested on development branches, feel free to contact us, e.g. on GitHub.

The remainder of this page discusses a select number of use cases and examples for the `openpmd-pipe` tool.

Conversion between backends

Converting from ADIOS2 to HDF5:

```
$ openpmd-pipe --infile simData_%T.bp --outfile simData_%T.h5
```

Converting from the ADIOS2 BP3 engine to the (newer) ADIOS2 BP5 engine:

```
$ openpmd-pipe --infile simData_%T.bp --outfile simData_%T.bp5

# or e.g. via inline TOML specification (also possible: JSON)
$ openpmd-pipe --infile simData_%T.bp --outfile output_folder/simData_%T.bp \
     --outconfig 'adios2.engine.type = "bp5"'
# the config can also be read from a file, e.g. --outconfig @cfg.toml
#                                          or   --outconfig @cfg.json
```

Converting between iteration encodings

Converting to group-based iteration encoding:

```
$ openpmd-pipe --infile simData_%T.h5 --outfile simData.h5
```

Converting to variable-based iteration encoding (not yet feature-complete):

```
# e.g. specified via inline JSON
$ openpmd-pipe --infile simData_%T.bp --outfile simData.bp \
    --outconfig '{"iteration_encoding": "variable_based"}'
```

Capturing a stream

Since the openPMD-api also supports streaming/staging I/O transports from ADIOS2, `openpmd-pipe` can be used to capture a stream in order to write it to disk. In the ADIOS2 SST engine, a stream can have any number of readers. This makes it possible to intercept a stream in a data processing pipeline.

```
$ cat << EOF > streamParams.toml
[adios2.engine.parameters]
DataTransport = "fabric"
OpenTimeoutSecs = 600
EOF

$ openpmd-pipe --infile streamContactFile.sst --inconfig @streamParams.toml \
    --outfile capturedStreamData_%06T.bp

# Just loading and discarding streaming data, e.g. for performance benchmarking:
```

(continues on next page)

(continued from previous page)

```
$ openpmd-pipe --infile streamContactFile.sst --inconfig @streamParams.toml \
    --outfile null.bp --outconfig 'adios2.engine.type = "nullcore"'
```

Defragmenting a file

Due to the file layout of ADIOS2, especially mesh-refinement-enabled simulation codes can create file output that is very strongly fragmented. Since only one `load_chunk()` and one `store_chunk()` call is issued per MPI rank, per dataset and per iteration, the file is implicitly defragmented by the backend when passed through `openpmd-pipe`:

```
$ openpmd-pipe --infile strongly_fragmented_%T.bp --outfile defragmented_%T.bp
```

Post-hoc compression

The openPMD-api can be directly used to compress data already when originally creating it. When however intending to compress data that has been written without compression enabled, `openpmd-pipe` can help:

```
$ cat << EOF > compression_cfg.json
{
  "adios2": {
    "dataset": {
      "operators": [
        {
          "type": "blosc",
          "parameters": {
            "clevel": 1,
            "doshuffle": "BLOSC_BITSHUFFLE"
          }
        }
      ]
    }
  }
}
EOF

$ openpmd-pipe --infile not_compressed_%T.bp --outfile compressed_%T.bp \
    --outconfig @compression_cfg.json
```

Starting point for custom transformation and analysis

`openpmd-pipe` is a Python script that can serve as basis for custom extensions, e.g. for adding, modifying, transforming or reducing data. The typical use case would be as a building block in a domain-specific data processing pipeline.

6.2 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.

6.2.1 Example Usage

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

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

#include <iostream>
#include <string>
#include <vector>

#if openPMD_HAVE_MPI
inline void print_help(std::string const &program_name)
{
    std::cout << "Usage: " << program_name << "\n";
    std::cout << "Run a simple parallel write and read benchmark.\n\n";
    std::cout << "Options:\n";
    std::cout
        << "    -w, --weak    run a weak scaling (default: strong scaling)\n";
    std::cout << "    -h, --help    display this help and exit\n";
    std::cout << "    -v, --version output version information and exit\n";
    std::cout << "\n";
    std::cout << "Examples:\n";
    std::cout << "    " << program_name << " --weak  # for a weak-scaling\n";
    std::cout << "    " << program_name << "  # for a strong scaling\n";
}

inline void print_version(std::string const &program_name)
{
    std::cout << program_name << " (openPMD-api) " << openPMD::getVersion()
              << "\n";
    std::cout << "Copyright 2017-2021 openPMD contributors\n";
    std::cout << "Authors: Franz Poeschel, Axel Huebl et al.\n";
    std::cout << "License: LGPLv3+\n";
    std::cout << "This is free software: you are free to change and "
                 "redistribute it.\n"
                 "There is NO WARRANTY, to the extent permitted by law.\n";
}

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

    // CLI parsing
    std::vector<std::string> str_argv;
    str_argv.reserve(argc);
    for (int i = 0; i < argc; ++i)
        str_argv.emplace_back(argv[i]);
    bool weak_scaling = false;

    for (int c = 1; c < int(argc); c++)
    {
```

(continues on next page)

(continued from previous page)

```
        if (std::string("--help") == argv[c] || std::string("-h") == argv[c])
        {
            print_help(argv[0]);
            return 0;
        }
        if (std::string("--version") == argv[c] || std::string("-v") == argv[c])
        {
            print_version(argv[0]);
            return 0;
        }
        if (std::string("--weak") == argv[c] || std::string("-w") == argv[c])
        {
            weak_scaling = true;
        }
    }

    if (argc > 2)
    {
        std::cerr << "Too many arguments! See: " << argv[0] << " --help\n";
        return 1;
    }

    // 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 = uint64_t;
#if openPMD_HAVE_ADIOS2 || openPMD_HAVE_HDF5
    openPMD::Datatype dt = openPMD::determineDatatype<type>();
#endif

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    const unsigned scale_up = weak_scaling ? unsigned(size) : 1u;

    // Total (in this case 3D) dataset across all MPI ranks.
    // Will be the same for all configured benchmarks.
    openPMD::Extent total{100 * scale_up, 100, 1000};

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

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

    // The Benchmark class will in principle allow a user to configure
    // runs that write and read different datatypes.
```

(continues on next page)

(continued from previous page)

```
    // 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:
    // * 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_ADIOS2
    benchmark.addConfiguration(
        R"({"adios2": {"dataset":{"operators":[{"type": "blosc"}]}}})",
        "bp",
        dt,
        10);
#endif
#if openPMD_HAVE_HDF5
    benchmark.addConfiguration("{}", "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>();

    if (rank == 0)
    {
        for (auto it = res.durations.begin(); it != res.durations.end(); it++)
        {
            auto time = it->second;
```

(continues on next page)

(continued from previous page)

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

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

CHAPTER

SEVEN

BACKENDS

7.1 Overview

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

Feature	ADIOS1	ADIOS2	HDF5	JSON
Operating Systems	Linux, OSX	Linux, OSX, Windows		
Status	end-of-life	active	active	active
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
Memory footprint	large	medium	small	small
Performance	A-	A	B	C
Native File Format	`.bp` (BP3)	`.bp` (BP3-5)	`.h5`	`.json`

ADIOS1 was removed in version 0.16.0. Please use ADIOS2 instead.

- 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)

7.1.1 Selected References

- Franz Poeschel, Juncheng E, William F. Godoy, Norbert Podhorszki, Scott Klasky, Greg Eisenhauer, Philip E. Davis, Lipeng Wan, Ana Gainaru, Junmin Gu, Fabian Koller, Rene Widera, Michael Bussmann, and Axel Huebl. *Transitioning from file-based HPC workflows to streaming data pipelines with openPMD and ADIOS2,* Part of *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation,* SMC 2021, Communications in Computer and Information Science (CCIS), vol 1512, 2022. arXiv:2107.06108, DOI:10.1007/978-3-030-96498-6_6
- 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

7.2 JSON/TOML

openPMD supports writing to and reading from JSON and TOML files. The JSON and TOML backends are always available.

Note: Both the JSON and the TOML backends are not intended for large-scale data I/O.

The JSON backend is mainly intended for prototyping and learning, or similar workflows where setting up a large IO backend such as HDF5 or ADIOS2 is perceived as obstructive. It can also be used for small datasets that need to be stored in text format rather than binary.

The TOML backend is intended for exchanging the *structure* of a data series without its "heavy" data fields. For instance, one can easily create and exchange human-readable, machine-actionable data configurations for experiments and simulations.

7.2.1 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`.

7.2.2 TOML File Format

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

The TOML backend internally works with JSON datasets and converts to/from TOML during I/O. As a result, data layout and usage are equivalent to the JSON backend.

7.2.3 JSON 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.

7.2.4 TOML Restrictions

Note that the JSON datatype-specific restrictions do not automatically hold for TOML, as those affect only the representation on disk, not the internal representation.

TOML supports most numeric types, with the support for long double and long integer types being platform-defined. Special floating point values such as NaN are also support.

TOML does not support null values.

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

7.2.5 Using in parallel (MPI)

Parallel I/O is not a first-class citizen in the JSON and TOML backends, and neither backend will "go out of its way" to support parallel workflows.

However there is a rudimentary form of read and write support in parallel:

Parallel reading

In order not to overload the parallel filesystem with parallel reads, read access to JSON datasets is done by rank 0 and then broadcast to all other ranks. Note that there is no granularity whatsoever in reading a JSON file. A JSON file is always read into memory and broadcast to all other ranks in its entirety.

Parallel writing

When executed in an MPI context, the JSON/TOML backends will not directly output a single text file, but instead a folder containing one file per MPI rank. Neither backend will perform any data aggregation at all.

Note: The parallel write support of the JSON/TOML backends is intended mainly for debugging and prototyping workflows.

The folder will use the specified Series name, but append the postfix `.parallel`. (This is a deliberate indication that this folder cannot directly be opened again by the openPMD-api as a JSON/TOML dataset.) This folder contains for each MPI rank *i* a file `mpi_rank_<i>.json` (resp. `mpi_rank_<i>.toml`), containing the serial output of that rank. A `README.txt` with basic usage instructions is also written.

Note: There is no direct support in the openPMD-api to read a JSON/TOML dataset written in this parallel fashion. The single files (e.g. `data.json.parallel/mpi_rank_0.json`) are each valid openPMD files and can be read separately, however.

Note that the auxiliary function `json::merge()` (or in Python `openpmd_api.merge_json()`) is not adequate for merging the single JSON/TOML files back into one, since it does not merge anything below the array level.

7.2.6 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": {
```

(continues on next page)

(continued from previous page)

```
      "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",
              "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": [
```

(continues on next page)

(continued from previous page)

```
                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
            ],
            [
              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,
```

(continues on next page)

(continued from previous page)

```
    "UCHAR": 1,
    "UINT": 4,
    "ULONG": 8,
    "ULONGLONG": 8,
    "USHORT": 2
  }
}
```

7.3 ADIOS1

The ADIOS1 library is no longer developed in favor of ADIOS2. Consequently, ADIOS1 support was removed in openPMD-api 0.16.0 and newer. Please transition to ADIOS2.

For reading legacy ADIOS1 BP3 files, either use an older version of openPMD-api or the BP3 backend in ADIOS2. Note that ADIOS2 does not support compression in BP3 files.

7.3.1 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

7.4 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*.

7.4.1 I/O Method and Engine Selection

ADIOS2 has several engines for alternative file formats and other kinds of backends, yet natively writes to `.bp` files. The openPMD API uses the File meta engine as the default file engine and the SST engine for streaming support.

The ADIOS2 engine can be selected in different ways:

1. Automatic detection via the selected file ending
2. Explicit selection of an engine by specifying the environment variable `OPENPMD_ADIOS2_ENGINE` (case-independent). This overrides the automatically detected engine.
3. Explicit selection of an engine by specifying the JSON/TOML key `adios2.engine.type` as a case-independent string. This overrides both previous options.

Automatic engine detection supports the following extensions:

Extension	Selected ADIOS2 Engine
`.bp`	`"file"`
`.bp4`	`"bp4"`
`.bp5`	`"bp5"`
`.sst`	`"sst"`
`.ssc`	`"ssc"`

Specifying any of these extensions will automatically activate the ADIOS2 backend. The ADIOS2 backend will create file system paths exactly as they were specified and not change file extensions. Exceptions to this are the BP3 and SST engines which require their endings `.bp` and `.sst` respectively.

For file engines, we currently leverage the default ADIOS2 transport parameters, i.e. `POSIX` on Unix systems and `FStream` on Windows.

7.4.2 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).

ADIOS2 release 2.6.0 contained a bug (fixed in ADIOS 2.7.0, see PR #2348) that disallows random-accessing steps in file-based engines. With this ADIOS2 release, files written with steps 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.

7.4.3 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	default	description
`OPENPMD_ADIOS2_HAVI`	1	Turns on/off profiling information right after a run.
`OPENPMD_ADIOS2_HAVI`	1	Online creation of the adios journal file (1: yes, `0`: no).
`OPENPMD_ADIOS2_NUM_`	`0`	Number of files to be created, 0 indicates maximum number possible.
`OPENPMD_ADIOS2_ENGI`	`File`	ADIOS2 engine
`OPENPMD2_ADIOS2_USI`	`0`	Use group table (see below)
`OPENPMD_ADIOS2_STAT`	`0`	whether to generate statistics for variables in ADIOS2. (1: yes, `0`: no).
`OPENPMD_ADIOS2_ASYI`	`0`	ADIOS2 BP5 engine: 1 means setting "AsyncWrite" in ADIOS2 to "on". Flushes will go to the buffer by default (see `preferred_flush_target`).
`OPENPMD_ADIOS2_BP5_`	`0`	ADIOS2 BP5 engine: applies when using either EveryoneWrites or EveryoneWritesSerial aggregation
`OPENPMD_ADIOS2_BP5_`	`0`	ADIOS2 BP5 engine: applies when using TwoLevelShm aggregation
`OPENPMD_ADIOS2_BP5_`	`0`	ADIOS2 BP5 engine: num of subfiles
`OPENPMD_ADIOS2_BP5_`	`0`	ADIOS2 BP5 engine: num of aggregators
`OPENPMD_ADIOS2_BP5_`	*empty*	ADIOS2 BP5 engine: aggregation type. (EveryoneWrites, EveryoneWritesSerial, TwoLevelShm)

Please refer to the ADIOS2 documentation for details on I/O tuning.

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"`.

The ADIOS2 backend overrides the default unlimited queueing behavior of the SST engine with a more cautious limit of 2 steps that may be held in the queue at one time. The default behavior may be restored by setting the *JSON parameter* `adios2.engine.parameters.QueueLimit = "0"`.

7.4.4 Best Practice at Large Scale

ADIOS2 distinguishes between "heavy" data of arbitrary size (i.e. the "actual" data) and lightweight metadata.

Heavy I/O

A benefitial configuration depends heavily on:

1. Hardware: filesystem type, specific file striping, network infrastructure and available RAM on the aggregator nodes.
2. Software: communication and I/O patterns in the data producer/consumer, ADIOS2 engine being used.

The BP4 engine optimizes aggressively for I/O efficiency at large scale, while the BP5 engine implements some compromises for tighter control of host memory usage.

ADIOS2 aggregates at two levels:

1. Aggregators: These are the processes that actually write data to the filesystem. In BP5, there must be at least one aggregatore per compute node.

2. Subfiles: In BP5, multiple aggregators might write to the same physical file on the filesystem. The BP4 engine does not distinguish the number of aggregators from the number of subfiles, each aggregator writes to one file.

The number of aggregators depends on the actual scale of the application. At low and mediocre scale, it is generally preferred to have every process write to the filesystem in order to make good use of parallel resources and utilize the full bandwidth. At higher scale, reducing the number of aggregators is suggested, in order to avoid competition for resources between too many writing processes. In the latter case, a good number of aggregators is usually the number of contributing nodes. A file count lower than the number of nodes might be chosen in both BP4 and BP5 with care, file counts of "number of nodes divided by four" have yielded good results in some setups.

Use of asynchronous I/O functionality (`BurstBufferPath` in BP4, `AsyncWrite` in BP5) depends on the application, and might increase the performance or decrease it. Asynchronous I/O can compete with MPI for communication resources, impacting the *compute* performance of an application.

For SST streaming, the default TCP-based backend does not scale well in HPC situations. Instead, a high-performance backend (`libfabric`, `ucx` or `mpi` (only supported for well-configured MPICH)) should be chosen. The preferred backend usually depends on the system's native software stack.

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.

Metadata

ADIOS2 will implicitly aggregate metadata specified from parallel MPI processes. Duplicate specification of metadata is eliminated in this process. Unlike in HDF5, specifying metadata collectively is not required and is even detrimental to performance. The *JSON/TOML key* `adios2.attribute_writing_ranks` can be used to restrict attribute writing to only a select handful of ranks (most typically a single one). The ADIOS2 backend of the openPMD-api will then ignore attributes from all other MPI ranks.

Tip: Treat metadata specification as a collective operation in order to retain compatibility with HDF5, and then specify `adios2.attribute_writing_ranks = 0` in order to achieve best performance in ADIOS2.

Warning: The ADIOS2 backend may also use attributes to encode openPMD groups (ref. "group table"). The `adios2.attribute_writing_ranks` key also applies to those attributes, i.e. also group creation must be treated as collective then (at least on the specified ranks).

7.4.5 Experimental group table feature

We are experimenting with a feature that will make the structure of an ADIOS2 file more explicit. Currently, the hierarchical structure of an openPMD dataset in ADIOS2 is recovered implicitly by inspecting variables and attributes found in the ADIOS2 file. Inspecting attributes is necessary since not every openPMD group necessarily contains an (array) dataset. The downside of this approach is that ADIOS2 attributes do not properly interact with ADIOS2 steps, resulting in many problems and workarounds when parsing an ADIOS2 dataset. An attribute, once defined, cannot be deleted, implying that the ADIOS2 backend will recover groups that might not actually be logically present in the current step.

As a result of this behavior, support for ADIOS2 steps is currently restricted.

For full support of ADIOS2 steps, we introduce a group table that makes use of modifiable attributes in ADIOS2 v2.9, i.e. attributes that can have different values across steps.

An openPMD group `<group>` is present if:

1. The integer attribute `__openPMD_groups/<group>` exists
2. and:

a. the file is either accessed in random-access mode

  b. or the current value of said attribute is equivalent to the current step index.

This feature can be activated via the JSON/TOML key `adios2.use_group_table = true` or via the environment variable `OPENPMD2_ADIOS2_USE_GROUP_TABLE=1`. It is fully backward-compatible with the old layout of openPMD in ADIOS2 and mostly forward-compatible (except the support for steps).

The variable-based encoding of openPMD automatically activates the group table feature.

7.4.6 Memory usage

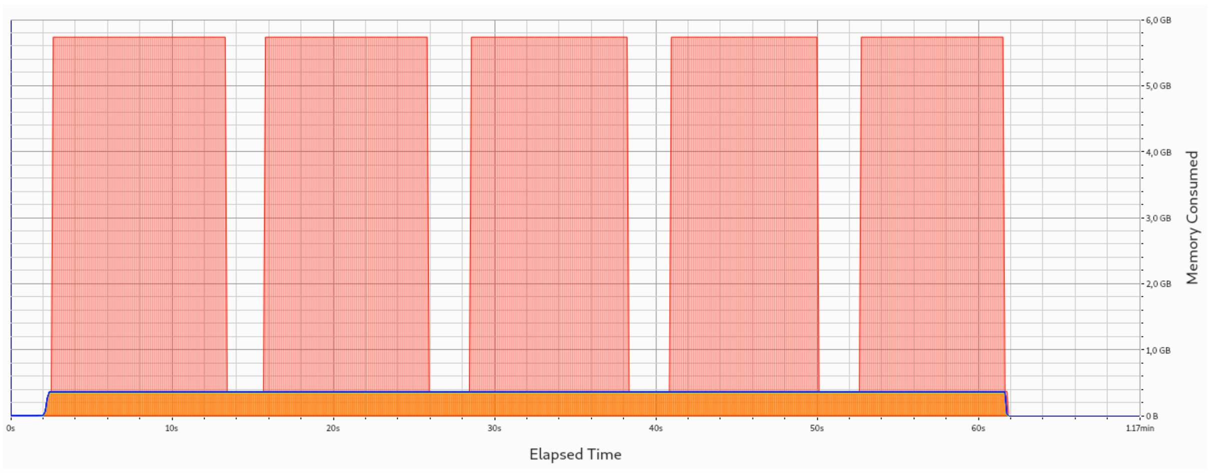
The IO strategy in ADIOS2 is to stage all written data in a large per-process buffer. This buffer is drained to storage only at specific times:

1. When an engine is closed.
2. When a step is closed.

The usage pattern of openPMD, especially the choice of iteration encoding influences the memory use of ADIOS2. The following graphs are created from a real-world application using openPMD (PIConGPU) using KDE Heaptrack.

The internal data structure of BP4 is one large buffer that holds all data written by a process. It is drained to the disk upon ending a step or closing the engine (in parallel applications, data will usually be aggregated at the node-level before this). This approach enables a very high IO performance by requiring only very few, very large IO operations, at the cost of a high memory consumption and some common usage pitfalls as detailed below:

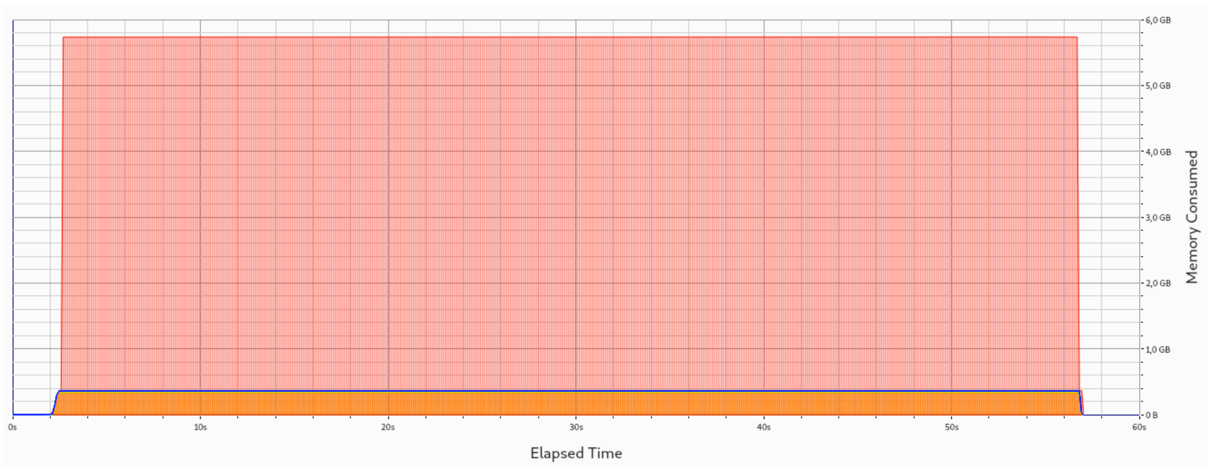
- **file-based iteration encoding:** A new ADIOS2 engine is opened for each iteration and closed upon `Iteration::close()`. Each iteration has its own buffer:

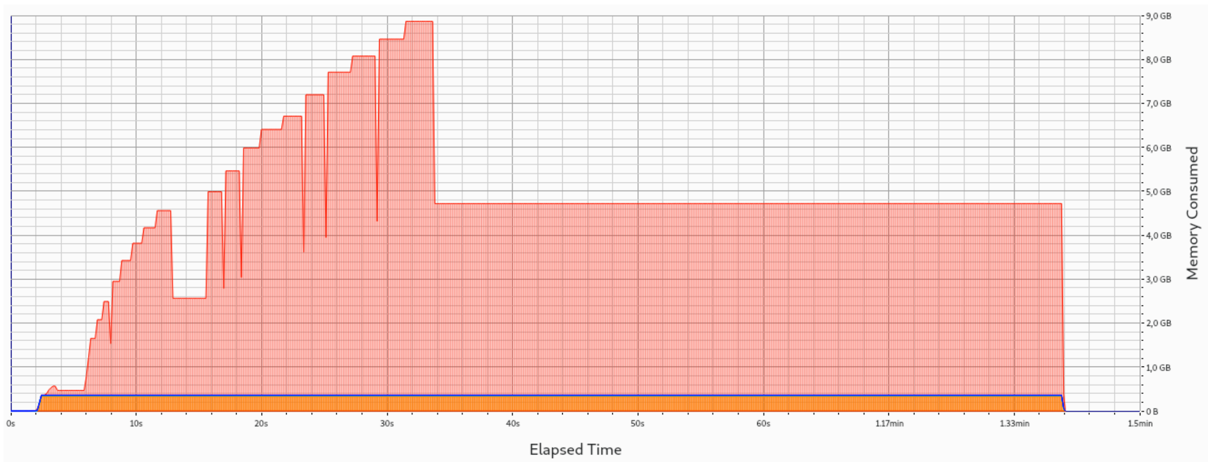


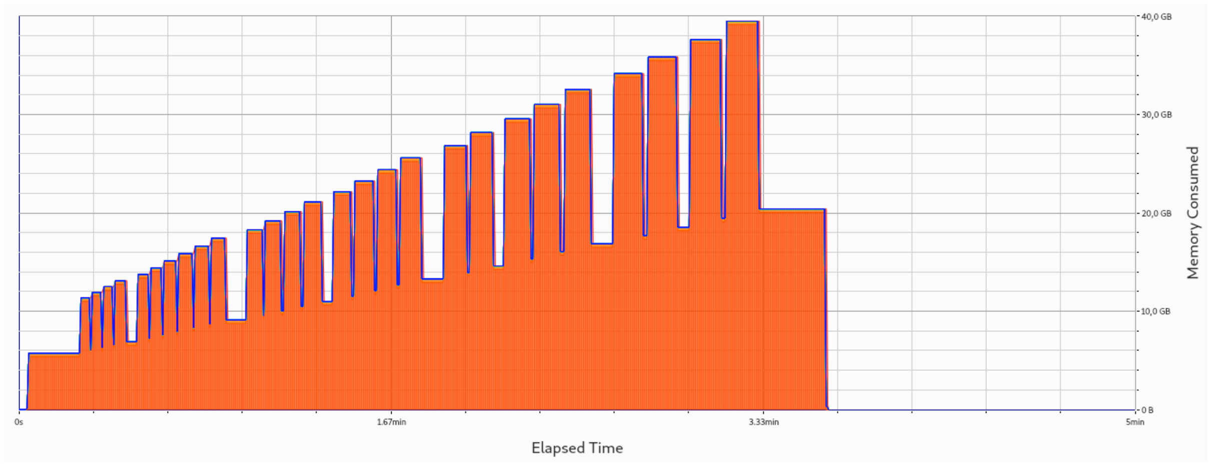
- **variable-based iteration encoding and group-based iteration encoding with steps**: One buffer is created and reused across all iterations. It is drained to disk when closing a step. If carefully selecting the correct `InitialBufferSize`, this is merely one single allocation held across all iterations. If selecting the `InitialBufferSize` too small, reallocations will occur. As usual with `std::vector` (which ADIOS2 uses internally), a reallocation will occupy both the old and new memory for a short time, leading to small memory spikes. These memory spikes can easily lead to out-of-memory (OOM) situations, motivating that the `InitialBufferSize` should not be chosen too small. Both behaviors are depicted in the following two pictures:
- **group-based iteration encoding without steps:** This encoding **should be avoided** in ADIOS2. No data will be written to disk before closing the `Series`, leading to a continuous buildup of memory, and most likely to an OOM situation:

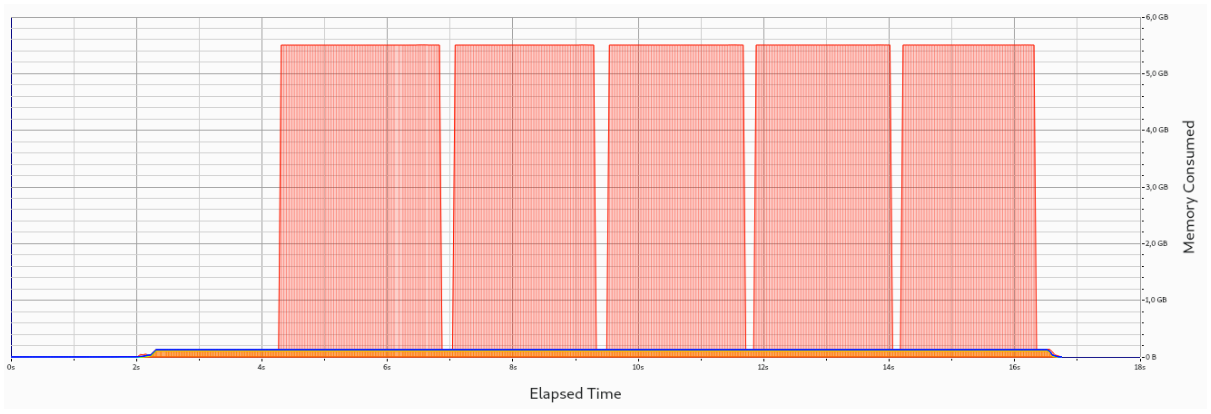
Like the BP4 engine, the SST engine uses one large buffer as an internal data structure.

Unlike the BP4 engine, however, a new buffer is allocated for each IO step, leading to a memory profile with clearly distinct IO steps:

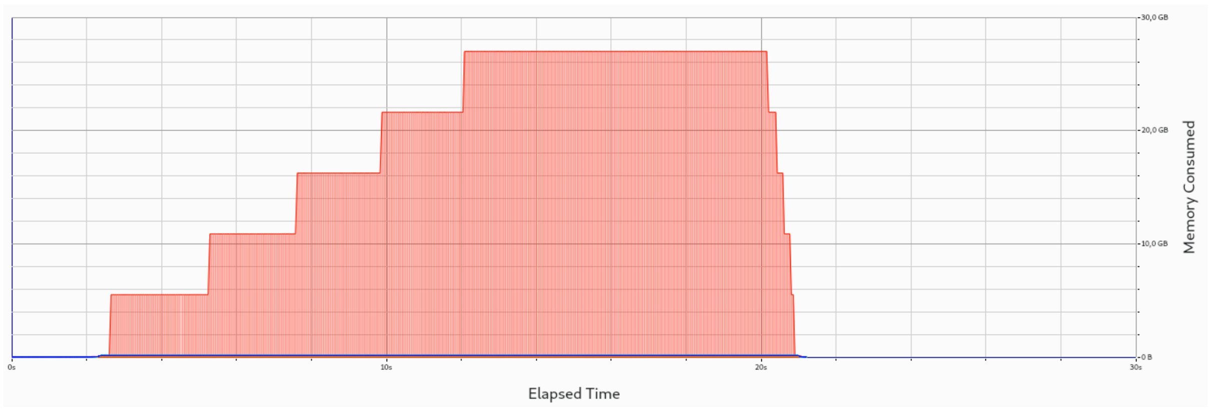

6,0 GB
5,0 GB
4,0 GB
3,0 GB
2,0 GB
1,0 GB
0 B
Memory Consumed
0s
10s
20s
30s
40s
50s
60s
Elapsed Time


9,0 GB
8,0 GB
7,0 GB
6,0 GB
5,0 GB
4,0 GB
3,0 GB
2,0 GB
1,0 GB
0 B
Memory Consumed
0s
10s
20s
30s
40s
50s
60s
1.17min
1.33min
1.5min
Elapsed Time


40,0 GB
30,0 GB
20,0 GB
10,0 GB
0 B
Memory Consumed
0s
1.67min
3.33min
5min
Elapsed Time


6,0 GB
5,0 GB
4,0 GB
3,0 GB
2,0 GB
1,0 GB
0 B
Memory Consumed
0s
2s
4s
6s
8s
10s
12s
14s
16s
18s
Elapsed Time

The SST engine performs all IO asynchronously in the background and releases memory only as soon as the reader is done interacting with an IO step. With slow readers, this can lead to a buildup of past IO steps in memory and subsequently to an out-of-memory condition:



This can be avoided by specifying the ADIOS2 parameter `QueueLimit`:

```
std::string const adios2Config = R"(
  {"adios2": {"engine": {"parameters": {"QueueLimit": 1}}}}
)";
Series series("simData.sst", Access::CREATE, adios2Config);
```

By default, the openPMD-api configures a queue limit of 2. Depending on the value of the ADIOS2 parameter `QueueFullPolicy`, the SST engine will either `"Discard"` steps or `"Block"` the writer upon reaching the queue limit.

The BP5 file engine internally uses a linked list of equally-sized buffers. The size of each buffer can be specified up to a maximum of 2GB with the ADIOS2 parameter `BufferChunkSize`:

```
std::string const adios2Config = R"(
  {"adios2": {"engine": {"parameters": {"BufferChunkSize": 2147381248}}}}
)";
Series series("simData.bp5", Access::CREATE, adios2Config);
```

This approach implies a sligthly lower IO performance due to more frequent and smaller writes, but it lets users control memory usage better and avoids out-of-memory issues when configuring ADIOS2 incorrectly.

The buffer is drained upon closing a step or the engine, but draining to the filesystem can also be triggered manually. In the openPMD-api, this can be done by specifying backend-specific parameters to the `Series::flush()` or `Attributable::seriesFlush()` calls:

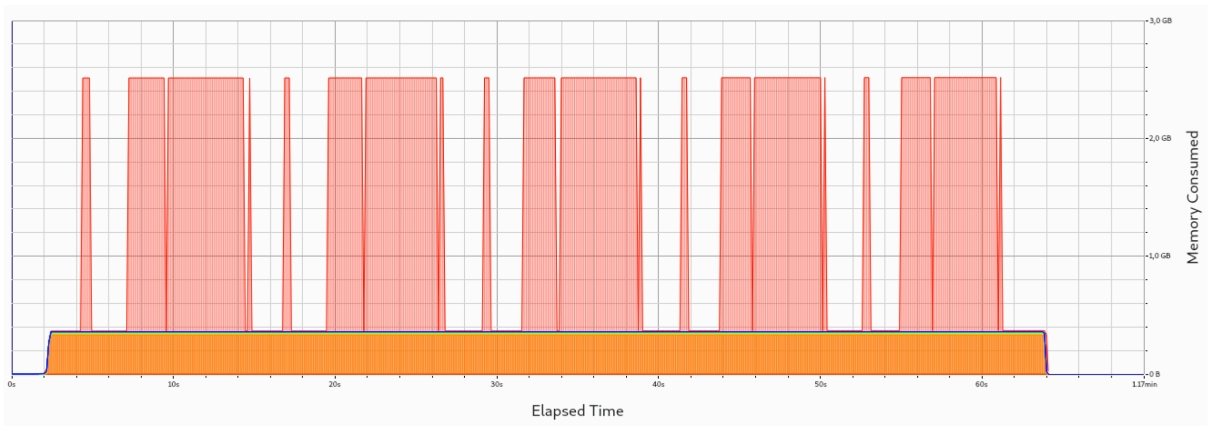
```
series.flush(R"({"adios2": {"engine": {"preferred_flush_target": "disk"}}})")
```

The memory consumption of this approach shows that the 2GB buffer is first drained and then recreated after each `flush()`:

Note: KDE Heaptrack tracks the **virtual memory** consumption. While the BP4 engine uses `std::vector<char>` for its internal buffer, BP5 uses plain `malloc()` (hence the 2GB limit), which does not initialize memory. Memory pages will only be allocated to physical memory upon writing. In applications with small IO sizes on systems with virtual memory, the physical memory usage will stay well below 2GB even if specifying the BufferChunkSize as 2GB.

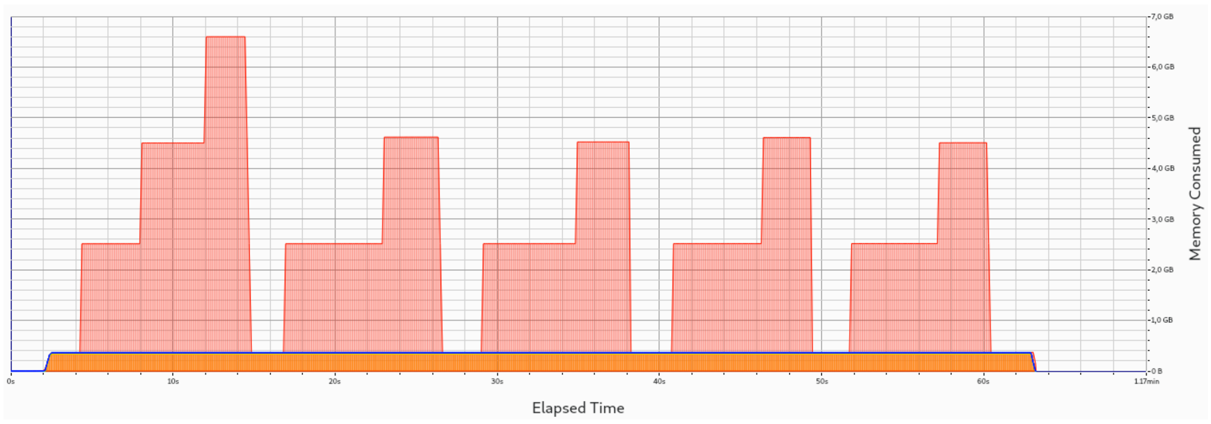
=> Specifying the buffer chunk size as 2GB as shown above is a good idea in most cases.

Alternatively, data can be flushed to the buffer. Note that this involves data copies that can be avoided by either flushing directly to disk or by entirely avoiding to flush until `Iteration::close()`:



```
series.flush(R"({"adios2": {"engine": {"preferred_flush_target": "buffer"}}})")
```

With this strategy, the BP5 engine will slowly build up its buffer until ending the step. Rather than by reallocation as in BP4, this is done by appending a new chunk, leading to a clearly more acceptable memory profile:



The default is to flush to disk (except when specifying `OPENPMD_ADIOS2_ASYNC_WRITE=1`), but the default `preferred_flush_target` can also be specified via JSON/TOML at the `Series` level.

7.4.7 Known Issues

Warning: Nov 1st, 2021 (ADIOS2 2887): The fabric selection in ADIOS2 has was designed for libfabric 1.6. With newer versions of libfabric, the following workaround is needed to guide the selection of a functional fabric for RDMA support:

The following environment variables can be set as work-arounds on Cray systems, when working with ADIOS2 SST:

```
export FABRIC_IFACE=mlx5_0   # ADIOS SST: select interface (1 NIC on Summit)
export FI_OFI_RXM_USE_SRX=1  # libfabric: use shared receive context from MSG␣
↪provider
```

7.4.8 Selected References

- William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. *ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management,* SoftwareX, vol. 12, 100561, 2020. DOI:10.1016/j.softx.2020.100561
- Franz Poeschel, Juncheng E, William F. Godoy, Norbert Podhorszki, Scott Klasky, Greg Eisenhauer, Philip E. Davis, Lipeng Wan, Ana Gainaru, Junmin Gu, Fabian Koller, Rene Widera, Michael Bussmann, and Axel Huebl. *Transitioning from file-based HPC workflows to streaming data pipelines with openPMD and ADIOS2,* Part of *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation,* SMC 2021, Communications in Computer and Information Science (CCIS), vol 1512, 2022. arXiv:2107.06108, DOI:10.1007/978-3-030-96498-6_6
- Lipeng Wan, Axel Huebl, Junmin Gu, Franz Poeschel, Ana Gainaru, Ruonan Wang, Jieyang Chen, Xin Liang, Dmitry Ganyushin, Todd Munson, Ian Foster, Jean-Luc Vay, Norbert Podhorszki, Kesheng Wu, and Scott Klasky. *Improving I/O Performance for Exascale Applications through Online Data Layout Reorganization,* IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 4, pp. 878-890, 2022. arXiv:2107.07108, DOI:10.1109/TPDS.2021.3100784
- Junmin Gu, Philip Davis, Greg Eisenhauer, William Godoy, Axel Huebl, Scott Klasky, Manish Parashar, Norbert Podhorszki, Franz Poeschel, Jean-Luc Vay, Lipeng Wan, Ruonan Wang, and Kesheng Wu. *Organizing Large Data Sets for Efficient Analyses on HPC Systems,* Journal of Physics: Conference Series, vol. 2224, in *2nd International Symposium on Automation, Information and Computing* (ISAIC 2021), 2022. DOI:10.1088/1742-6596/2224/1/012042
- 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

7.5 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*.

7.5.1 I/O Method

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

Virtual File Drivers

Rudimentary support for HDF5 VFDs (virtual file driver) is available (currently only the *subfiling* VFD). Note that the subfiling VFD needs to be enabled explicitly when configuring HDF5 and threaded MPI must be used.

Virtual file drivers are configured via JSON/TOML. Refer to the page on JSON/TOML configuration for further details.

7.5.2 Backend-Specific Controls

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

Environment variable	Default	Description
`OPENPMD_HDF5_INDEPENDEN`	`ON`	Sets the MPI-parallel transfer mode to collective (`OFF`) or independent (`ON`).
`OPENPMD_HDF5_ALIGNMENT`	`1`	Tuning parameter for parallel I/O, choose an alignment which is a multiple of the disk block size.
`OPENPMD_HDF5_THRESHOLD`	`0`	Tuning parameter for parallel I/O, where `0` aligns all requests and other values act as a threshold.
`OPENPMD_HDF5_CHUNKS`	`auto`	Defaults for `H5Pset_chunk`: `"auto"` (heuristic) or `"none"` (no chunking).
`OPENPMD_HDF5_COLLECTIVE_`	`ON`	Sets the MPI-parallel transfer mode for metadata operations to collective (`ON`) or independent (`OFF`).
`OPENPMD_HDF5_PAGED_ALLO`	`ON`	Tuning parameter for parallel I/O in HDF5 to enable paged allocation.
`OPENPMD_HDF5_PAGED_ALLO`	`335544`	Size of the page, in bytes, if HDF5 paged allocation optimization is enabled.
`OPENPMD_HDF5_DEFER_META`	`ON`	Tuning parameter for parallel I/O in HDF5 to enable deferred HDF5 metadata operations.
`OPENPMD_HDF5_DEFER_META`	`ON`	Size of the buffer, in bytes, if HDF5 deferred metadata optimization is enabled.
`HDF5_USE_FILE_LOCKING`	`TRUE`	Work-around: Set to `FALSE` in case you are on an HPC or network file system that hang in open for reads.
`HDF5_DO_MPI_FILE_SYNC`	driver-dep.	Work-around: Set to `FALSE` to overcome MPI-I/O synchronization issues on some filesystems, e.g., NFS.
`H5_COLL_API_SANITY_CHEC`	unset	Debug: Set to `1` to perform an `MPI_Barrier` inside each meta-data operation.
`OMPI_MCA_io`	unset	Work-around: Disable OpenMPI's I/O implementation for older releases by setting this to `^ompio`.

`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).

`OPENPMD_HDF5_THRESHOLD`: this sets the threshold for the alignment of HDF5 operations via the `H5Pset_alignment` function. Setting it to `0` will force all requests to be aligned. Any file object greater than or equal in size to threshold bytes will be aligned on an address which is a multiple of `OPENPMD_HDF5_ALIGNMENT`.

`OPENPMD_HDF5_CHUNKS`: this sets defaults for data chunking via H5Pset_chunk. Chunking generally improves performance and only needs to be disabled in corner-cases, e.g. when heavily relying on independent, parallel I/O that non-collectively declares data records. The chunk size can alternatively (or additionally) be specified explicitly per dataset, by specifying a dataset-specific chunk size in the JSON/TOML configuration of `resetDataset()`/`reset_dataset()`.

`OPENPMD_HDF5_COLLECTIVE_METADATA`: this is an option to enable collective MPI calls for HDF5 metadata operations via H5Pset_all_coll_metadata_ops and H5Pset_coll_metadata_write. By default, this optimization is enabled as it has proven to provide performance improvements. This option is only available from HDF5 1.10.0 onwards. For previous version it will fallback to independent MPI calls.

`OPENPMD_HDF5_PAGED_ALLOCATION`: this option enables paged allocation for HDF5 operations via H5Pset_file_space_strategy. The page size can be controlled by the `OPENPMD_HDF5_PAGED_ALLOCATION_SIZE` option.

`OPENPMD_HDF5_PAGED_ALLOCATION_SIZE`: this option configures the size of the page if `OPENPMD_HDF5_PAGED_ALLOCATION` optimization is enabled via H5Pset_file_space_page_size. Values are expressed in bytes. Default is set to 32MB.

`OPENPMD_HDF5_DEFER_METADATA`: this option enables deffered HDF5 metadata operations. The metadata buffer size can be controlled by the `OPENPMD_HDF5_DEFER_METADATA_SIZE` option.

`OPENPMD_HDF5_DEFER_METADATA_SIZE`: this option configures the size of the buffer if `OPENPMD_HDF5_DEFER_METADATA` optimization is enabled via H5Pset_mdc_config. Values are expressed in bytes. Default is set to 32MB.

`HDF5_USE_FILE_LOCKING`: this is a HDF5 1.10.1+ control option that disables HDF5 internal file locking operations (see HDF5 1.10.1 release notes). This mechanism is mainly used to ensure that a file that is still being written to cannot (yet) be opened by either a reader or another writer. On some HPC and Jupyter systems, parallel/network file systems like GPFS are mounted in a way that interferes with this internal, HDF5 access consistency check. As a result, read-only operations like `h5ls some_file.h5` or openPMD `Series` open can hang indefinitely. If you are sure that the file was written completely and is closed by the writer, e.g., because a simulation finished that created HDF5 outputs, then you can set this environment variable to `FALSE` to work-around the problem. You should also report this problem to your system support, so they can fix the file system mount options or disable locking by default in the provided HDF5 installation.

`HDF5_DO_MPI_FILE_SYNC`: this is an MPI-parallel HDF5 1.14+ control option that adds an `MPI_File_sync()` call after every collective write operation. This is sometimes needed by the underlying parallel MPI-I/O driver if the filesystem has very limited parallel features. Examples are NFS and UnifyFS, where this can be used to overcome synchronization issues/crashes. The default value for this is *MPI-IO driver-dependent* and defaults to `TRUE` for these filesystems in newer HDF5 versions. Setting the value back to `FALSE` has been shown to overcome issues on NFS with parallel HDF5. Note that excessive sync calls can severely reduce parallel write performance, so `TRUE` should only be used when truly needed for correctness/stability.

`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.

`OMPI_MCA_io`: this is an OpenMPI control variable. OpenMPI implements its own MPI-I/O implementation backend *OMPIO*, starting with OpenMPI 2.x . This backend is known to cause problems in older releases that might still be in use on some systems. Specifically, we found and reported a silent data corruption issue that was fixed only in OpenMPI versions 3.0.4, 3.1.4, 4.0.1 and newer. There are also problems in OMPIO with writes larger than 2GB, which have only been fixed in OpenMPI version 3.0.5, 3.1.5, 4.0.3 and newer. Using `export OMPI_MCA_io=^ompio` before `mpiexec/mpirun/srun/jsrun` will disable OMPIO and instead fall back to the older *ROMIO* MPI-I/O backend in OpenMPI.

7.5.3 Known Issues

Warning: Jul 23th, 2021 (HDFFV-11260): Collective HDF5 metadata reads (`OPENPMD_HDF5_COLLECTIVE_METADATA=ON`) broke in 1.10.5, falling back to individual metadata operations. HDF5 releases 1.10.4 and earlier are not affected; versions 1.10.9+, 1.12.2+ and 1.13.1+ fixed the issue.

Warning: The ROMIO backend of OpenMPI has a bug that leads to segmentation faults in combination with parallel HDF5 I/O with chunking enabled. This bug usually does not occur when using default configurations as OpenMPI uses the OMPIO component by default. The bug affects at least the entire OpenMPI 4.* release range and is currently set to be fixed for release 5.0 (release candidate available at the time of writing this).

7.5.4 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

CHAPTER

EIGHT

DATA ANALYSIS

8.1 openPMD-viewer

openPMD-viewer (documentation) is a Python package to access openPMD data.

It allows to:

- Quickly browse through the data, with a GUI-type interface in the Jupyter notebook
- Have access to the data numpy array, for more detailed analysis

8.1.1 Installation

openPMD-viewer can be installed via conda or pip:

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

```
python3 -m pip install openPMD-viewer openPMD-api
```

8.1.2 Usage

openPMD-viewer can be used either in simple Python scripts or in Jupyter. For interactive plots in Jupyter lab, add this "cell magic" to the first line of your notebook:

```
%matplotlib widget
```

and for Jupyter notebook use this instead:

```
%matplotlib notebook
```

If none of those work, e.g. because ipympl is not properly installed, you can as a last resort always try `%matplotlib inline` for non-interactive plots.

In both interactive and scripted usage, you can import openPMD-viewer, and load the data with the following commands:

```
from openpmd_viewer import OpenPMDTimeSeries
ts = OpenPMDTimeSeries('path/to/data/series/')
```

Note: If you are using the Jupyter notebook, then you can start a pre-filled notebook, which already contains the above lines, by typing in a terminal:

```
openPMD_notebook
```

When using the Jupyter notebook, you can quickly browse through the data by using the command:

```
ts.slider()
```

You can also access the particle and field data as numpy arrays with the methods `ts.get_field` and `ts.get_particle`. See the openPMD-viewer tutorials on read-the-docs for more info.

8.2 3D Visualization: ParaView

openPMD data can be visualized by ParaView, an open source visualization and analysis software. ParaView can be downloaded and installed from httpshttps://www.paraview.org. Use the latest version for best results.

8.2.1 Tutorials

ParaView is a powerful, general parallel rendering program. If this is your first time using ParaView, consider starting with a tutorial.

- https://www.paraview.org/Wiki/The_ParaView_Tutorial
- https://www.youtube.com/results?search_query=paraview+introduction
- https://www.youtube.com/results?search_query=paraview+tutorial

8.2.2 openPMD

openPMD files can be visualized with ParaView 5.9+, using 5.11+ is recommended. ParaView supports ADIOS2 and HDF5 files, as it implements against the Python bindings of openPMD-api.

For openPMD output to be recognized, create a small textfile with `.pmd` ending per data series, which can be opened with ParaView:

```
$ cat paraview.pmd
openpmd_%06T.bp
```

The file contains the same string as one would put in an openPMD `Series("....")` object.

Tip: When you first open ParaView, adjust its global `Settings` (Linux: under menu item `Edit`). `General` -> `Advanced` -> Search for `data` -> `Data Processing Options`. Check the box `Auto Convert Properties`.

This will simplify application of filters, e.g., contouring of components of vector fields, without first adding a calculator that extracts a single component or magnitude.

Warning: As of ParaView 5.11 and older, the axisLabel is not yet read for fields. See, e.g., WarpX issue 21162. Please apply rotation of, e.g., `0 -90 0` to mesh data where needed.

Warning: ParaView issue 21837: In order to visualize particle traces with the `Temporal Particles To Pathlines`, you need to apply the `Merge Blocks` filter first.

If you have multiple species, you may have to extract the species you want with `Extract Block` before applying `Merge Blocks`.

8.3 Pandas

The Python bindings of openPMD-api provide direct methods to load data into the Pandas data analysis ecosystem.

Pandas computes on the CPU, for GPU-accelerated data analysis see *RAPIDS*.

8.3.1 How to Install

Among many package managers, PyPI ships the latest packages of pandas:

```
python3 -m pip install -U pandas
```

8.3.2 Dataframes

The central Python API call to convert to openPMD particles to a Pandas dataframe is the `ParticleSpecies.to_df` method.

```python
import openpmd_api as io

s = io.Series("samples/git-sample/data%T.h5", io.Access.read_only)
electrons = s.iterations[400].particles["electrons"]

df = electrons.to_df()

type(df)  # pd.DataFrame
print(df)

# note: no series.flush() needed
```

One can also combine all iterations in a single dataframe like this:

```python
import pandas as pd

df = pd.concat(
    (
        s.iterations[i].particles["electrons"].to_df().assign(iteration=i)
        for i in s.iterations
    ),
    axis=0,
    ignore_index=True,
)

# like before but with a new column "iteration" and all particles
print(df)
```

8.3.3 openPMD to ASCII

Once converted to a Pandas dataframe, export of openPMD data to text is very simple. We generally do not recommend this because ASCII processing is slower, uses significantly more space on disk and has less precision than the binary data usually stored in openPMD data series. Nonetheless, in some cases and especially for small, human-readable data sets this can be helpful.

The central Pandas call for this is DataFrame.to_csv.

```
# creates a electrons.csv file
df.to_csv("electrons.csv", sep=",", header=True)
```

8.3.4 openPMD as SQL Database

Once converted to a Pandas dataframe, one can query and process openPMD data also with SQL syntax as provided by many databases.

A project that provides such syntax is for instance pandasql.

```
python3 -m pip install -U pandasql
```

or one can export into an SQL database.

8.3.5 Example

A detailed example script for particle and field analysis is documented under as `11_particle_dataframe.py` in our *examples*.

8.4 DASK

The Python bindings of openPMD-api provide direct methods to load data into the parallel, DASK data analysis ecosystem.

8.4.1 How to Install

Among many package managers, PyPI ships the latest packages of DASK:

```
python3 -m pip install -U dask
python3 -m pip install -U pyarrow
```

8.4.2 How to Use

The central Python API calls to convert to DASK datatypes are the `ParticleSpecies.to_dask` and `Record_Component.to_dask_array` methods.

```
s = io.Series("samples/git-sample/data%T.h5", io.Access.read_only)
electrons = s.iterations[400].particles["electrons"]

# the default schedulers are local/threaded. We can also use local
# "processes" or for multi-node "distributed", among others.
dask.config.set(scheduler='processes')

df = electrons.to_dask()
```

(continues on next page)

(continued from previous page)

```
type(df)  # ...

E = s.iterations[400].meshes["E"]
E_x = E["x"]
darr_x = E_x.to_dask_array()
type(darr_x)  # ...

# note: no series.flush() needed
```

The `to_dask_array` method will automatically set Dask array chunking based on the available chunks in the read data set. The default behavior can be overridden by passing an additional keyword argument `chunks`, see the dask.array.from_array documentation for more details. For example, to chunk only along the outermost axis in a 3D dataset using the default Dask array chunk size, call `to_dask_array(chunks={0:  'auto', 1:  -1, 2: -1})`.

8.4.3 Example

A detailed example script for particle and field analysis is documented under as `11_particle_dataframe.py` in our *examples*.

See a video of openPMD on DASK in action in pull request #963 (part of openPMD-api v0.14.0 and later).

8.5 RAPIDS

The Python bindings of openPMD-api enable easy loading into the GPU-accelerated RAPIDS.ai datascience & AI/ML ecosystem.

8.5.1 How to Install

Follow the official documentation to install RAPIDS.

```
# preparation
conda update -n base conda
conda install -n base conda-libmamba-solver
conda config --set solver libmamba

# install
conda create -n rapids -c rapidsai -c conda-forge -c nvidia rapids python cudatoolkit␣
↪openpmd-api pandas
conda activate rapids
```

8.5.2 Dataframes

The central Python API call to convert to openPMD particles to a cuDF dataframe is the `ParticleSpecies.to_df` method.

```
import openpmd_api as io
import cudf

s = io.Series("samples/git-sample/data%T.h5", io.Access.read_only)
electrons = s.iterations[400].particles["electrons"]

```

(continues on next page)

(continued from previous page)

```
cdf = cudf.from_pandas(electrons.to_df())

type(cdf)  # cudf.DataFrame
print(cdf)

# note: no series.flush() needed
```

One can also combine all iterations in a single dataframe like this:

```
cdf = cudf.concat(
    (
        cudf.from_pandas(s.iterations[i].particles["electrons"].to_df().
↪assign(iteration=i))
        for i in s.iterations
    ),
    axis=0,
    ignore_index=True,
)

# like before but with a new column "iteration" and all particles
print(cdf)
```

8.5.3 openPMD as SQL Database

Once converted to a dataframe, one can query and process openPMD data also with SQL syntax as provided by many databases.

A project that provides such syntax is for instance BlazingSQL (see the BlazingSQL install documentation).

```
import openpmd_api as io
from blazingsql import BlazingContext

s = io.Series("samples/git-sample/data%T.h5", io.Access.read_only)
electrons = s.iterations[400].particles["electrons"]

bc = BlazingContext(enable_progress_bar=True)
bc.create_table('electrons', electrons.to_df())

# all properties for electrons > 3e11 kg*m/s
bc.sql('SELECT * FROM electrons WHERE momentum_z > 3e11')

# selected properties
bc.sql('SELECT momentum_x, momentum_y, momentum_z, weighting FROM electrons WHERE␣
↪momentum_z > 3e11')
```

8.5.4 Example

A detailed example script for particle and field analysis is documented under as `11_particle_dataframe.py` in our *examples*.

8.6 Contributed

This page contains contributed projects and third party integrations to analyze openPMD data. See the openPMD-projects catalog for more community integrations.

8.6.1 3D Visualization: VisualPIC

openPMD data can be visualized with the domain-specific VisualPIC renderer. Please see the WarpX page for details.

8.6.2 3D Visualization: VisIt

openPMD **HDF5** data can be visualized with VisIt 3.1.0+. VisIt supports openPMD HDF5 files and requires to rename the files from `.h5` to `.opmd` to be automatically detected.

8.6.3 yt-project

openPMD **HDF5** data can be visualized with yt-project. Please see the yt documentation for details.

CHAPTER

NINE

DEVELOPMENT

9.1 Contribution Guide

9.1.1 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.

Maintainers organize prioritites and progress in the projects tab.

9.1.2 Style Guide

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

9.2 Repository Structure

9.2.1 Branches

- `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

9.2.2 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

9.3 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.

9.3.1 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:

```
/** Type of IO operation between logical and persistent data.
 */
OPENPMDAPI_EXPORT_ENUM_CLASS(Operation){
    CREATE_FILE,      CHECK_FILE,     OPEN_FILE,     CLOSE_FILE,
    DELETE_FILE,

    CREATE_PATH,      CLOSE_PATH,     OPEN_PATH,     DELETE_PATH,
    LIST_PATHS,

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

    DELETE_ATT,       WRITE_ATT,      READ_ATT,      LIST_ATTS,

    ADVANCE,
    AVAILABLE_CHUNKS, //!< Query chunks that can be loaded in a dataset
    DEREGISTER //!< Inform the backend that an object has been deleted.
}; // note: if you change the enum members here, please update
   // docs/source/dev/design.rst

namespace internal
{
    /*
     * The returned strings are compile-time constants, so no worries about
     * pointer validity.
     */
    OPENPMDAPI_EXPORT std::string operationAsString(Operation);
} // namespace internal

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 handlingi 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 extensibilty, 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*

9.3.2 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 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 …

9.3.3 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)

openPMD frontend classes are designed as handles that user code may copy an arbitrary amount of times, all copies sharing common resources. The internal shared state of such handles is stored as a single `shared_ptr`. This serves to separate data from implementation, demonstrated by the class hierarchy of `Series`:

- `SeriesData` contains all actual data members of `Series`, representing the resources shared between instances. Its copy/move constructors/assignment operators are deleted, thus pinning it at one memory location. It has no implementation.
- `Series` defines the interface of the class and is used by client code. Its only data member is a shared pointer to its associated instance of `SeriesData`. (This pointer-based design allows us to avoid a similar template-heavy design based on CRTP.)
- A non-owning instance of the `Series` class can be constructed on the fly from a `SeriesData` object by passing a `shared_ptr` with no-op destructor when constructing the `Series`. Care must be taken to not expose such an object to users as the type system does not distinguish between an owning and a non-owning `Series` object.
- Frontend classes that define the openPMD group hierarchy follow this same design (except for some few that do not define data members). These classes all inherit from `Attributable`. Their shared state inherits from `AttributableData`. A downside of this design is that the `shared_ptr` pointing to the internal state is redundantly stored for each level in the class hierarchy, `static_cast`-ed to the corresponding level. All these pointers reference the same internal object.

The class hierarchy of `Attributable` follows a similar design, with some differences due to its nature as a mixin class that is not instantiated directly:

The `Series` class is the entry point to the openPMD-api. An instance of this class always represents an entire series (of iterations), regardless of how this series is modeled in storage or transport (e.g. as multiple files, as a stream, as variables containing multiple snapshots of a dataset or as one file containing all iterations at once).

9.4 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.

9.4.1 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.

9.4.2 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`:

```cpp
/* 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
```

```cpp
/* 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`:

```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;
```

(continues on next page)

(continued from previous page)

```
            return std::make_shared< JSONIOHandler >(path, at);
    }
}
#endif

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.

9.4.3 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;
```

(continues on next page)

(continued from previous page)

```
    void deleteDataset(Writable*, Parameter< Operation::DELETE_DATASET > const&)␣
↪override;
    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_ATTS > &) override;
}
} // openPMD
```

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

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

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

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

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

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

namespace openPMD
{
struct JSONFilePosition : public AbstractFilePosition
{
```

(continues on next page)

(continued from previous page)

```
    JSONFilePosition(uint64_t id)
        : id{id}
    { }

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

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

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

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

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

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

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

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

9.5 Build Dependencies

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

9.5.1 Required

- CMake 3.15.0+
- C++17 capable compiler, e.g., g++ 7+, clang 7+, MSVC 19.15+, icpc 19+, icpx

9.5.2 Shipped internally

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

- Catch2 2.13.10+ (BSL-1.0)
- pybind11 2.11.1+ (new BSD)
- NLohmann-JSON 3.9.1+ (MIT)
- toml11 3.7.1+ (MIT)

9.5.3 Optional: I/O backends

- JSON
- HDF5 1.8.13+
- ADIOS2 2.7.0+

while those can be build either with or without:

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

9.5.4 Optional: language bindings

- Python:
  - Python 3.8 - 3.12
  - pybind11 2.11.1+
  - numpy 1.15+
  - mpi4py 2.1+ (optional, for MPI)
  - pandas 1.0+ (optional, for dataframes)
  - dask 2021+ (optional, for dask dataframes)
- CUDA C++ (optional, currently used only in tests)

9.5.5 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
```

9.6 Build Options

9.6.1 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
`openPMD_USE_MPI`	**AUTO**/ON/OFF	Parallel, Multi-Node I/O for clusters
`openPMD_USE_HDF5`	**AUTO**/ON/OFF	HDF5 backend (`.h5` files)
`openPMD_USE_ADIOS2`	**AUTO**/ON/OFF	ADIOS2 backend (`.bp` files in BP3, BP4 or higher)
`openPMD_USE_PYTHON`	**AUTO**/ON/OFF	Enable Python bindings
`openPMD_USE_INVASIVE_TES`	ON/**OFF**	Enable unit tests that modify source code [1]
`openPMD_USE_VERIFY`	**ON**/OFF	Enable internal VERIFY (assert) macro independent of build type [2]
`openPMD_INSTALL`	**ON**/OFF	Add installation targets
`openPMD_INSTALL_RPATH`	**ON**/OFF	Add RPATHs to installed binaries
`Python_EXECUTABLE`	(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

9.6.2 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
`openPMD_BUILD_SHARED_LIBS`	**ON**/OFF	Build the C++ API as shared library
`HDF5_USE_STATIC_LIBRARIES`	ON/**OFF**	Require static HDF5 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.

9.6.3 Debug

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

9.6.4 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
`openPMD_USE_INTERNAL_CATCH`	**ON**/OFF	No	Catch2	2.13.10+
`openPMD_USE_INTERNAL_PYBIND11`	**ON**/OFF	No	pybind11	2.11.1+
`openPMD_USE_INTERNAL_JSON`	**ON**/OFF	No	NLohmann-JSON	3.9.1+
`openPMD_USE_INTERNAL_TOML11`	**ON**/OFF	No	toml11	3.7.1+

9.6.5 Tests, Examples and Command Line Tools

By default, tests, examples and command line tools are built. In order to skip building those, pass `OFF` to these `cmake` options:

CMake Option	Values	Description
`openPMD_BUILD_TESTING`	**ON**/OFF	Build tests
`openPMD_BUILD_EXAMPLES`	**ON**/OFF	Build examples
`openPMD_BUILD_CLI_TOOLS`	**ON**/OFF	Build command-line tools
`openPMD_USE_CUDA_EXAMPLES`	ON/**OFF**	Use CUDA in examples

9.7 Linking to C++

The install will contain header files and libraries in the path set with the `-DCMAKE_INSTALL_PREFIX` option *from the previous section*.

9.7.1 CMake

If your project is using CMake for its build, one can conveniently use our provided `openPMDConfig.cmake` package, which is installed alongside the library.

First set the following environment hint if openPMD-api was *not* installed in a system path:

```
# optional: only needed if installed outside of system paths
export CMAKE_PREFIX_PATH=$HOME/somepath:$CMAKE_PREFIX_PATH
```

Use the following lines in your project's `CMakeLists.txt`:

```
# supports:                       COMPONENTS MPI NOMPI HDF5 ADIOS2
find_package(openPMD 0.15.0 CONFIG)

if(openPMD_FOUND)
    target_link_libraries(YourTarget PRIVATE openPMD::openPMD)
endif()
```

Alternatively, add the openPMD-api repository source directly to your project and use it via:

```
add_subdirectory("path/to/source/of/openPMD-api")

target_link_libraries(YourTarget PRIVATE openPMD::openPMD)
```

For development workflows, you can even automatically download and build openPMD-api from within a depending CMake project. Just replace the `add_subdirectory` call with:

```
include(FetchContent)
set(CMAKE_POLICY_DEFAULT_CMP0077 NEW)
set(openPMD_BUILD_CLI_TOOLS OFF)
set(openPMD_BUILD_EXAMPLES OFF)
set(openPMD_BUILD_TESTING OFF)
set(openPMD_BUILD_SHARED_LIBS OFF)  # precedence over BUILD_SHARED_LIBS if needed
set(openPMD_INSTALL OFF)            # or instead use:
# set(openPMD_INSTALL ${BUILD_SHARED_LIBS})  # only install if used as a shared␣
↪library
set(openPMD_USE_PYTHON OFF)
FetchContent_Declare(openPMD
  GIT_REPOSITORY "https://github.com/openPMD/openPMD-api.git"
  GIT_TAG        "0.15.0")
FetchContent_MakeAvailable(openPMD)
```

9.7.2 Manually

If your (Linux/OSX) project is build by calling the compiler directly or uses a manually written `Makefile`, consider using our `openPMD.pc` helper file for `pkg-config`, which are installed alongside the library.

First set the following environment hint if openPMD-api was *not* installed in a system path:

```
# optional: only needed if installed outside of system paths
export PKG_CONFIG_PATH=$HOME/somepath/lib/pkgconfig:$PKG_CONFIG_PATH
```

Additional linker and compiler flags for your project are available via:

```
# switch to check if openPMD-api was build as static library
# (via BUILD_SHARED_LIBS=OFF) or as shared library (default)
if [ "$(pkg-config --variable=static openPMD)" == "true" ]
then
    pkg-config --libs --static openPMD
    # -L/usr/local/lib -L/usr/lib/x86_64-linux-gnu/openmpi/lib -lopenPMD -pthread /
↪usr/lib/libmpi.so -pthread /usr/lib/x86_64-linux-gnu/openmpi/lib/libmpi_cxx.so /usr/
↪lib/libmpi.so /usr/lib/x86_64-linux-gnu/hdf5/openmpi/libhdf5.so /usr/lib/x86_64-
↪linux-gnu/libsz.so /usr/lib/x86_64-linux-gnu/libz.so /usr/lib/x86_64-linux-gnu/
↪libdl.so /usr/lib/x86_64-linux-gnu/libm.so -pthread /usr/lib/libmpi.so -pthread /
↪usr/lib/x86_64-linux-gnu/openmpi/lib/libmpi_cxx.so /usr/lib/libmpi.so
else
    pkg-config --libs openPMD
    # -L${HOME}/somepath/lib -lopenPMD
fi

pkg-config --cflags openPMD
# -I${HOME}/somepath/include
```

9.8 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.

9.8.1 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

# 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
```

9.8.2 Useful Links

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

CHAPTER

TEN

MAINTENANCE

10.1 Release Channels

10.1.1 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.

10.1.2 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

10.1.3 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

10.1.4 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), macOS (10.9+) and Windows.

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

10.1.5 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.

10.1.6 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.