# CONTENTS

1 Contact us 3  
1.1 Code of Conduct 3  
1.2 Acknowledge WarpX 4  

2 Installation 7  
2.1 Users 7  
2.2 Developers 9  
2.3 HPC 19  

3 Usage 63  
3.1 Run WarpX 63  
3.2 Input Parameters 64  
3.3 Python (PICMI) 103  
3.4 Examples 104  
3.5 In-Depth: PWFA 106  
3.6 Workflows 110  
3.7 FAQ 123  

4 Data Analysis 125  
4.1 Output formats 125  
4.2 yt-project 142  
4.3 openPMD-viewer 148  
4.4 openPMD-api 149  
4.5 3D Visualization: ParaView 149  
4.6 3D Visualization: VisIt 149  
4.7 VisualPIC 151  
4.8 PICViewer 151  
4.9 Back-Transformed Diagnostics 153  
4.10 Reduced diagnostics 154  

5 Theory 157  
5.1 Introduction 157  
5.2 Particle-in-Cell Method 160  
5.3 Filtering 168  
5.4 Mesh refinement 170  
5.5 Boundary conditions 174  
5.6 Moving window and optimal Lorentz boosted frame 177  
5.7 Inputs and Outputs 184  

6 Development 189  
6.1 Contribute to WarpX 189
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Workflows</td>
<td>193</td>
</tr>
<tr>
<td>6.3</td>
<td>Implementation Details</td>
<td>200</td>
</tr>
<tr>
<td>6.4</td>
<td>C++ Objects &amp; Functions</td>
<td>224</td>
</tr>
<tr>
<td>6.5</td>
<td>GNUmake Build System (Legacy)</td>
<td>224</td>
</tr>
<tr>
<td>6.6</td>
<td>FAQ</td>
<td>232</td>
</tr>
<tr>
<td>7</td>
<td>Maintenance</td>
<td>235</td>
</tr>
<tr>
<td>7.1</td>
<td>Dependencies &amp; Releases</td>
<td>235</td>
</tr>
<tr>
<td>7.2</td>
<td>Automated performance tests</td>
<td>236</td>
</tr>
<tr>
<td>8</td>
<td>Epilogue</td>
<td>239</td>
</tr>
<tr>
<td>8.1</td>
<td>Glossary</td>
<td>239</td>
</tr>
<tr>
<td>8.2</td>
<td>Funding and Acknowledgements</td>
<td>241</td>
</tr>
</tbody>
</table>

Index 243
WarpX is an advanced *electromagnetic Particle-In-Cell* code.

It supports many features including:

- Perfectly-Matched Layers (PML)
- Boosted-frame simulations
- Mesh refinement

For details on the algorithms that WarpX implements, see the *theory section*.

WarpX is a *highly-parallel and highly-optimized code*, which can run on GPUs and multi-core CPUs, and includes load balancing capabilities. In addition, WarpX is also a *multi-platform code* and runs on Linux, macOS and Windows.
If you are starting using WarpX, or if you have a user question, please pop in our Gitter chat room and get in touch with the community.

The WarpX GitHub repo is the main communication platform. Have a look at the action icons on the top right of the web page: feel free to watch the repo if you want to receive updates, or to star the repo to support the project. For bug reports or to request new features, you can also open a new issue.

We also have a discussion page on which you can find already answered questions, add new questions, get help with installation procedures, discuss ideas or share comments.

1.1 Code of Conduct

1.1.1 Our Pledge

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

1.1.2 Our Standards

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

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

Examples of unacceptable behavior by participants include:

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

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

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

1.1.4 Scope

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

1.1.5 Enforcement

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

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

1.1.6 Attribution

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

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

1.2 Acknowledge WarpX

Please acknowledge the role that WarpX played in your research.

1.2.1 In presentations

For your presentations, you can find WarpX slides here. Several flavors are available:

- full slide
- half-slide (portrait or landscape format)
- small inset.

Feel free to use the one that fits into your presentation and adequately acknowledges the part that WarpX played in your research.
1.2.2 In publications

Please add the following sentence to your publications, it helps contributors keep in touch with the community and promote the project.

Plain text:

This research used the open-source particle-in-cell code WarpX https://github.com/ECP-WarpX/WarpX, primarily funded by the US DOE Exascale Computing Project. Primary WarpX contributors are with LBNL, LLNL, CEA-LIDYL, SLAC, DESY, CERN, and Modern Electron. We acknowledge all WarpX contributors.

Latex:

\usepackage{hyperref}
This research used the open-source particle-in-cell code WarpX \url{https://github.com/ECP-WarpX/WarpX}, primarily funded by the US DOE Exascale Computing Project. Primary WarpX contributors are with LBNL, LLNL, CEA-LIDYL, SLAC, DESY, CERN, and Modern Electron. We acknowledge all WarpX contributors.

1.2.3 Latest WarpX reference

If your project leads to a scientific publication, please consider citing the paper below.


Prior WarpX references

If your project uses the specific algorithms, please consider citing the respective publications in addition.


2.1 Users

Our community is here to help. Please report installation problems in case you should get stuck. Choose one of the installation methods below to get started:

2.1.1 HPC Systems

If want to use WarpX on a specific high-performance computing (HPC) systems, jump directly to our HPC system-specific documentation.

2.1.2 Using the Conda Package

A package for WarpX is available via the Conda package manager.

```
conda create -n warpx -c conda-forge warpx
conda activate warpx
```

Note: the warpx conda package does not yet provide GPU support.

2.1.3 Using the Spack Package

Packages for WarpX are available via the Spack package manager. The package warpx installs executables and the package py-warpx includes Python bindings, i.e. PICMI.

```
# optional: -mpi ^warpx dims=2 compute=cuda
spack install py-warpx
spack load py-warpx
```

See spack info warpx or spack info py-warpx and the official Spack tutorial for more information.
2.1.4 Using the PyPI Package

Given that you have the WarpX dependencies installed, you can use pip to install WarpX with PICMI from source:

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

python3 -m pip wheel -v git+https://github.com/ECP-WarpX/WarpX.git
# optional: --user
python3 -m pip install *whl
```

In the future, will publish pre-compiled binary packages on PyPI for faster installs. (Consider using conda in the meantime.)

2.1.5 Using the Brew Package

Note: Coming soon.

2.1.6 From Source with CMake

After installing the WarpX dependencies, you can also install WarpX from source with CMake:

```
# get the source code
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
cd $HOME/src/warpx

# configure
cmake -S . -B build

# optional: change configuration
ccmake build

# compile
# on Windows: --config RelWithDebInfo
cmake --build build -j 4

# executables for WarpX are now in build/bin/
```

We document the details in the developer installation.
2.1.7 Tips for macOS Users

**Tip:** Before getting started with package managers, please check what you manually installed in /usr/local. If you find entries in `bin/`, `lib/` et al. that look like you manually installed MPI, HDF5 or other software in the past, then remove those files first.

If you find software such as MPI in the same directories that are shown as symbolic links then it is likely you brew installed software before. If you are trying another package manager than brew, run `brew unlink...` on such packages first to avoid software incompatibilities.

See also: A. Huebl, Working With Multiple Package Managers, Collegeville Workshop (CW20), 2020

2.2 Developers

CMake is our primary build system. If you are new to CMake, this short tutorial from the HEP Software foundation is the perfect place to get started. If you just want to use CMake to build the project, jump into sections 1. Introduction, 2. Building with CMake and 9. Finding Packages.

2.2.1 Dependencies

Before you start, you will need a copy of the WarpX source code:

```bash
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
cd $HOME/src/warpx
```

WarpX depends on popular third party software.
- On your development machine, follow the instructions here.
- If you are on an HPC machine, follow the instructions here.

**Dependencies**

WarpX depends on the following popular third party software. Please see installation instructions below.
- a mature C++17 compiler, e.g., GCC 7, Clang 7, NVCC 11.0, MSVC 19.15 or newer
- CMake 3.18.0+
- Git 2.18+
- AMReX: we automatically download and compile a copy of AMReX
- PICSAR: we automatically download and compile a copy of PICSAR

Optional dependencies include:
- MPI 3.0+: for multi-node and/or multi-GPU execution
- CUDA Toolkit 11.0+: for Nvidia GPU support (see matching host-compilers)
- OpenMP 3.1+: for threaded CPU execution (currently not fully accelerated)
- FFTW3: for spectral solver (PSATD) support
  - also needs the `pkg-config` tool on Unix
• **BLAS++ and LAPACK++**: for spectral solver (PSATD) support in RZ geometry
• **Boost 1.66.0+**: for QED lookup tables generation support
• **openPMD-api 0.14.2+**: we automatically download and compile a copy of openPMD-api for openPMD I/O support
  – see optional I/O backends
• **CCache**: to speed up rebuilds (For CUDA support, needs version 3.7.9+ and 4.2+ is recommended)
• **Ninja**: for faster parallel compiles
• **Python 3.6+**
  – mpi4py
  – numpy
  – periodictable
  – picmistandard
  – see our requirements.txt file for compatible versions

**Install**

Pick *one* of the installation methods below to install all dependencies for WarpX development in a consistent manner.

**Spack (macOS/Linux)**

```
spack env create warpx-dev
spack env activate warpx-dev

spack add adios2       # for openPMD
spack add blaspp       # for PSATD in RZ
spack add ccache       
spack add cmake        
spack add fftw         # for PSATD
spack add hdf5         # for openPMD
spack add lapackpp     # for PSATD in RZ
spack add mpi          
spack add openpmd-api  # for openPMD
spack add pkgconfig    # for fftw

# OpenMP support on macOS
[[ $OSTYPE == 'darwin'* ]] && spack add llvm-openmp

# optional:
# spack add python
# spack add py-pip
# spack add cuda

spack install
```

In new terminal sessions, re-activate the environment with `spack env activate warpx-dev` again.
If you also want to run runtime tests and added Python (spack add python and spack add py-pip) above, install also these additional Python packages in the active Spack environment:

```bash
python3 -m pip install matplotlib yt scipy pandas numpy openpmd-api virtualenv
```

If you want to run the `./run_test.sh` test script, which uses our legacy GNUmake build system, you need to set the following environment hints after `spack env activate warpx-dev` for dependent software:

```bash
export FFTW_HOME=${SPACK_ENV}/.spack-env/view
export BLASPP_HOME=${SPACK_ENV}/.spack-env/view
export LAPACKPP_HOME=${SPACK_ENV}/.spack-env/view
```

**Brew (macOS/Linux)**

```bash
brew update
brew tap openpmd/openpmd
brew install adios2       # for openPMD
brew install ccache
brew install cmake
brew install fftw          # for PSATD
brew install git
brew install hdf5-mpi     # for openPMD
brew install libomp
brew install pkg-config    # for fftw
brew install open-mpi
brew install openblas     # for PSATD in RZ
brew install openpmd-api  # for openPMD
```

If you also want to compile with PSATD in RZ, you need to manually install BLAS++ and LAPACK++:

```bash
sudo mkdir -p /usr/local/bin/
sudo curl -L -o /usr/local/bin/cmake-easyinstall https://git.io/JvLxY
sudo chmod a+x /usr/local/bin/cmake-easyinstall

cmake-easyinstall --prefix=/usr/local git+https://bitbucket.org/icl/blaspp.git \
    -Duse_openmp=OFF -Dbuild_tests=OFF -DCMAKE_VERBOSE_MAKEFILE=ON
cmake-easyinstall --prefix=/usr/local git+https://bitbucket.org/icl/lapackpp.git \
    -Duse_cmake_find_lapack=ON -Dbuild_tests=OFF -DCMAKE_VERBOSE_MAKEFILE=ON
```

**Conda (Linux/macOS/Windows)**

Without MPI:

```bash
conda create -n warpx-dev -c conda-forge blaspp ccache cmake compilers git lapackpp␣˓→openpmd-api python numpy pandas scipy fftw pkg-config matplotlib mamba ninja pip␣˓→virtualenv
source activate warpx-dev
# compile WarpX with -DWarpX_MPI=OFF
```

With MPI (only Linux/macOS):

2.2. Developers
conda create -n warpx-dev -c conda-forge blaspp ccache cmake compilers git lapackpp
   "openpmd-api=*=mpi_openmpi" python numpy pandas scipy yt "fftw=*=mpi_openmpi" pkg-config matplotlib mamba ninja openmpi pip virtualenv
source activate warpx-dev

For legacy GNUmake builds, after each `source activate warpx-dev`, you also need to set:

```bash
export FFTW_HOME=${CONDA_PREFIX}
export BLASPP_HOME=${CONDA_PREFIX}
export LAPACKPP_HOME=${CONDA_PREFIX}
```

### Apt (Debian/Ubuntu)

```bash
sudo apt update
sudo apt install build-essential ccache cmake g++ git libfftw3-mpi-dev libfftw3-dev-
   libhdf5-openmpi-dev libopenmpi-dev pkg-config python3 python3-matplotlib python3-
   pandas python3-pip python3-scipy python3-venv

# optional:
# for CUDA, either install
# https://developer.nvidia.com/cuda-downloads (preferred)
# or, if your Debian/Ubuntu is new enough, use the packages
# sudo apt install nvidia-cuda-dev libcub-dev
```

#### 2.2.2 Compile

From the base of the WarpX source directory, execute:

```bash
# find dependencies & configure
# see additional options below, e.g.
# -DCMAKE_INSTALL_PREFIX=$HOME/sw/warpx
cmake -S . -B build

# compile, here we use four threads
 cmake --build build -j 4
```

**That's it!** A 3D WarpX binary is now in `build/bin/` and *can be run* with a 3D example inputs file. Most people execute the binary directly or copy it out.

If you want to install the executables in a programmatic way, run this:

```bash
# for default install paths, you will need administrator rights, e.g. with sudo:
 cmake --build build --target install
```

You can inspect and modify build options after running `cmake -S . -B build` with either

```
cmake build
```

or by adding arguments with `-D<OPTION>=<VALUE>` to the first CMake call. For example, this builds WarpX in 2D geometry and enables Nvidia GPU (CUDA) support:
cmake -S . -B build -DWarpX_DIMS=2 -DWarpX_COMPUTE=CUDA

2.2.3 Build Options

<table>
<thead>
<tr>
<th>CMake Option</th>
<th>Default &amp; Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_BUILD_TYPE</td>
<td>RelWithDebInfo/Release/Debug</td>
<td>Type of build, symbols &amp; optimizations</td>
</tr>
<tr>
<td>CMAKE_INSTALL_PREFIX</td>
<td>system-dependent path</td>
<td>Install path prefix</td>
</tr>
<tr>
<td>CMAKE_VERBOSE_MAKEFILE</td>
<td>ON/OFF</td>
<td>Print all compiler commands to the terminal during build</td>
</tr>
<tr>
<td>PYINSTALLOPTIONS</td>
<td>Additional options for pip install, e.g., --user</td>
<td></td>
</tr>
<tr>
<td>WarpX_APP</td>
<td>ON/OFF</td>
<td>Build the WarpX executable application</td>
</tr>
<tr>
<td>WarpX_ASCENT</td>
<td>ON/OFF</td>
<td>Ascent in situ visualization</td>
</tr>
<tr>
<td>WarpX_COMPUTE</td>
<td>NOACC/OMP/CUDA/SYCL/hip</td>
<td>On-node, accelerated computing backend</td>
</tr>
<tr>
<td>WarpX_DIMS</td>
<td>3/2/1/RZ</td>
<td>Simulation dimensionality</td>
</tr>
<tr>
<td>WarpX_EB</td>
<td>ON/OFF</td>
<td>Embedded boundary support (not supported in RZ yet)</td>
</tr>
<tr>
<td>WarpX_GPU_CLOCK</td>
<td>ON/OFF</td>
<td>Add GPU kernel timers (cost function, +4 registers/kernel)</td>
</tr>
<tr>
<td>WarpX_IPO</td>
<td>ON/OFF</td>
<td>Compile WarpX with interprocedural optimization (aka LTO)</td>
</tr>
<tr>
<td>WarpX_LIB</td>
<td>ON/OFF</td>
<td>Build WarpX as a shared library, e.g., for PICMI Python</td>
</tr>
<tr>
<td>WarpX_MPI</td>
<td>ON/OFF</td>
<td>Multi-node support (message-passing)</td>
</tr>
<tr>
<td>WarpX_MPI_THREAD_MULTIPLE</td>
<td>ON/OFF</td>
<td>MPI thread-multiple support, i.e. for async_io</td>
</tr>
<tr>
<td>WarpX_OPENPMD</td>
<td>ON/OFF</td>
<td>openPMD I/O (HDF5, ADIOS)</td>
</tr>
<tr>
<td>WarpX_PRECISION</td>
<td>SINGLE/Doubles</td>
<td>Floating point precision (single/double)</td>
</tr>
<tr>
<td>WarpX_PSATD</td>
<td>ON/OFF</td>
<td>Spectral solver</td>
</tr>
<tr>
<td>WarpX_QED</td>
<td>ON/OFF</td>
<td>QED support (requires PICSAR)</td>
</tr>
<tr>
<td>WarpX_QED_TABLE_GEN</td>
<td>ON/OFF</td>
<td>QED table generation support (requires PICSAR and Boost)</td>
</tr>
<tr>
<td>WarpX_SENSEI</td>
<td>ON/OFF</td>
<td>SENSEI in situ visualization</td>
</tr>
</tbody>
</table>

WarpX can be configured in further detail with options from AMReX, which are documented in the AMReX manual:

- general AMReX build options
- GPU-specific options.

Developers might be interested in additional options that control dependencies of WarpX. By default, the most important dependencies of WarpX are automatically downloaded for convenience:
### CMake Option

<table>
<thead>
<tr>
<th>CMake Option</th>
<th>Default &amp; Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCACHE_PROGRAM</td>
<td>First found ccache executable.</td>
<td>Set to <code>-DCCACHE_PROGRAM=NO</code> to disable CCache.</td>
</tr>
<tr>
<td>AMReX_CUDA_PTX_VERBOSE</td>
<td>ON/OFF</td>
<td>Print CUDA code generation statistics from ptxas.</td>
</tr>
<tr>
<td>WarpX_amrex_src</td>
<td>None</td>
<td>Path to AMReX source directory (preferred if set)</td>
</tr>
<tr>
<td>WarpX_amrex_repo</td>
<td><code>https://github.com/AMReX-Codes/amrex.git</code></td>
<td>Repository URI to pull and build AMReX from</td>
</tr>
<tr>
<td>WarpX_amrex_branch</td>
<td><code>we set and maintain a compatible commit</code></td>
<td>Repository branch for <code>WarpX_amrex_repo</code></td>
</tr>
<tr>
<td>WarpX_amrex_internal</td>
<td>ON/OFF</td>
<td>Needs a pre-installed AMReX library if set to OFF</td>
</tr>
<tr>
<td>WarpX_openpmd_src</td>
<td>None</td>
<td>Path to openPMD-api source directory (preferred if set)</td>
</tr>
<tr>
<td>WarpX_openpmd_repo</td>
<td><code>https://github.com/openPMD/openPMD-api.git</code></td>
<td>Repository URI to pull and build openPMD-api from</td>
</tr>
<tr>
<td>WarpX_openpmd_branch</td>
<td>0.14.3</td>
<td>Repository branch for <code>WarpX_openpmd_repo</code></td>
</tr>
<tr>
<td>WarpX_openpmd_internal</td>
<td>ON/OFF</td>
<td>Needs a pre-installed openPMD-api library if set to OFF</td>
</tr>
<tr>
<td>WarpX_picsar_src</td>
<td>None</td>
<td>Path to PICSAR source directory (preferred if set)</td>
</tr>
<tr>
<td>WarpX_picsar_repo</td>
<td><code>https://github.com/ECP-WarpX/picsar.git</code></td>
<td>Repository URI to pull and build PICSAR from</td>
</tr>
<tr>
<td>WarpX_picsar_branch</td>
<td><code>we set and maintain a compatible commit</code></td>
<td>Repository branch for <code>WarpX_picsar_repo</code></td>
</tr>
<tr>
<td>WarpX_picsar_internal</td>
<td>ON/OFF</td>
<td>Needs a pre-installed PICSAR library if set to OFF</td>
</tr>
</tbody>
</table>

For example, one can also build against a local AMReX copy. Assuming AMReX’s source is located in `$HOME/src/amrex`, add the cmake argument `-DWarpX_amrex_src=$HOME/src/amrex`. Relative paths are also supported, e.g. `-DWarpX_amrex_src=../amrex`.

Or build against an AMReX feature branch of a colleague. Assuming your colleague pushed AMReX to `https://github.com/WeiQuanZhang/amrex/` in a branch `new-feature` then pass to cmake the arguments: `-DWarpX_amrex_repo=https://github.com/WeiQuanZhang/amrex.git` `-DWarpX_amrex_branch=new-feature`.

You can speed up the install further if you pre-install these dependencies, e.g. with a package manager. Set `-DWarpX_<dependency-name>_internal=OFF` and add installation prefix of the dependency to the environment variable `CMAKE_PREFIX_PATH`. Please see the *introduction to CMake* if this sounds new to you.

If you re-compile often, consider installing the Ninja build system. Pass `-G Ninja` to the CMake configuration call to speed up parallel compiles.
2.2.4 Configure your compiler

If you don’t want to use your default compiler, you can set the following environment variables. For example, using a Clang/LLVM:

```bash
export CC=$(which clang)
export CXX=$(which clang++)
```

If you also want to select a CUDA compiler:

```bash
export CUDACXX=$(which nvcc)
export CUDAHOSTCXX=$(which clang++)
```

We also support adding additional compiler flags via environment variables such as CXXFLAGS/LDFLAGS:

```bash
# example: treat all compiler warnings as errors
export CXXFLAGS="-Werror"
```

**Note:** Please clean your build directory with `rm -rf build/` after changing the compiler. Now call `cmake -S . -B build` (+ further options) again to re-initialize the build configuration.

2.2.5 Run

An executable WarpX binary with the current compile-time options encoded in its file name will be created in `build/bin/`. Note that you need separate binaries to run 1D, 2D, 3D, and RZ geometry inputs scripts. Additionally, a symbolic link named `warpx` can be found in that directory, which points to the last built WarpX executable.

More details on running simulations are in the section Run WarpX. Alternatively, read on and also build our PICMI Python interface.

2.2.6 PICMI Python Bindings

**Note:** Preparation: make sure you work with up-to-date Python tooling.

```bash
python3 -m pip install -U pip setuptools wheel
```

For PICMI Python bindings, configure WarpX to produce a library and call our `pip_install` CMake target:

```bash
# find dependencies & configure
cmake -S . -B build -DWarpX_LIB=ON

# build and then call "python3 -m pip install ..."
cmake --build build --target pip_install -j 4
```

That’s it! You can now run a first 3D PICMI script from our examples.

Developers could now change the WarpX source code and then call the build line again to refresh the Python installation.
Note: These commands build one -DWarpX_DIMS=... dimensionality (default: 3) at a time. If your build/lib*/ directory contains previously built libwarpx* libraries, then --target pip_install picks them up as well. A new call to cmake --build build ... will only rebuild one dimensionality, as set via WarpX_DIMS.

If you like to build a WarpX Python package that supports all dimensionalities, you can run this:

```bash
for d in 1 2 3 RZ; do
cmake -S . -B build -DWarpX_DIMS=$d -DWarpX_LIB=ON
cmake --build build -j 4
done
cmake --build build --target pip_install
```

Tip: If you do not develop with a user-level package manager, e.g., because you rely on a HPC system’s environment modules, then consider to set up a virtual environment via Python venv. Otherwise, without a virtual environment, you likely need to add the CMake option -DPYINSTALLOPTIONS="--user".

2.2.7 Python Bindings (Package Management)

This section is relevant for Python package management, mainly for maintainers or people that rather like to interact only with pip.

One can build and install pywarpx from the root of the WarpX source tree:

```bash
python3 -m pip wheel -v .
python3 -m pip install pywarpx*whl
```

This will call the CMake logic above implicitly. Using this workflow has the advantage that it can build and package up multiple libraries with varying WarpX_DIMS into one pywarpx package.

Environment variables can be used to control the build step:
<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default &amp; Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WARPX_COMPUTE</td>
<td>NOACC/OMP/CUDA/SYCL</td>
<td>On-node, accelerated computing backend</td>
</tr>
<tr>
<td>WARPX_DIMS</td>
<td>&quot;1;2;3;RZ&quot;</td>
<td>Simulation dimensionalities (semicolon-separated list)</td>
</tr>
<tr>
<td>WARPX_EB</td>
<td>ON/OFF</td>
<td>Embedded boundary support (not supported in RZ yet)</td>
</tr>
<tr>
<td>WARPX_MPI</td>
<td>ON/OFF</td>
<td>Multi-node support (message-passing)</td>
</tr>
<tr>
<td>WARPX_OPENPMD</td>
<td>ON/OFF</td>
<td>openPMD I/O (HDF5, ADIOS)</td>
</tr>
<tr>
<td>WARPX_PRECISION</td>
<td>SINGLE/Doubles</td>
<td>Floating point precision (single/double)</td>
</tr>
<tr>
<td>WARPX_PSATD</td>
<td>ON/OFF</td>
<td>Spectral solver</td>
</tr>
<tr>
<td>WARPX_QED</td>
<td>ON/OFF</td>
<td>PICSAR QED (requires PICSAR)</td>
</tr>
<tr>
<td>WARPX_QED_TABLE_GEN</td>
<td>ON/OFF</td>
<td>QED table generation (requires PICSAR and Boost)</td>
</tr>
<tr>
<td>BUILD_PARALLEL</td>
<td>2</td>
<td>Number of threads to use for parallel builds</td>
</tr>
<tr>
<td>BUILD_SHARED_LIBS</td>
<td>ON/OFF</td>
<td>Build shared libraries for dependencies</td>
</tr>
<tr>
<td>HDF5_USE_STATIC_LIBRARIES</td>
<td>ON/OFF</td>
<td>Prefer static libraries for HDF5 dependency (openPMD)</td>
</tr>
<tr>
<td>ADIOS_USE_STATIC_LIBS</td>
<td>ON/OFF</td>
<td>Prefer static libraries for ADIOS1 dependency (openPMD)</td>
</tr>
<tr>
<td>WARPX_AMREX_SRC</td>
<td>None</td>
<td>Absolute path to AMReX source directory (preferred if set)</td>
</tr>
<tr>
<td>WARPX_AMREX_REPO</td>
<td>None (uses cmake default)</td>
<td>Repository URI to pull and build AMReX from</td>
</tr>
<tr>
<td>WARPX_AMREX_BRANCH</td>
<td>None (uses cmake default)</td>
<td>Repository branch for WARPX_AMREX_REPO</td>
</tr>
<tr>
<td>WARPX_AMREX_INTERNAL</td>
<td>ON/OFF</td>
<td>Needs a pre-installed AMReX library if set to OFF</td>
</tr>
<tr>
<td>WARPX_OPENPMD_SRC</td>
<td>None</td>
<td>Absolute path to openPMD-api source directory (preferred if set)</td>
</tr>
<tr>
<td>WARPX_OPENPMD_INTERNAL</td>
<td>ON/OFF</td>
<td>Needs a pre-installed openPMD-api source directory (preferred if set)</td>
</tr>
<tr>
<td>WARPX_PICSAR_SRC</td>
<td>None</td>
<td>Absolute path to PICSAR source directory (preferred if set)</td>
</tr>
<tr>
<td>WARPX_PICSAR_INTERNAL</td>
<td>ON/OFF</td>
<td>Needs a pre-installed PICSAR library if set to OFF</td>
</tr>
<tr>
<td>WARPX_CCACHE_PROGRAM</td>
<td>First found ccache executable.</td>
<td>Set to NO to disable CCLine.</td>
</tr>
<tr>
<td>PYWARPX_LIB_DIR</td>
<td>None</td>
<td>If set, search for pre-built WarpX C++ libraries (see below)</td>
</tr>
</tbody>
</table>

Note that we currently change the WARPX_MPI default intentionally to OFF, to simplify a first install from source.

Some hints and workflows follow. Developers, that want to test a change of the source code but did not change the pywarpx version number, can force a reinstall via:

```bash
python3 -m pip install --force-reinstall --no-deps -v .
```

Some Developers like to code directly against a local copy of AMReX, changing both code-bases at a time:

```bash
WARPX_AMREX_SRC=$PWD/../amrex python3 -m pip install --force-reinstall --no-deps -v .
```

Additional environment control as common for CMake (see above) can be set as well, e.g. CC, CXX, and CMAKE_PREFIX_PATH hints. So another sophisticated example might be: use Clang as the compiler, build with local source copies of PICSAR and AMReX, support the PSATD solver, MPI and openPMD, hint a parallel HDF5 installation in $HOME/sw/hdf5-parallel-1.10.4, and only build 3D geometry:

```bash
CC=$(which clang) CXX=$(which clang++) WARPX_AMREX_SRC=$PWD/../amrex WARPX_PICSAR_SRC=.../piccar WARPX_PSATD=ON WARPX_MPI=ON WARPX_DIMS=3 CMAKE_PREFIX_PATH=$HOME/sw/...hdf5-parallel-1.10.4:$CMAKE_PREFIX_PATH python3 -m pip install --force-reinstall --no-deps -v .
```

## 2.2. Developers

17
Here we wrote this all in one line, but one can also set all environment variables in a development environment and keep the pip call nice and short as in the beginning. Note that you need to use absolute paths for external source trees, because pip builds in a temporary directory, e.g. `export WARPX_AMREX_SRC=$HOME/src/amrex`.

The Python library `pywarpx` can also be created by pre-building WarpX into one or more shared libraries externally. For example, a package manager might split WarpX into a C++ package and a Python package. If the C++ libraries are already pre-compiled, we can pick them up in the Python build step instead of compiling them again:

```
# build WarpX executables and libraries
for d in 1 2 3 RZ; do
cmake -S . -B build -DWarpX_DIMS=$d -DWarpX_LIB=ON
cmake --build build -j 4
done

# Python package
PYWARPX_LIB_DIR=$PWD/build/lib python3 -m pip wheel .

# install
python3 -m pip install pywarpx-*whl
```

WarpX release managers might also want to generate a self-contained source package that can be distributed to exotic architectures:

```
python setup.py sdist --dist-dir .
python3 -m pip wheel -v pywarpx-*tar.gz
python3 -m pip install *whl
```

The above steps can also be executed in one go to build from source on a machine:

```
python3 setup.py sdist --dist-dir .
python3 -m pip install -- --dist-dir .
```

Last but not least, you can uninstall `pywarpx` as usual with:

```
python3 -m pip uninstall pywarpx
```
2.3 HPC

On selected high-performance computing (HPC) systems, WarpX has documented or even pre-build installation routines. Follow the guide here instead of the generic installation routines for optimal stability and best performance.

2.3.1 warpx.profile

Use a warpx.profile file to set up your software environment without colliding with other software. Ideally, store that file directly in your $HOME/ and source it after connecting to the machine:

```
source $HOME/warpx.profile
```

We list example warpx.profile files below, which can be used to set up WarpX on various HPC systems.

2.3.2 HPC Systems

**Cori (NERSC)**

The Cori cluster is located at NERSC.

If you are new to this system, please see the following resources:

- GPU nodes
- Cori user guide
- Batch system: Slurm
- Jupyter service
- Production directories:
  - $SCRATCH: per-user production directory (20TB)
  - /global/cscratch1/sd/m3239: shared production directory for users in the project m3239 (50TB)
  - /global/cfs/cdirs/m3239/: community file system for users in the project m3239 (100TB)

**Installation**

Use the following commands to download the WarpX source code and switch to the correct branch:

```
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

**KNL**

We use the following modules and environments on the system ($HOME/knl_warpx.profile).

2.3. HPC
And install ADIOS2, BLAS++ and LAPACK++:

```bash
source $HOME/knl_warpx.profile

# c-blosc (I/O compression)
git clone -b v1.21.1 https://github.com/Blosc/c-blosc.git src/c-blosc
rm -rf src/c-blosc-knl-build
cmake -S src/c-blosc -B src/c-blosc-knl-build -DBUILD_TESTS=OFF -DBUILD_BENCHMARKS=OFF -DDEACTIVATE_AVX2=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/c-blosc-1.12.1-knl-install
cmake --build src/c-blosc-knl-build --target install --parallel 16

# ADIOS2
git clone -b v2.7.1 https://github.com/ornladios/ADIOS2.git src/adios2
rmdir -rf src/adios2-knl-build
cmake -S src/adios2 -B src/adios2-knl-build -DADIOS2_USE_Blosc=ON -DADIOS2_USE_Fortran=OFF -DADIOS2_USE_Python=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/adios2-2.7.1-knl-install
cmake --build src/adios2-knl-build --target install --parallel 16

# BLAS++ (for PSATD+RZ)
git clone https://bitbucket.org/icl/blaspp.git src/blaspp
rmdir -rf src/blaspp-knl-build
cmake -S src/blaspp -B src/blaspp-knl-build -DBLAS_LIBRARIES=$(CRAY_LIBSCI_PREFIX_DIR)/lib/libsci_gnu.a -DDEACTIVATE_AVX2=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/blaspp-master-knl-install
cmake --build src/blaspp-knl-build --target install --parallel 16

# LAPACK++ (for PSATD+RZ)
git clone https://bitbucket.org/icl/lapackpp.git src/lapackpp
```
(continues on next page)
rm -rf src/lapackpp-knl-build
CXXFLAGS="-DLAPACK_FORTRAN_ADD_" cmake -S src/lapackpp -B src/lapackpp-knl-build -Duse_ ...
cmake --build src/lapackpp-knl-build --target install --parallel 16

For PICMI and Python workflows, also install a virtual environment:

```bash
# establish Python dependencies
python3 -m pip install --user --upgrade pip
go python3 -m pip install --user virtualenv

python3 -m venv $HOME/sw/venvs/knl_warpx
source $HOME/sw/venvs/knl_warpx/bin/activate
python3 -m pip install --upgrade pip
MPICC="cc -shared" python3 -m pip install -U --no-cache-dir -v mpi4py
```

**Haswell**

We use the following modules and environments on the system ($HOME/haswell_warpx.profile).

```
module swap PrgEnv-intel PrgEnv-gnu
module load cmake/3.21.3
module switch cray-libsci cray-libsci/20.09.1
module load cray-hdf5-parallel/1.10.5.2
module load cray-fftw/3.3.8.4
module load cray-python/3.7.3.2

export PKG_CONFIG_PATH=$FFTW_DIR/pkgconfig:$PKG_CONFIG_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/c-blosc-1.12.1-haswell-install:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/adios2-2.7.1-haswell-install:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/blaspp-master-haswell-install:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/lapackpp-master-haswell-install:$CMAKE_PREFIX_PATH

if [ -d "$HOME/sw/venvs/haswell_warpx" ]
then
  source $HOME/sw/venvs/haswell_warpx/bin/activate
fi
```

And install ADIOS2, BLAS++ and LAPACK++:

```
source $HOME/haswell_warpx.profile

# c-blosc (I/O compression)
git clone -b v1.21.1 https://github.com/Blosc/c-blosc.git src/c-blosc
rm -rf src/c-blosc-haswell-build
```

(continues on next page)
cmake --build src/c-blosc-haswell-build --target install --parallel 16

# ADIOS2
git clone -b v2.7.1 https://github.com/ornladios/ADIOS2.git src/adios2
rm -rf src/adios2-haswell-build
cmake -S src/adios2 -B src/adios2-haswell-build -DADIOS2_USE_Blosc=ON -DADIOS2_USE_Fortran=OFF -DADIOS2_USE_Python=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/adios2-2.7.1-haswell-install
cmake --build src/adios2-haswell-build --target install --parallel 16

# BLAS++ (for PSATD+RZ)
git clone https://bitbucket.org/icl/blaspp.git src/blaspp
rm -rf src/blaspp-haswell-build
cmake -S src/blaspp -B src/blaspp-haswell-build -Duse_openmp=ON -Duse_cmake_find_blas=ON -DBLAS_LIBRARIES=${CRAY_LIBSCI_PREFIX_DIR}/lib/libsci_gnu.a -DCMAKE_CXX_STANDARD=17 -DCMAKE_INSTALL_PREFIX=$HOME/sw/blaspp-master-haswell-install
cmake --build src/blaspp-haswell-build --target install --parallel 16

# LAPACK++ (for PSATD+RZ)
git clone https://bitbucket.org/icl/lapackpp.git src/lapackpp
rm -rf src/lapackpp-haswell-build
cmake --build src/lapackpp-haswell-build --target install --parallel 16

For PICMI and Python workflows, also install a virtual environment:

```
# establish Python dependencies
python3 -m pip install --user --upgrade pip
python3 -m pip install --user virtualenv

python3 -m venv $HOME/sw/venvs/haswell_warpx
copy $HOME/sw/venvs/haswell_warpx/bin/activate

python3 -m pip install --upgrade pip
MPICC="cc -shared" python3 -m pip install -U --no-cache-dir -v mpi4py
```
GPU (V100)

Cori provides a partition with 18 nodes that include V100 (16 GB) GPUs. We use the following modules and environments on the system ($HOME/gpu_warpx.profile). You can copy this file from Tools/machines/cori-nersc/gpu_warpx.profile.example:

```
Listing 2.3: You can copy this file from Tools/machines/cori-nersc/gpu_warpx.profile.example.
```

```bash
export proj=m1759

module purge
module load modules
module load cgpu
module load esslurm
module load gcc/8.3.0 cuda/11.4.0 cmake/3.21.3
module load openmpi
export CMAKE_PREFIX_PATH=$HOME/sw/c-blosc-1.12.1-gpu-install:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/adios2-2.7.1-gpu-install:$CMAKE_PREFIX_PATH
if [ -d "$HOME/sw/venvs/gpu_warpx" ]
then
  source $HOME/sw/venvs/gpu_warpx/bin/activate
fi

# compiler environment hints
export CC=$(which gcc)
export CXX=$(which g++)
export FC=$(which gfortran)
export CUDACXX=$(which nvcc)
export CUDAOHOSTCXX=$(which g++)

# optimize CUDA compilation for V100
export AMREX_CUDA_ARCH=7.0

# allocate a GPU, e.g. to compile on
# 10 logical cores (5 physical), 1 GPU
function getNode() {
  salloc -C gpu -N 1 -t 30 -c 10 --gres=gpu:1 -A $proj
}
```

And install ADIOS2:

```
source $HOME/gpu_warpx.profile

# c-blosc (I/O compression)
git clone -b v1.21.1 https://github.com/Blosc/c-blosc.git src/c-blosc
rm -rf src/c-blosc-gpu-build
cmake -S src/c-blosc -B src/c-blosc-gpu-build
-DBUILD_TESTS=OFF -DBUILD_BENCHMARKS=OFF -DDEACTIVATE_AVX2=OFF
DCMAKE_INSTALL_PREFIX=$HOME/sw/c-blosc-1.12.1-gpu-install
cmake --build src/c-blosc-gpu-build --target install --parallel 16
```

(continues on next page)
git clone -b v2.7.1 https://github.com/ornladios/ADIOS2.git src/adios2
rm -rf src/adios2-gpu-build
cmake -S src/adios2 -B src/adios2-gpu-build -DADIOS2_USE_Blosc=ON -DADIOS2_USE_...
→
cmake --build src/adios2-gpu-build --target install --parallel 16

For PICMI and Python workflows, also install a virtual environment:

```
# establish Python dependencies
python3 -m pip install --user --upgrade pip
python3 -m pip install --user virtualenv

python3 -m venv $HOME/sw/venvs/gpu_warpx
source $HOME/sw/venvs/gpu_warpx/bin/activate

python3 -m pip install --upgrade pip
python3 -m pip install -U --no-cache-dir -v mpi4py
```

**Building WarpX**

We recommend to store the above lines in individual `warpx.profile` files, as suggested above. If you want to run on either of the three partitions of Cori, open a new terminal, log into Cori and `source` the environment you want to work with:

```
# KNL:
source $HOME/knl_warpx.profile

# Haswell:
#source $HOME/haswell_warpx.profile

# GPU:
#source $HOME/gpu_warpx.profile
```

**Warning:** Consider that all three Cori partitions are *incompatible*.

Do not `source` multiple `warpx.profile` files in the same terminal session. Open a new terminal and log into Cori again, if you want to switch the targeted Cori partition.

If you re-submit an already compiled simulation that you ran on another day or in another session, *make sure to source* the corresponding `warpx.profile` again after login!

Then, cd into the directory `$HOME/src/warpx` and use the following commands to compile:

```
cd $HOME/src/warpx
rm -rf build

# append if you target GPUs:   -DWarpx_COMPUTE=CUDA
cmake -S . -B build -DWarpx_DIMS=3
cmake --build build -j 16
```
The general *cmake compile-time options and instructions for Python (PICMI) bindings* apply as usual:

```
# PICMI build
cd $HOME/src/warpx

# compile parallel PICMI interfaces with openPMD support and 3D, 2D and RZ
WARPX_MPI=ON BUILD_PARALLEL=16 python3 -m pip install --force-reinstall -v .
```

### Running

Navigate (i.e. `cd`) into one of the production directories (e.g. `$SCRATCH`) before executing the instructions below.

### KNL

The batch script below can be used to run a WarpX simulation on 2 KNL nodes on the supercomputer Cori at NERSC. Replace descriptions between chevrons `<>` by relevant values, for instance `<job name>` could be `laserWakefield`. Do not forget to first `source $HOME/knl_warpx.profile` if you have not done so already for this terminal session.

For PICMI Python runs, the `<path/to/executable>` has to read `python3` and the `<input file>` is the path to your PICMI input script.

```
#!/bin/bash -l

# Copyright 2019 Maxence Thevenet
#
# This file is part of WarpX.
#
# License: BSD-3-Clause-LBNL

#SBATCH -N 2
#SBATCH -t 01:00:00
#SBATCH -q regular
#SBATCH -C knl
#SBATCH -S 4
#SBATCH -J <job name>
#SBATCH -A <allocation ID>
#SBATCH -e WarpX.e%j
#SBATCH -o WarpX.o%j

export OMP_PLACES=threads
export OMP_PROC_BIND=spread

# KNLs have 4 hyperthreads max
export CORI_MAX_HYPETHREAD_LEVEL=4
# We use 64 cores out of the 68 available on Cori KNL,
# and leave 4 to the system (see "#SBATCH -S 4" above).
export CORI_NCORES_PER_NODE=64
```

(continues on next page)
# Typically use 8 MPI ranks per node without hyperthreading,
# i.e., OMP_NUM_THREADS=8
export WARPX_NMPI_PER_NODE=8
export WARPX_HYPERTHREAD_LEVEL=1

# Compute OMP_NUM_THREADS and the thread count (-c option)
export CORI_NHYPERTHREADS_MAX=$(($CORI_MAX_HYPETHREAD_LEVEL * $CORI_NCORES_PER_NODE))
export WARPX_NTHREADS_PER_NODE=$(($WARPX_HYPERTHREAD_LEVEL * $CORI_NCORES_PER_NODE))
export OMP_NUM_THREADS=$(($WARPX_NTHREADS_PER_NODE / $WARPX_NMPI_PER_NODE))
export WARPX_THREAD_COUNT=$(($CORI_NHYPERTHREADS_MAX / $WARPX_NMPI_PER_NODE))

# for async_io support: (optional)
export MPICH_MAX_THREAD_SAFETY=multiple

srun --cpu_bind=cores -n $(($SLURM_JOB_NUM_NODES * $WARPX_NMPI_PER_NODE)) -c $(WARPX_THREAD_COUNT) \<path/to/executable> <input file> \> output.txt

To run a simulation, copy the lines above to a file cori_knl.sbatch and run

sbatch cori_knl.sbatch

to submit the job.

For a 3D simulation with a few (1-4) particles per cell using FDTD Maxwell solver on Cori KNL for a well load-balanced problem (in our case laser wakefield acceleration simulation in a boosted frame in the quasi-linear regime), the following set of parameters provided good performance:

- **amr.max_grid_size=64** and **amr.blocking_factor=64** so that the size of each grid is fixed to $64^3$ (we are not using load-balancing here).

- **8 MPI ranks per KNL node**, with OMP_NUM_THREADS=8 (that is 64 threads per KNL node, i.e. 1 thread per physical core, and 4 cores left to the system).

- **2 grids per MPI, i.e., 16 grids per KNL node.**

### Haswell

The batch script below can be used to run a WarpX simulation on 1 Haswell node on the supercomputer Cori at NERSC. Do not forget to first source $HOME/haswell_warpx.profile if you have not done so already for this terminal session.

```
#!/bin/bash -l
# Just increase this number of you need more nodes.
#SBATCH -N 1
```

Listing 2.5: You can copy this file from Tools/machines/cori-nersc/cori_haswell.sbatch.
To run a simulation, copy the lines above to a file cori_haswell.sbatch and run

sbatch cori_haswell.sbatch

to submit the job.

For a 3D simulation with a few (1-4) particles per cell using FDTD Maxwell solver on Cori Haswell for a well load-balanced problem (in our case laser wakefield acceleration simulation in a boosted frame in the quasi-linear regime), the following set of parameters provided good performance:

- **4 MPI ranks per Haswell node** (2 MPI ranks per Intel Xeon E5-2698 v3), with `OMP_NUM_THREADS=16` (which uses 2x hyperthreading)
GPU (V100)

Do not forget to first source $HOME/gpu_warpx.profile if you have not done so already for this terminal session.

Due to the limited amount of GPU development nodes, just request a single node with the above defined getNode function. For single-node runs, try to run one grid per GPU.

A multi-node batch script template can be found below:

```bash
#!/bin/bash -l

# Copyright 2021 Axel Huebl
# This file is part of WarpX.
# License: BSD-3-Clause-LBNL
#
# Ref:
# - https://docs-dev.nersc.gov/cgpu/hardware/
# - https://docs-dev.nersc.gov/cgpu/access/
# - https://docs-dev.nersc.gov/cgpu/usage/#controlling-task-and-gpu-binding
#
# Just increase this number of you need more nodes.
#SBATCH -N 2
#SBATCH -t 03:00:00
#SBATCH -J <job name>
#SBATCH -A m1759
#SBATCH -q regular
#SBATCH -C gpu
#SBATCH --gres=gpu:8
#SBATCH --exclusive
#SBATCH --tasks-per-node=8
#SBATCH --cpus-per-task=10
#SBATCH -e WarpX.e%j
#SBATCH -o WarpX.o%j

# each Cori GPU node has 2 sockets of Intel Xeon Gold 6148 ('Skylake') @ 2.40 GHz
export WARPX_NMPI_PER_NODE=8

# each MPI rank per half-socket has 10 physical cores
# or 20 logical (virtual) cores
# we split half-sockets again by 2 to have one MPI rank per GPU
# over-subscribing each physical core with 2x
# hyperthreading leads to often to slight speedup on Intel
# the settings below make sure threads are close to the
# controlling MPI rank (process) per half socket and
# distribute equally over close-by physical cores and,
# for N>20, also equally over close-by logical cores
export OMP_PROC_BIND=spread
```

(continues on next page)
export OMP_PLACES=threads
export OMP_NUM_THREADS=10

# for async_io support: (optional)
export MPICH_MAX_THREAD_SAFETY=multiple

EXE="<path/to/executable>"
srun --cpu_bind=cores --gpus-per-task=1 --gpu-bind=map_gpu:0,1,2,3,4,5,6,7 \
    -n $((${SLURM_JOB_NUM_NODES} * ${WARPX_NMPI_PER_NODE})) \
    ${EXE} <input file> \
> output.txt

Post-Processing

For post-processing, most users use Python via NERSC’s Jupyter service (Docs).

As a one-time preparatory setup, create your own Conda environment as described in NERSC docs. In this manual, we often use this conda create line over the officially documented one:

conda create -n myenv -c conda-forge python mamba ipykernel ipympl==0.8.6 matplotlib␣
→ numpy pandas yt openpmd-viewer openpmd-api h5py fast-histogram dask dask-jobqueue␣
→ pyarrow

We then follow the Customizing Kernels with a Helper Shell Script section to finalize the setup of using this conda-environment as a custom Jupyter kernel.

When opening a Jupyter notebook, just select the name you picked for your custom kernel on the top right of the notebook.

Additional software can be installed later on, e.g., in a Jupyter cell using !mamba install -c conda-forge ....
Software that is not available via conda can be installed via !python -m pip install ....

**Warning:** Jan 6th, 2022 (NERSC-INC0179165 and ipympl #416): Above, we fixated the ipympl version to not take the latest release of Matplotlib Jupyter Widgets. This is an intentional work-around; the ipympl version needs to exactly fit the version pre-installed on the Jupyter base system.

Perlmutter (NERSC)

**Warning:** Perlmutter is still in acceptance testing and environments change often. Please reach visit this page often for updates and reach out to us if something needs an update.

The Perlmutter cluster is located at NERSC.

If you are new to this system, please see the following resources:

- NERSC user guide
- Batch system: Slurm
- Jupyter service

2.3. HPC
• Production directories:
  – $PSCRATCH: per-user production directory (<TBD>TB)
  – /global/cscratch1/sd/m3239: shared production directory for users in the project m3239 (50TB)
  – /global/cfs/cdirs/m3239/: community file system for users in the project m3239 (100TB)

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```bash
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system ($HOME/perlmutter_warpx.profile).

Listing 2.7: You can copy this file from Tools/machines/ perlmutter-nersc/perlmutter_warpx.profile.example.

```bash
# please set your project account
#export proj=<yourProject>  # LBNL/AMP: m3906_g

# required dependencies
module load cmake/3.22.0
module swap PrgEnv-nvidia PrgEnv-gnu
module load cudatoolkit

# optional: just an additional text editor
#module load nano  # TODO: request from support

# optional: for openPMD and PSATD+RZ support
module load cray-hdf5-parallel/1.12.0.7
export CMAKE_PREFIX_PATH=$HOME/sw/perlmutter/c-blosc-1.21.1:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/perlmutter/adios2-2.7.1:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/perlmutter/blaspp-master:$CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=$HOME/sw/perlmutter/lapackpp-master:$CMAKE_PREFIX_PATH
export LD_LIBRARY_PATH=$HOME/sw/perlmutter/c-blosc-1.21.1/lib64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$HOME/sw/perlmutter/adios2-2.7.1/lib64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$HOME/sw/perlmutter/blaspp-master/lib64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$HOME/sw/perlmutter/lapackpp-master/lib64:$LD_LIBRARY_PATH

# optional: for Python bindings or libEnsemble
module load cray-python/3.9.4.2

if [ -d "$HOME/sw/perlmutter/venvs/warpx" ]
then
  source $HOME/sw/perlmutter/venvs/warpx/bin/activate
fi

# an alias to request an interactive batch node for one hour
# for parallel execution, start on the batch node: srun <command>
alias getNode="salloc -N 1 --ntasks-per-node=4 -t 1:00:00 -q interactive -C gpu --gpu-...
  bind=single:1 -c 32 -G 4 -A $proj"
```

(continues on next page)
We recommend to store the above lines in a file, such as `$HOME/perlmutter_warpx.profile`, and load it into your shell after a login:

```
source $HOME/perlmutter_warpx.profile
```

And since Perlmutter does not yet provide a module for them, install ADIOS2, BLAS++ and LAPACK++:

```
# c-blosc (I/O compression)
git clone -b v1.21.1 https://github.com/Blosc/c-blosc.git src/c-blosc
rm -rf src/c-blosc-pm-build
cmake -S src/c-blosc -B src/c-blosc-pm-build -DBUILD_TESTS=OFF -DBUILD_BENCHMARKS=OFF -DDEACTIVATE_AVX2=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/perlmutter/c-blosc-1.21.1
cmake --build src/c-blosc-pm-build --target install --parallel 16

# ADIOS2
git clone -b v2.7.1 https://github.com/ornladios/ADIOS2.git src/adios2
rm -rf src/adios2-pm-build
cmake -S src/adios2 -B src/adios2-pm-build -DADIOS2_USE_Blosc=ON -DADIOS2_USE_Fortran=OFF -DADIOS2_USE_Python=OFF -DADIOS2_USE_ZeroMQ=OFF -DCMAKE_INSTALL_PREFIX=$HOME/sw/perlmutter/adios2-2.7.1
cmake --build src/adios2-pm-build --target install --parallel 16

# BLAS++ (for PSATD+RZ)
git clone https://bitbucket.org/icl/blaspp.git src/blaspp
rm -rf src/blaspp-pm-build
CXX=$(which CC) cmake -S src/blaspp -B src/blaspp-pm-build -Duse_openmp=ON -Dgpu_backend=CUDA -DCMAKE_FIND_BLAS=ON -DBLAS_LIBRARIES=$(CRAK_LIBSCI_PREFIX_DIR)/lib/libsci_gnu.a -DCMAKE_CXX_STANDARD=17 -DCMAKE_INSTALL_PREFIX=$HOME/sw/perlmutter/blaspp-master
cmake --build src/blaspp-pm-build --target install --parallel 16
```
# LAPACK++ (for PSATD+RZ)

```bash
git clone https://bitbucket.org/icl/lapackpp.git src/lapackpp
rm -rf src/lapackpp-pm-build
CXX=$(
which CC)
CXXFLAGS="-DLAPACK_FORTRAN_ADD_" cmake -S src/lapackpp -B src/lapackpp-
˓→pm-build -Duse_cmake_find_lapack=ON -DBLAS_LIBRARIES=$(
CRAY_LIBSCI_PREFIX_DIR)/lib/
˓→libsci_gnu.a -DLAPACK_LIBRARIES=$(
CRAY_LIBSCI_PREFIX_DIR)/lib/libsci_gnu.a -DCMAKE_CXX_
˓→STANDARD=17 -Dbuild_tests=OFF -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=ON -DCMAKE_INSTALL_
˓→PREFIX=$HOME/sw/perlmutter/lapackpp-master
cmake --build src/lapackpp-pm-build --target install --parallel 16
```

Optionally, download and install Python packages for PICMI or dynamic ensemble optimizations (libEnsemble):

```bash
python3 -m pip install --user --upgrade pip
python3 -m pip install --user virtualenv
python3 -m pip cache purge
rm -rf $HOME/sw/perlmutter/venvs/warpx
python3 -m venv $HOME/sw/perlmutter/venvs/warpx
source $HOME/sw/perlmutter/venvs/warpx/bin/activate
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade wheel
python3 -m pip install --upgrade cython
python3 -m pip install --upgrade numpy
python3 -m pip install --upgrade pandas
python3 -m pip install --upgrade scipy
MPICC="cc -target-accel=nvidia80 -shared" python3 -m pip install --upgrade mpi4py --no-
˓→build-isolation --no-binary mpi4py
python3 -m pip install --upgrade openpmd-api
python3 -m pip install --upgrade matplotlib
python3 -m pip install --upgrade yt
# optional: for libEnsemble
python3 -m pip install -r $HOME/src/warpx/Tools/LibEnsemble/requirements.txt
```

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

```bash
cd $HOME/src/warpx
rm -rf build

cmake -S. -B build -DWarpx_DIMS=3 -DWarpx_COMPUTE=CUDA

cmake --build build -j 16
```

The general `cmake compile-time options` apply as usual.

For a full PICMI install, follow the instructions for Python (PICMI) bindings:

```bash
# PICMI build

cd $HOME/src/warpx

# compile parallel PICMI interfaces in 3D, 2D, 1D and RZ
WARPX_MPI=ON WARPX_COMPUTE=CUDA WARPX_PSATD=ON BUILD_PARALLEL=16 python3 -m pip install -
˓→--force-reinstall --no-deps -v
```

Or, if you are developing, do a quick PICMI install of a single geometry (see: `Warpx_DIMS`) using:
# find dependencies & configure

cmake -S . -B build -DWarpX_COMPUTE=CUDA -DWarpX_PSATD=ON -DWarpX_LIB=ON -DWarpX_DIMS=RZ

# build and then call “python3 -m pip install ...”
cmake --build build --target pip_install -j 16

Running

A100 GPUs (40 GB)

The batch script below can be used to run a WarpX simulation on multiple nodes (change -N accordingly) on the supercomputer Perlmutter at NERSC. Replace descriptions between chevrons <> by relevant values, for instance <input file> could be plasma_mirror_inputs. Note that we run one MPI rank per GPU.

Listing 2.8: You can copy this file from Tools/machines/
perlmutter-nersc/perlmutter.sbatch.

#!/bin/bash -l

# Copyright 2021 Axel Huebl, Kevin Gott
#
# This file is part of WarpX.
#
# License: BSD-3-Clause-LBNL

#SBATCH -t 01:00:00
#SBATCH -N 4
#SBATCH -J WarpX
#SBATCH -A <proj>
#SBATCH -C gpu
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-task=1
#SBATCH --gpus-per-node=4
#SBATCH --gpu-bind=single:1
#SBATCH -o WarpX.o%j
#SBATCH -e WarpX.e%j

# ============
# -N = nodes
# -n = tasks (MPI ranks, usually = G)
# -G = GPUs (full Perlmutter node, 4)
# -c = CPU per task (128 total threads on CPU, 32 per GPU)
#
# --ntasks-per-node= number of tasks (MPI ranks) per node (full node, 4)
# --gpus-per-task= number of GPUs per task (MPI rank) (full node, 4)
# --gpus-per-node= number of GPUs per node (full node, 4)
#
# --gpu-bind=single:1 sets only one GPU to be visible to each MPI rank

(continues on next page)
(continued from previous page)

```bash
# (quiets AMReX init warnings)
#
# Recommend using --ntasks-per-node=4, --gpus-per-task=1 and --gpu-bind=single:1, 
# as they are fixed values and allow for easy scaling with less adjustments.
#
# ============

# GPU-aware MPI
export MPICH_GPU_SUPPORT_ENABLED=1

EXE=./warpx
#EXE=../WarpX/build/bin/warpx.3d.MPI.CUDA.DP.OPMD.QED
#EXE=./main3d.gnu.TPROF.MPI.CUDA.ex
INPUTS=inputs_small

srun ${EXE} ${INPUTS} > output.txt
```

To run a simulation, copy the lines above to a file `perlmutter.sbatch` and run

```
sbatch perlmutter.sbatch
```

to submit the job.

**Post-Processing**

For post-processing, most users use Python via NERSC’s Jupyter service (Docs).

Please follow the same process as for NERSC Cori post-processing. **Important:** The environment + Jupyter kernel must separate from the one you create for Cori.

The Perlmutter $PSCRATCH filesystem is currently not yet available on Jupyter. Thus, store or copy your data to Cori’s $SCRATCH or use the Community FileSystem (CFS) for now.

**Summit (OLCF)**

The Summit cluster is located at OLCF.

If you are new to this system, please see the following resources:

- Summit user guide
- Batch system: LSF
- Jupyter service
- Production directories:
  - $PROJWORK/$proj/: shared with all members of a project (recommended)
  - $MEMBERWORK/$proj/: single user (usually smaller quota)
  - $WORLDWORK/$proj/: shared with all users
- Note that the $HOME directory is mounted as read-only on compute nodes. That means you cannot run in your $HOME.
### Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system ($HOME/summit_warpx.profile).

Listing 2.9: You can copy this file from Tools/machines/summit-olcf/summit_warpx.profile.example.

```bash
# please set your project account
export proj=<yourProject>

# optional: just an additional text editor
module load nano

# required dependencies
module load cmake/3.20.2
module load gcc/9.3.0
module load cuda/11.3.1

# optional: faster re-builds
module load ccache

# optional: for PSATD in RZ geometry support
module load blaspp/2021.04.01
module load lapackpp/2021.04.00

# optional: for PSATD support (CPU only)
#module load fftw/3.3.9

# optional: for QED lookup table generation support
module load boost/1.76.0

# optional: for openPMD support
module load adios2/2.7.1
module load hdf5/1.10.7

# optional: for openPMD support (GNUmake only)
#module load ums
#module load ums-aph114
#module load openpmd-api/0.14.2

# often unstable at runtime with dependencies
module unload darshan-runtime

# optional: Ascent in situ support
# note: build WarpX with CMake
export Ascent_DIR=/gpfs/alpine/csc340/world-shared/software/ascent/2021_09_01_gcc_9_3_0_→warpx/summit/cuda/gnu/ascent-install

# optional: for Python bindings or libEnsemble
```

(continues on next page)
module load python/3.8.10
module load freetype/2.10.4    # matplotlib

# dependencies for numpy, blaspp & lapackpp
module load openblas/0.3.5-omp
export BLAS=${OLCF_OPENBLAS_ROOT}/lib/libopenblas.so
export LAPACK=${OLCF_OPENBLAS_ROOT}/lib/libopenblas.so

if [ -d "$HOME/sw/venvs/warpx" ]
then
    source $HOME/sw/venvs/warpx/bin/activate
fi

# an alias to request an interactive batch node for two hours
# for parallel execution, start on the batch node: jsrun <command>
alias getNode="bsub -q debug -P $proj -W 2:00 -nnodes 1 -Is /bin/bash"

# an alias to run a command on a batch node for up to 30min
# usage: nrun <command>
alias runNode="bsub -q debug -P $proj -W 0:30 -nnodes 1 -I"

# fix system defaults: do not escape $ with a \ on tab completion
shopt -s direxpand

# make output group-readable by default
umask 0027

# optimize CUDA compilation for V100
export AMREX_CUDA_ARCH=7.0

# compiler environment hints
export CC=$(which gcc)
export CXX=$(which g++)
export FC=$(which gfortran)
export CUDA_CXX=$(which nvcc)
export CUDA_HOST_CXX=$(which g++)

We recommend to store the above lines in a file, such as $HOME/summit_warpx.profile, and load it into your shell after a login:

source $HOME/summit_warpx.profile

Optionally, download and install Python packages for PICMI or dynamic ensemble optimizations (libEnsemble):

```
pip3 -m pip install --user --upgrade pip
pip3 -m pip install --user virtualenv
pip3 -m pip cache purge
rm -rf $HOME/sw/venvs/warpx
pip3 -m venv $HOME/sw/venvs/warpx
source $HOME/sw/venvs/warpx/bin/activate
pip3 -m pip install --upgrade pip
pip3 -m pip install --upgrade wheel
pip3 -m pip install --upgrade cython
```
python3 -m pip install --upgrade numpy
python3 -m pip install --upgrade pandas
python3 -m pip install --upgrade scipy
python3 -m pip install --upgrade mpi4py --no-binary mpi4py
python3 -m pip install --upgrade openpmd-api
python3 -m pip install --upgrade matplotlib==3.2.2 # does not try to build freetype itself
python3 -m pip install --upgrade yt
# WIP: issues with nlopt
# python3 -m pip install -r $HOME/src/warpx/Tools/LibEnsemble/requirements.txt

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

cd $HOME/src/warpx
rm -rf build
cmake -S . -B build -DWarpX_DIMS=3 -DWarpX_COMPUTE=CUDA
cmake --build build -j 6

The general `cmake compile-time options` apply as usual.

For a full PICMI install, follow the instructions for Python (PICMI bindings). We only prefix it to request a node for the compilation (runNode), so we can compile faster:

```
# PICMI build
cd $HOME/src/warpx

# compile parallel PICMI interfaces in 3D, 2D, 1D and RZ
runNode WARPX_MPI=ON WARPX_COMPUTE=CUDA WARPX_PSATD=ON BUILD_PARALLEL=32 python3 -m pip ...
--install --force-reinstall --no-deps -v .
```

Or, if you are developing, do a quick PICMI install of a single geometry (see: WarpX_DIMS) using:

```
# find dependencies & configure
cmake -S . -B build -DWarpX_COMPUTE=CUDA -DWarpX_PSATD=ON -DWarpX_LIB=ON -DWarpX_DIMS=RZ

# build and then call "python3 -m pip install ..."
cmake --build build --target pip_install -j 6
```

**Running**

**V100 GPUs (16GB)**

The batch script below can be used to run a WarpX simulation on 2 nodes on the supercomputer Summit at OLCF. Replace descriptions between chevrons <> by relevant values, for instance <input file> could be plasma_mirror_inputs. Note that WarpX runs with one MPI rank per GPU and there are 6 GPUs per node:

```
Listing 2.10: You can copy this file from Tools/machines/
summit-olcf/summit_v100.bsub.
```

```bash
#!/bin/bash
```
# Copyright 2019-2020 Maxence Thevenet, Axel Huebl
#
# This file is part of WarpX.
#
# License: BSD-3-Clause-LBNL
#
# Refs.:
# https://jsrunvisualizer.olcf.ornl.gov/?s4f0o11n6c7g1r11d1b1l0=
# https://docs.olcf.ornl.gov/systems/summit_user_guide.html#cuda-aware-mpi

#$bsub -P <allocation ID>
#$bsub -W 00:10
#$bsub -nnodes 2
#$bsub -J WarpX
#$bsub -o WarpXo.%J
#$bsub -e WarpXe.%J

# make output group-readable by default
umask 0027

# fix problems with collectives since RHEL8 update: OLCFHELP-3545
# disable all the IBM optimized barriers and drop back to HCOLL or OMPI's barrier implementations
export OMPI_MCA_coll.ibm_skip_barrier=true

# libfabric 1.6+: limit the visible devices
# Needed for ADIOS2 SST staging/streaming workflows since RHEL8 update
# https://github.com/ornladios/ADIOS2/issues/2887
#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

# ROMIO has a hint for GPFS named IBM_largeblock_io which optimizes I/O with operations on large blocks
export IBM_largeblock_io=true

# MPI-I/O: ROMIO hints for parallel HDF5 performance
export OMPI_MCA_io=romio321
export ROMIO_HINTS=./romio-hints
# number of hosts: unique node names minus batch node
NUM_HOSTS=$((($( echo $LSB_HOSTS | tr ' ' '' | uniq | wc -l) - 1 ))
cat > romio-hints << EOL
   romio_cb_write enable
   romio_ds_write enable
   cb_buffer_size 16777216
   cb_nodes ${NUM_HOSTS}
EOL

# OpenMP: 1 thread per MPI rank
export OMP_NUM_THREADS=1

# run WarpX

(continues on next page)
To run a simulation, copy the lines above to a file `summit_v100.bsub` and run

```
bsub summit_v100.bsub
```

to submit the job.

For a 3D simulation with a few (1-4) particles per cell using FDTD Maxwell solver on Summit for a well load-balanced problem (in our case laser wakefield acceleration simulation in a boosted frame in the quasi-linear regime), the following set of parameters provided good performance:

- `amr.max_grid_size=256` and `amr.blocking_factor=128`.
- **One MPI rank per GPU** (e.g., 6 MPI ranks for the 6 GPUs on each Summit node)
- **Two `128x128x128` grids per GPU**, or **one `128x128x256` grid per GPU**.

A batch script with more options regarding profiling on Summit can be found at [Summit batch script](https://jsrunvisualizer.olcf.ornl.gov/?s1f0o121n2c21g0r11d1b1l0=).

### Power9 CPUs

Similar to above, the batch script below can be used to run a WarpX simulation on 1 node on the supercomputer Summit at OLCF, on Power9 CPUs (i.e., the GPUs are ignored).

```
#!/bin/bash

# Copyright 2019-2020 Maxence Thevenet, Axel Huebl, Michael Rowan
# # This file is part of WarpX.
# # License: BSD-3-Clause-LBNL
# # Refs.: https://jsrunvisualizer.olcf.ornl.gov/?s1f0o121n2c21g0r11d1b1l0=

#BSUB -P <allocation ID>
#BSUB -W 00:10
#BSUB -n nodes 1
#BSUB -alloc_flags "smt1"
#BSUB -J WarpX
#BSUB -o WarpXo.%J
#BSUB -e WarpXe.%J

# make output group-readable by default
umask 0027

# fix problems with collectives since RHEL8 update: OLCFHELP-3545
# disable all the IBM optimized barriers and drop back to HCOLL or OMPI's barrier__implementations
```

(continues on next page)
export OMPI_MCA_coll_ibm_skip_barrier=true

# libfabric 1.6+: limit the visible devices
# Needed for ADIOS2 SST staging/streaming workflows since RHEL8 update
# https://github.com/ornladios/ADIOS2/issues/2887
#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

# ROMIO has a hint for GPFS named IBM_largeblock_io which optimizes I/O with operations on large blocks
export IBM_largeblock_io=true

# MPI-I/O: ROMIO hints for parallel HDF5 performance
export OMPI_MCA_io=romio321
export ROMIO_HINTS=./romio-hints
# number of hosts: unique node names minus batch node
NUM_HOSTS=$(($(echo $LSB_HOSTS | tr ' ' '
' | uniq | wc -l)) - 1)
cat > romio-hints << EOL
  romio_cb_write enable
  romio_ds_write enable
  cb_buffer_size 16777216
  cb_nodes ${NUM_HOSTS}
EOL

# OpenMP: 21 threads per MPI rank
export OMP_NUM_THREADS=21

# run WarpX
jsrun -n 2 -a 1 -c 21 -r 2 -l CPU-CPU -d packed -b rs <path/to/executable> <input file> > output.txt

For a 3D simulation with a few (1-4) particles per cell using FDTD Maxwell solver on Summit for a well load-balanced problem, the following set of parameters provided good performance:

- `amr.max_grid_size=64` and `amr.blocking_factor=64`
- **Two MPI ranks per node** (i.e. 2 resource sets per node; equivalently, 1 resource set per socket)
- **21 physical CPU cores per MPI rank**
- **21 OpenMP threads per MPI rank** (i.e. 1 OpenMP thread per physical core)
- **SMT 1** (Simultaneous Multithreading level 1)
- **Sixteen ‘64x64x64’ grids per MPI rank** (with default tiling in WarpX, this results in ~49 tiles per OpenMP thread)
I/O Performance Tuning

GPFS Large Block I/O

Setting `IBM_largeblock_io` to `true` disables data shipping, saving overhead when writing/reading large contiguous I/O chunks.

```
export IBM_largeblock_io=true
```

ROMIO MPI-IO Hints

You might notice some parallel HDF5 performance improvements on Summit by setting the appropriate ROMIO hints for MPI-IO operations.

```
export OMP_MCA_io=romio321
export ROMIO_HINTS=./romio-hints
```

You can generate the `romio-hints` by issuing the following command. Remember to change the number of `cb_nodes` to match the number of compute nodes you are using (example here: 64).

```
cat > romio-hints << EOL
romio_cb_write enable
romio_ds_write enable
cb_buffer_size 16777216
cb_nodes 64
EOL
```

The `romio-hints` file contains pairs of key-value hints to enable and tune collective buffering of MPI-IO operations. As Summit’s Alpine file system uses a 16MB block size, you should set the collective buffer size to 16GB and tune the number of aggregators (`cb_nodes`) to the number of compute nodes you are using, i.e., one aggregator per node.

Further details are available at Summit’s documentation page.

Known System Issues

**Warning:** Sep 16th, 2021 (OLCFHELP-3685): The Jupyter service cannot open HDF5 files without hanging, due to a filesystem mounting problem.

Please apply this work-around in a Jupyter cell before opening any HDF5 files for read:

```
import os
os.environ['HDF5_USE_FILE_LOCKING'] = "FALSE"
```

**Warning:** Aug 27th, 2021 (OLCFHELP-3442): Created simulation files and directories are no longer accessible by your team members, even if you create them on `$PROJWORK`. Setting the proper “user mask” (umask) does not yet work to fix this.

Please run those commands after running a simulation to fix this. You can also append this to the end of your job scripts after the `jsrun` line:
```
# cd your-simulation-directory
find . -type d -exec chmod g+rwx {} \;
find . -type f -exec chmod g+rw {} \;
```

**Warning:**  Sep 3rd, 2021 (OLCFHELP-3545): The implementation of barriers in IBM’s MPI fork is broken and leads to crashes at scale. This is seen with runs using 200 nodes and above.

Our batch script templates above apply this work-around before the call to jsrun, which avoids the broken routines from IBM and trades them for an OpenMPI implementation of collectives:

```
export OMPI_MCA_coll_ibm_skip_barrier=true
```

**Warning:**  Sep 3rd, 2021 (OLCFHELP-3319): If you are an active developer and compile middleware libraries (e.g., ADIOS2) yourself that use MPI and/or infiniband, be aware of libfabric: IBM forks the open source version of this library and ships a patched version.

Avoid conflicts with mainline versions of this library in MPI that lead to crashes at runtime by loading alongside the system MPI module:

```
module load libfabric/1.12.1-sysrdma
```

For instance, if you compile large software stacks with Spack, make sure to register libfabric with that exact version as an external module.

If you load the documented ADIOS2 module above, this problem does not affect you, since the correct libfabric version is chosen for this one.

**Warning:**  Related to the above issue, the fabric selection in ADIOS2 was designed for libfabric 1.6. With newer versions of libfabric, a workaround is needed to guide the selection of a functional fabric for RDMA support. Details are discussed in ADIOS2 issue #2887.

The following environment variables can be set as work-arounds, 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
```

**Warning:**  Oct 12th, 2021 (OLCFHELP-4242): There is currently a problem with the pre-installed Jupyter extensions, which can lead to connection splits at long running analysis sessions.

Work-around this issue by running in a single Jupyter cell, before starting analysis:

```
!jupyter serverextension enable --py --sys-prefix dask_labextension
```
Post-Processing

For post-processing, most users use Python via OLCFs’s Jupyter service (Docs).

We usually just install our software on-the-fly on Summit. When starting up a post-processing session, run this in your first cells:

```bash
# work-around for OLCFHELP-4242
!jupyter serverextension enable --py --sys-prefix dask_labextension

# next Jupyter cell: install a faster & better conda package manager
!conda install --c conda-forge -y mamba

# next cell: the software you want
!mamba install -c conda-forge -y openpmd-api openpmd-viewer ipympl ipywidgets fast-
˓→histogram yt

# restart notebook
```

Spock (OLCF)

The Spock cluster is located at OLCF.

If you are new to this system, please see the following resources:

- Spock user guide
- Batch system: Slurm
- Production directories:
  - $PROJWORK/$proj/: shared with all members of a project (recommended)
  - $MEMBERWORK/$proj/: single user (usually smaller quota)
  - $WORLDWORK/$proj/: shared with all users
- Note that the $HOME directory is mounted as read-only on compute nodes. That means you cannot run in your $HOME.

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```bash
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system ($HOME/spock_warpx.profile).

Listing 2.12: You can copy this file from Tools/machines/spock-olcf/spock_warpx.profile.example.

```bash
# please set your project account
#export proj=<yourProject>

# required dependencies
module load cmake/3.20.2
```

(continues on next page)
module load craype-accel-amd-gfx908
module load rocm/4.3.0

# optional: faster builds
module load ccache
module load ninja

# optional: just an additional text editor
module load nano

# optional: an alias to request an interactive node for one hour
alias getNode="salloc -A $proj -J warpx -t 01:00:00 -p ecp -N 1"

# fix system defaults: do not escape $ with a \ on tab completion
shopt -s direxpand

# optimize CUDA compilation for MI100
export AMREX_AMD_ARCH=gfx908

# compiler environment hints
export CC=$ROCM_PATH/llvm/bin/clang
export CXX=$(which hipcc)
export LDFLAGS="-L${CRAYLIBS_X86_64} $(CC --cray-print-opts=libs) -lmpi"

# GPU aware MPI: ${PE_MPICH_GTL_DIR_gfx908} -lmpi_gtl_hsa

We recommend to store the above lines in a file, such as $HOME/spock_warpx.profile, and load it into your shell after a login:

```
source $HOME/spock_warpx.profile
```

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

```
cd $HOME/src/warpx
rm -rf build

cmake -S . -B build -DWarpx_DIMS=3 -DWarpx_COMPUTE=HIP -DAMReX_AMD_ARCH=gfx908 -DMPI_CXX_COMPILER=$(which CC) -DMPI_C_COMPILER=$(which cc) -DMPI_COMPILER_FLAGS="--cray-print-opts=all"
cmake --build build -j 10
```

The general `cmake compile-time options` apply as usual.

**Running**

**MI100 GPUs (32 GB)**

After requesting an interactive node with the `getNode` alias above, run a simulation like this, here using 4 MPI ranks:

```
srun -n 4 -c 2 --ntasks-per-node=4 ./warpx inputs
```

Or in non-interactive runs started with `sbatch`:
Listing 2.13: You can copy this file from Tools/machines/spock-olcf/spock_mi100.sbatch.

```
#!/bin/bash

#SBATCH -A <project id>
#SBATCH -J warpx
#SBATCH -o %x-%j.out
#SBATCH -t 00:10:00
#SBATCH -p ecp
#SBATCH -N 1

export OMP_NUM_THREADS=1
srun -n 4 -c 2 --ntasks-per-node=4 ./warpx inputs > output.txt
```

We can currently use up to 4 nodes with 4 GPUs each (maximum: \(-N 4 \ -n 16\)).

**Post-Processing**

For post-processing, most users use Python via OLCFs's Jupyter service (Docs).

Please follow the same guidance as for *OLCF Summit post-processing*.

**Crusher (OLCF)**

The Crusher cluster is located at OLCF. Each node contains 4 AMD MI250X GPUs, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. You can think of the 8 GCDs as 8 separate GPUs, each having 64 GB of high-bandwidth memory (HBM2E).

If you are new to this system, please see the following resources:

- Crusher user guide
- Batch system: Slurm
- Production directories:
  - $PROJWORK/$proj/: shared with all members of a project (recommended)
  - $MEMBERWORK/$proj/: single user (usually smaller quota)
  - $WORLDWORK/$proj/: shared with all users
- Note that the $HOME directory is mounted as read-only on compute nodes. That means you cannot run in your $HOME.
Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system ($HOME/crusher_warpx.profile).

Listing 2.14: You can copy this file from Tools/machines/crusher-olcf/crusher_warpx.profile.example.

```
# please set your project account
export proj=<yourProject>

# required dependencies
module load cmake/3.22.1
module load craype-accel-amd-gfx90a
module load rocm/5.0.0
module load cray-mpich
module load cce/13.0.1  # must be loaded after rocm

# optional: faster builds
module load ccache
module load ninja

# optional: just an additional text editor
module load nano

# optional: for PSATD in RZ geometry support (not yet available)
#module load blaspp
#module load lapackpp

# optional: for QED lookup table generation support
module load boost/1.77.0-cxx17

# optional: for openPMD support
module load adios2/2.7.1
module load hdf5/1.10.7

# optional: for Python bindings or libEnsemble
module load cray-python/3.9.4.2

# fix system defaults: do not escape $ with a \ on tab completion
shopt -s direxpand

# an alias to request an interactive batch node for one hour
# for paralle execution, start on the batch node: srun <command>
def get@node="salloc -A $proj -J warpx -t 01:00:00 -p batch -N 1 -c 8 --ntasks-per-node=8"

# an alias to run a command on a batch node for up to 30min
# usage: run@node <command>
def run@node="srun -A $proj -J warpx -t 00:30:00 -p batch -N 1 -c 8 --ntasks-per-node=8"
```

(continues on next page)
# GPU-aware MPI
export MPICH_GPU_SUPPORT_ENABLED=1

# optimize CUDA compilation for MI250X
export AMREX_AMD_ARCH=gfx90a

# compiler environment hints
export CC=$(which cc)
export CXX=$(which CC)
export FC=$(which ftn)
export CFLAGS="-I${ROCM_PATH}/include"
export CXXFLAGS="-I${ROCM_PATH}/include"
export LDFLAGS="-L${ROCM_PATH}/lib -lamdhip64"

We recommend to store the above lines in a file, such as $HOME/crusher_warpx.profile, and load it into your shell after a login:

source $HOME/crusher_warpx.profile

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

cd $HOME/src/warpx
rm -rf build

make -S . -B build -DWarpX_DIMS=3 -DWarpX_COMPUTE=HIP
make --build build -j 10

The general `make compile-time options` apply as usual.

Running

**MI250X GPUs (2x64 GB)**

After requesting an interactive node with the `getNode` alias above, run a simulation like this, here using 8 MPI ranks and a single node:

runNode ./warpx inputs

Or in non-interactive runs:

Listing 2.15: You can copy this file from `Tools/machines/crusher-olcf/submit.sh`.

```
#!/usr/bin/env bash

#SBATCH -A <project id>
#SBATCH -J warpx
#SBATCH -o %x-%j.out
#SBATCH -t 00:10:00
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
```

(continues on next page)
Post-Processing

For post-processing, most users use Python via OLCFs's Jupyter service (Docs). Please follow the same guidance as for OLCF Summit post-processing.

Juwels (JSC)

Note: For the moment, WarpX doesn’t run on Juwels with MPI_THREAD_MULTIPLE. Please compile with this compilation flag: MPI_THREAD_MULTIPLE=FALSE.

The Juwels supercomputer is located at JSC.

See this page for a quick introduction. (Full user guide).

• Batch system: Slurm
• Production directories:
  – $SCRATCH: Scratch filesystem for temporary data (90 day purge)
  – $FASTDATA/: Storage location for large data (backed up)
  – Note that the $HOME directory is not designed for simulation runs and producing output there will impact performance.

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system.
Listing 2.16: You can copy this file from Tools/machines/juwels-jsc/juwels_warpx.profile.example.

```bash
# please set your project account
define proj=<yourProject>

# required dependencies
module load ccache
module load CMake
module load GCC
module load CUDA/11.3
module load OpenMPI
module load FFTW
module load HDF5
module load Python

# JUWELS' job scheduler may not map ranks to GPUs, 
# so we give a hint to AMReX about the node layout. 
# This is usually done in Make.<supercomputing center> files in AMReX 
# but there is no such file for JSC yet.
define GPUS_PER_SOCKET=2
define GPUS_PER_NODE=4

# optimize CUDA compilation for V100 (7.0) or for A100 (8.0)
define AMREX_CUDA_ARCH=8.0
```

Note that for now WarpX must rely on OpenMPI instead of the recommended MPI implementation on this platform MVAPICH2.

We recommend to store the above lines in a file, such as `$HOME/juwels_warpx.profile`, and load it into your shell after a login:

```bash
source $HOME/juwels_warpx.profile
```

Then, cd into the directory `$HOME/src/warpx` and use the following commands to compile:

```bash
cd $HOME/src/warpx
rm -rf build

cmake -S . -B build -DWarpX_COMPUTE=CUDA -DWarpX_MPI_THREAD_MULTIPLE=OFF
cmake --build build -j 16
```

The other general compile-time options apply as usual.

The executable will be generated in `build/bin/`.

**Note:** Currently, if you want to use HDF5 output with openPMD, you need to add

```bash
export OMPI_MCA_io=romio321
```

in your job scripts, before running the `srun` command.

---

2.3. HPC
Running

Queue: gpus (4 x Nvidia V100 GPUs)

The Juwels GPUs are V100 (16GB) and A100 (40GB).

An example submission script reads

```
#!/bin/bash -l
#SBATCH -A $proj
#SBATCH --partition=booster
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --gres=gpu:4
#SBATCH --time=00:05:00
#SBATCH --job-name=warpx
#SBATCH --output=warpx-%j-%N.txt
#SBATCH --error=warpx-%j-%N.err

export OMP_NUM_THREADS=1
export OMPI_MCA_io=romio321  # for HDF5 support in openPMD

# you can comment this out if you sourced the warpx.profile
# files before running sbatch:
module load GCC
module load OpenMPI
module load CUDA/11.3
module load HDF5
module load Python

srun -n 8 --cpu_bind=sockets $HOME/src/warpx/build/bin/warpx.3d.MPI.CUDA.DP.OPMD.QED␣
˓
→ inputs
```

Queue: batch (2 x Intel Xeon Platinum 8168 CPUs, 24 Cores + 24 Hyperthreads/CPU)

todo

See the data analysis section for more information on how to visualize the simulation results.
Lassen (LLNL)

The Lassen V100 GPU cluster is located at LLNL.

If you are new to this system, please see the following resources:

- LLNL user account
- Lassen user guide
- Batch system: LSF
- Production directories:
  - `/p/gpfs1/$(whoami)`: personal directory on the parallel filesystem
  - Note that the `$HOME` directory and the `/usr/workspace/$(whoami)` space are NFS mounted and not suitable for production quality data generation.

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```bash
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system (`$HOME/lassen_warpx.profile`).

Listing 2.18: You can copy this file from `Tools/machines/lassen-llnl/lassen_warpx.profile.example`.

```bash
# please set your project account
#export proj=<yourProject>

# required dependencies
module load cmake/3.20.2
module load gcc/8.3.1
module load cuda/11.2.0

# optional: for PSATD support
module load fftw/3.3.8

# optional: for QED lookup table generation support
module load boost/1.70.0

# optional: for openPMD support
# TODO ADIOS2
module load hdf5-parallel/1.10.4

# optional: for PSATD in RZ geometry support
# TODO: blaspp lapackpp

# optional: for Python bindings
module load python/3.8.2

# optional: an alias to request an interactive node for two hours
alias getNode="bsub -G $proj -W 2:00 -nnodes 1 -Is /bin/bash"
```

(continues on next page)
We recommend to store the above lines in a file, such as $HOME/lassen_warpx.profile, and load it into your shell after a login:

```
source $HOME/lassen_warpx.profile
```

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

```
cd $HOME/src/warpx
rm -rf build

cmake -S . -B build -DWarpX_COMPUTE=CUDA

cmake --build build -j 10
```

This will build an executable in build/bin/. The other general compile-time options apply as usual.

**Running**

**V100 GPUs (16GB)**

The batch script below can be used to run a WarpX simulation on 2 nodes on the supercomputer Lassen at LLNL. Replace descriptions between chevrons <> by relevant values, for instance <input file> could be plasma_mirror_inputs. Note that the only option so far is to run with one MPI rank per GPU.

```
Listing 2.19: You can copy this file from Tools/machines/lassen-llnl/lassen.bsub.

#!/bin/bash

# Copyright 2020-2022 Axel Huebl
#
# This file is part of WarpX.
#
# License: BSD-3-Clause-LBNL
#
# Refs.:
# https://jsrunvisualizer.olcf.ornl.gov/?s4f0o11n6c7g1r11d1b1l0=
# https://hpc.llnl.gov/training/tutorials/using-lcs-sierra-system#quick16
```

(continues on next page)
To run a simulation, copy the lines above to a file `lassen.bsub` and run

```bash
bsub lassen.bsub
```

to submit the job.

For a 3D simulation with a few (1-4) particles per cell using FDTD Maxwell solver on V100 GPUs for a well load-balanced problem (in our case laser wakefield acceleration simulation in a boosted frame in the quasi-linear regime), the following set of parameters provided good performance:

- `amr.max_grid_size=256` and `amr.blocking_factor=128`.
- **One MPI rank per GPU** (e.g., 4 MPI ranks for the 4 GPUs on each Lassen node)
- **Two `128x128x128` grids per GPU, or one `128x128x256` grid per GPU.**

### Known System Issues

**Warning**: Feb 17th, 2022 (INC0278922): The implementation of AllGatherv in IBM’s MPI optimization library “libcollectives” is broken and leads to HDF5 crashes for multi-node runs.

Our batch script templates above apply this work-around **before** the call to `jsrun`, which avoids the broken routines from IBM and trades them for an OpenMPI implementation of collectives:

```bash
export OMP_MCA_coll.ibm_skip_allgatherv=true
```
Quartz (LLNL)

The Quartz Intel CPU cluster is located at LLNL.

If you are new to this system, please see the following resources:

- LLNL user account
- Quartz user guide
- Batch system: Slurm
- Production directories:

  - `/p/lustre1/$(whoami)` and `/p/lustre2/$(whoami)`: personal directory on the parallel filesystem
  - Note that the `$HOME` directory and the `/usr/workspace/$(whoami)` space are NFS mounted and not suitable for production quality data generation.

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```
git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system (`$HOME/quartz_warpx.profile`).

Listing 2.20: You can copy this file from `Tools/machines/quartz-llnl/quartz_warpx.profile.example`.

```
# please set your project account
#export proj=<yourProject>

# required dependencies
module load cmake/3.20.2
module load intel/2021.4
module load mvapich2/2.3

# optional: for PSATD support
module load fftw/3.3.8

# optional: for QED lookup table generation support
module load boost/1.73.0

# optional: for openPMD support
# TODO ADIOS2
module load hdf5-parallel/1.10.2

# optional: for PSATD in RZ geometry support
# TODO: blaspp lapackpp

# optional: for Python bindings
module load python/3.8.2

# optional: an alias to request an interactive node for two hours
alias getNode="srun --time=0:30:00 --nodes=1 --ntasks-per-node=2 --cpus-per-task=18 -p pdebug --pty hash"
```

(continues on next page)
# fix system defaults: do not escape $ with a \ on tab completion
shopt -s direxpand

# compiler environment hints
export CC=$(which icc)
export CXX=$(which icpc)
export FC=$(which ifort)
# we need a newer libstdc++:
export CFLAGS="-gcc-name=/usr/tce/packages/gcc/gcc-8.3.1/bin/gcc ${CFLAGS}"
export CXXFLAGS="-gxx-name=/usr/tce/packages/gcc/gcc-8.3.1/bin/g++ ${CXXFLAGS}"  

We recommend to store the above lines in a file, such as $HOME/quartz_warpx.profile, and load it into your shell after a login:

```
source $HOME/quartz_warpx.profile
```

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

```
cd $HOME/src/warpx
rm -rf build
cmake -S . -B build
cmake --build build -j 6
```

This will build an executable in build/bin/. The other general compile-time options apply as usual.

## Running

### Intel Xeon E5-2695 v4 CPUs

The batch script below can be used to run a WarpX simulation on 2 nodes on the supercomputer Quartz at LLNL. Replace descriptions between chevrons <> by relevant values, for instance <input file> could be plasma_mirror_inputs.

```
#!/bin/bash -l
# Just increase this number of you need more nodes.
#SBATCH -N 2
#SBATCH -t 24:00:00
#SBATCH -A <allocation ID>
#SBATCH --qos=normal
#SBATCH --license=lustre1,lustre2
#SBATCH --export=ALL
#SBATCH -e error.txt
```

Listing 2.21: You can copy this file from Tools/machines/quartz-llnl/quartz.sbatch.

(continues on next page)
#SBATCH -o output.txt
# one MPI rank per half-socket (see below)
#SBATCH --tasks-per-node=2
# request all logical (virtual) cores per half-socket
#SBATCH --cpus-per-task=18

# each Quartz node has 1 socket of Intel Xeon E5-2695 v4
# each Xeon CPU is divided into 2 bus rings that each have direct L3 access
export WARPX_NMPI_PER_NODE=2

# each MPI rank per half-socket has 9 physical cores
# or 18 logical (virtual) cores
# over-subscribing each physical core with 2x
# hyperthreading led to a slight (3.5%) speedup on Cori’s Intel Xeon E5-2698 v3,
# so we do the same here
# the settings below make sure threads are close to the
# controlling MPI rank (process) per half socket and
# distribute equally over close-by physical cores and,
# for N>9, also equally over close-by logical cores
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_NUM_THREADS=18

EXE="<path/to/executable>"  # e.g. ./warpx

srun --cpu_bind=cores -n $(( ${SLURM_JOB_NUM_NODES} * ${WARPX_NMPI_PER_NODE} )) $EXE
  <input file>

To run a simulation, copy the lines above to a file quartz.sbatch and run

sbatch quartz.sbatch

to submit the job.

Lawrencium (LBNL)

The Lawrencium cluster is located at LBNL.

If you are new to this system, please see the following resources:

- Lawrencium user guide
- Batch system: Slurm
- Production directories:
  - /global/scratch/$USER/: production directory
  - /global/home/groups/$GROUP/: group production directory
  - /global/home/users/$USER: home directory (10 GB)

Note: to do
Ookami (Stony Brook)

The Ookami cluster is located at Stony Brook University.

If you are new to this system, please see the following resources:

- Ookami documentation
- Batch system: Slurm (see available queues)
- Filesystem locations:
  - /lustre/home/<netid> (30GByte, backed up)
  - /lustre/scratch/<netid> (14 day purge)
  - /lustre/projects/<your_group>* (1TBByte default, up to 8TB possible, shared within our group/project, backed up, prefer this location)

We use Ookami as a development cluster for A64FX. The cluster also provides a few extra nodes, e.g. two Thunder X2 (ARM) nodes.

Installation

Use the following commands to download the WarpX source code and switch to the correct branch:

```
$ git clone https://github.com/ECP-WarpX/WarpX.git $HOME/src/warpx
```

We use the following modules and environments on the system ($HOME/warpx_gcc10.profile).

Listing 2.22: You can copy this file from Tools/machines/ookami-sbu/ookami_warpx.profile.example.

```
# please set your project account (not relevant yet)
export proj=<yourProject>

# required dependencies
module load cmake/3.19.0
module load gcc/10.3.0
module load openmpi/gcc10/4.1.0

# optional: faster builds (not available yet)
#module load ccache
#module load ninja

# optional: for PSATD support (not available yet)
#module load fftw

# optional: for QED lookup table generation support (not available yet)
#module load boost

# optional: for openPMD support
#module load adios2 # not available yet
#module load hdf5 # only serial

# compiler environment hints
export CC=$(which gcc)
```

(continues on next page)
We recommend to store the above lines in a file, such as $HOME/warpx_gcc10.profile, and load it into your shell after a login:

```bash
source $HOME/warpx_gcc10.profile
```

Then, cd into the directory $HOME/src/warpx and use the following commands to compile:

```bash
cd $HOME/src/warpx
rm -rf build
cmake -S . -B build -DWarpX_COMPUTE=OMP
cmake --build build -j 10

# or (currently better performance)
cmake -S . -B build -DWarpX_COMPUTE=NOACC
cmake --build build -j 10
```

The general `cmake` compile-time options apply as usual.

### Running

For running on 48 cores of a single node:

```bash
srun -p short -N 1 -n 48 --pty bash
OMP_NUM_THREADS=1 mpiexec -n 48 --map-by ppr:12:numa:pe=1 --report-bindings ./warpx_inputs

# alternatively, using 4 MPI ranks with each 12 threads on a single node:
OMP_NUM_THREADS=12 mpiexec -n 4 --map-by ppr:4:numa:pe=12 --report-bindings ./warpx_inputs
```

The Ookami HPE Apollo 80 system has 174 A64FX compute nodes each with 32GB of high-bandwidth memory.

### Additional Compilers

This section is just a note for developers. We compiled with the Fujitsu Compiler (Clang) with the following build string:

```bash
cmake -S . -B build
  -DCMAKE_C_COMPILER=$($which mpifcc)
  -DCMAKE_C_COMPILER_ID="Clang"
  -DCMAKE_C_COMPILER_VERSION=12.0
  -DCMAKE_C_STANDARD_COMPUTED_DEFAULT="11"
  -DCMAKE_CXX_COMPILER=$($which mpiFCC)
  -DCMAKE_CXX_COMPILER_ID="Clang"
  -DCMAKE_CXX_COMPILER_VERSION=12.0
  -DCMAKE_CXX_STANDARD_COMPUTED_DEFAULT="14"
```

(continues on next page)
Note that the best performance for A64FX is currently achieved with the GCC or ARM compilers.

**LXPLUS (CERN)**

The LXPLUS cluster is located at CERN.

- Lxplus documentation
- Batch system: HTCondor
- Filesystem locations:
  - User folder: /afs/cern.ch/user/<a>/<account> (10GByte)
  - Work folder: /afs/cern.ch/work/<a>/<account> (100GByte)
  - Eos storage: /eos/home-<a>/<account> (1T)

Through LXPLUS we have access to CPU and GPU nodes (the latter equipped with NVIDIA V100 and T4 GPUs).

**Installation**

Only very little software is pre-installed on LXPLUS so we show how to install from scratch all the dependencies using Spack.

For size reasons it is not advisable to install WarpX in the `$HOME` directory, while it should be installed in the “work directory”. For this purpose we set an environment variable with the path to the “work directory”

```
export WORK=/afs/cern.ch/work/${USER:0:1}/${USER}/
```

We clone WarpX in `$WORK`:

```
cd $WORK
git clone https://github.com/ECP-WarpX/WarpX.git warpx
```

**Installation profile file**

The easiest way to install the dependencies is to use the pre-prepared `warpx.profile` as follows:

```
cp $WORK/warpx/WarpX/Tools/machines/lxplus-cern/lxplus_warpx.profile.example $WORK/...
lxplus_warpx.profile
source $WORK/lxplus_warpx.profile
```

When doing this one can directly skip to the *Building WarpX* section.

To have the environment activated at every login it is then possible to add the following lines to the `.bashrc`

```
export WORK=/afs/cern.ch/work/${USER:0:1}/${USER}/
source $WORK/lxplus_warpx.profile
```
GCC

The pre-installed GNU compiler is outdated so we need a more recent compiler. Here we use the gcc 11.2.0 from the LCG project, but other options are possible.

We activate it by doing

```
source /cvmfs/sft.cern.ch/lcg/releases/gcc/11.2.0-ad950/x86_64-centos7/setup.sh
```

In order to avoid using different compilers this line could be added directly into the $HOME/.bashrc file.

Spack

We download and activate Spack in $WORK:

```
cd $WORK
git clone -c feature.manyFiles=true https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
```

Now we add our gcc 11.2.0 compiler to spack:

```
spack compiler find /cvmfs/sft.cern.ch/lcg/releases/gcc/11.2.0-ad950/x86_64-centos7/bin
```

Installing the Dependencies

To install the dependencies we create a virtual environment, which we call warpx-lxplus:

```
spack env create warpx-lxplus $WORK/WarpX/Tools/machines/lxplus-cern/spack.yaml
spack env activate warpx-lxplus
spack install
```

If the GPU support or the Python bindings are not needed, it's possible to skip the installation by respectively setting the following environment variables export SPACK_STACK_USE_PYTHON=0 and export SPACK_STACK_USE_CUDA = 0 before running the previous commands.

After the installation is done once, all we need to do in future sessions is just activate the environment again:

```
spack env activate warpx-lxplus
```

The environment warpx-lxplus (or -cuda or -cuda-py) must be reactivated everytime that we log in so it could be a good idea to add the following lines to the .bashrc:

```
source $WORK/spack/share/spack/setup-env.sh
spack env activate -d warpx-lxplus
cd $HOME
```
Building WarpX

We prepare and load the Spack software environment as above. Then we build WarpX:

```sh
  cmake -S . -B build
  cmake --build build -j 6
```

Or if we need to compile with CUDA:

```sh
  cmake -S . -B build -DWarpX_COMPUTE=CUDA
  cmake --build build -j 6
```

Python Bindings

Here we assume that a Python interpreter has been set up as explained previously.

Now, ensure Python tooling is up-to-date:

```sh
  python3 -m pip install -U pip setuptools wheel
```

Then we compile WarpX as in the previous section (with or without CUDA) adding `-DWarpX_LIB=ON` and then we install it into our Python:

```sh
  cmake -S . -B build -DWarpX_COMPUTE=CUDA -DWarpX_LIB=ON
  cmake --build build --target pip_install -j 6
```

This builds WarpX for 3D geometry.

Alternatively, if you like to build WarpX for all geometries at once, use:

```
  BUILD_PARALLEL=6 python3 -m pip wheel .
  python3 -m pip install pywarpx-*whl
```

Tip: Your HPC system is not in the list? Open an issue and together we can document it!
3.1 Run WarpX

In order to run a new simulation:

1. create a new directory, where the simulation will be run
2. make sure the WarpX executable is either copied into this directory or in your PATH environment variable
3. add an inputs file and on HPC systems a submission script to the directory
4. run

3.1.1 1. Run Directory

On Linux/macOS, this is as easy as this

```
mkdir -p <run_directory>
```

Where `<run_directory>` by the actual path to the run directory.

3.1.2 2. Executable

If you installed warpX with a package manager, a warpx-prefixed executable will be available as a regular system command to you. Depending on the chosen build options, the name is suffixed with more details. Try it like this:

```
warpx<TAB>
```

Hitting the `<TAB>` key will suggest available WarpX executables as found in your PATH environment variable.

**Note:** WarpX needs separate binaries to run in dimensionality of 1D, 2D, 3D, and RZ. We encode the supported dimensionality in the binary file name.

If you compiled the code yourself, the WarpX executable is stored in the source folder under build/bin. We also create a symbolic link that is just called warpx that points to the last executable you built, which can be copied, too. Copy the executable to this directory:

```
cp build/bin/<warpx_executable> <run_directory>/
```

where `<warpx_executable>` should be replaced by the actual name of the executable (see above) and `<run_directory>` by the actual path to the run directory.
3.1.3 3. Inputs

Add an input file in the directory (see examples and parameters). This file contains the numerical and physical parameters that define the situation to be simulated.

On HPC systems, also copy and adjust a submission script that allocated computing nodes for you. Please reach out to us if you need help setting up a template that runs with ideal performance.

3.1.4 4. Run

Run the executable, e.g. with MPI:

```bash
cd <run_directory>

# run with an inputs file:
mpirun -np <n_ranks> ./warpx <input_file>

or

# run with a PICMI input script:
mpirun -np <n_ranks> python <python_script>
```

Here, `<n_ranks>` is the number of MPI ranks used, and `<input_file>` is the name of the input file ( `<python_script>` is the name of the PICMI script). Note that the actual executable might have a longer name, depending on build options.

We used the copied executable in the current directory (./); if you installed with a package manager, skip the ./ because WarpX is in your PATH.

On an HPC system, you would instead submit the job script at this point, e.g. `sbatch <submission_script>` (SLURM on Cori/NERSC) or `bsub <submission_script>` (LSF on Summit/OLCF).

Tip: In the next sections, we will explain parameters of the `<input_file>`. You can overwrite all parameters inside this file also from the command line, e.g.:

```bash
mpirun -np 4 ./warpx <input_file> max_step=10 warpx.numprocs=1 2 2
```

3.2 Input Parameters

Note: The AMReX parser (see Math parser and user-defined constants) is used for the right-hand-side of all input parameters that consist of one or more integers or floats, so expressions like `<species_name>.density_max = "2. +1."` and/or using user-defined constants are accepted.
3.2.1 Overall simulation parameters

- **authors** *(string; e.g. "Jane Doe <jane@example.com>, Jimmy Joe <jimmy@example.com>"*) Authors of an input file / simulation setup. When provided, this information is added as metadata to (openPMD) output files.

- **max_step** *(integer)* The number of PIC cycles to perform.

- **stop_time** *(float; in seconds)* The maximum physical time of the simulation. Can be provided instead of max_step. If both max_step and stop_time are provided, both criteria are used and the simulation stops when the first criterion is hit.

- **warpx.gamma_boost** *(float)* The Lorentz factor of the boosted frame in which the simulation is run. (The corresponding Lorentz transformation is assumed to be along warpx.boost_direction.)

  When using this parameter, some of the input parameters are automatically converted to the boosted frame. (See the corresponding documentation of each input parameter.)

  **Note:** For now, only the laser parameters will be converted.

- **warpx.boost_direction** *(string: x, y or z)* The direction of the Lorentz-transform for boosted-frame simulations (The direction y cannot be used in 2D simulations.)

- **warpx.zmax_plasma_to_compute_max_step** *(float)* optional Can be useful when running in a boosted frame. If specified, automatically calculates the number of iterations required in the boosted frame for the lower z end of the simulation domain to reach warpx.zmax_plasma_to_compute_max_step (typically the plasma end, given in the lab frame). The value of max_step is overwritten, and printed to standard output. Currently only works if the Lorentz boost and the moving window are along the z direction.

- **warpx.verbose** *(0 or 1; default is 1 for true)* Controls how much information is printed to the terminal, when running WarpX.

- **warpx.always_warn_immediately** *(0 or 1; default is 0 for false)* If set to 1, WarpX immediately prints every warning message as soon as it is generated. It is mainly intended for debug purposes, in case a simulation crashes before a global warning report can be printed.

- **warpx.abort_on_warning_threshold** *(string: low, medium or high) optional* Optional threshold to abort as soon as a warning is raised. If the threshold is set, warning messages with priority greater than or equal to the threshold trigger an immediate abort. It is mainly intended for debug purposes, and is best used with warpx.always_warn_immediately=1.

- **warpx.random_seed** *(string or int > 0) optional* If provided warpx.random_seed = random, the random seed will be determined using std::random_device and std::clock(), thus every simulation run produces different random numbers. If provided warpx.random_seed = n, and it is required that n > 0, the random seed for each MPI rank is (mpi_rank+1) * n, where mpi_rank starts from 0. n = 1 and warpx.random_seed = default produce the default random seed. Note that when GPU threading is used, one should not expect to obtain the same random numbers, even if a fixed warpx.random_seed is provided.

- **warpx.do_electrostatic** *(string) optional (default none)* Specifies the electrostatic mode. When turned on, instead of updating the fields at each iteration with the full Maxwell equations, the fields are recomputed at each iteration from the Poisson equation. There is no limitation on the timestep in this case, but electromagnetic effects (e.g. propagation of radiation, lasers, etc.) are not captured. There are two options:

  - **labframe:** Poisson's equation is solved in the lab frame with the charge density of all species combined. There will only be E fields.

  - **relativistic:** Poisson's equation is solved for each species separately taking into account their averaged velocities. The field is mapped to the simulation frame and will produce both E and B fields.
See the AMReX documentation for details of the MLMG solver (the default solver used with electrostatic simulations). The default behavior of the code is to check whether there is non-zero charge density in the system and if so force the MLMG solver to use the solution max norm when checking convergence. If there is no charge density, the MLMG solver will switch to using the initial guess max norm error when evaluating convergence and an absolute error tolerance of $10^{-6} \text{ V/m}^2$ will be used (unless a different non-zero value is specified by the user via `warpx.self_fields_absolute_tolerance`).

- **warpx.self_fields_required_precision** *(float, default: 1.e-11)* The relative precision with which the electrostatic space-charge fields should be calculated. More specifically, the space-charge fields are computed with an iterative Multi-Level Multi-Grid (MLMG) solver. This solver can fail to reach the default precision within a reasonable time. This only applies when `warpx.do_electrostatic = labframe`.

- **warpx.self_fields_absolute_tolerance** *(float, default: 0.0)* The absolute tolerance with which the space-charge fields should be calculated in units of $\text{V/m}^2$. More specifically, the acceptable residual with which the solution can be considered converged. In general this should be left as the default, but in cases where the simulation state changes very little between steps it can occur that the initial guess for the MLMG solver is so close to the converged value that it fails to improve that solution sufficiently to reach the `self_fields_required_precision` value.

- **warpx.self_fields_max_iters** *(integer, default: 200)* Maximum number of iterations used for MLMG solver for space-charge fields calculation. In case if MLMG converges but fails to reach the desired `self_fields_required_precision`, this parameter may be increased. This only applies when `warpx.do_electrostatic = labframe`.

- **warpx.self_fields_verbosity** *(integer, default: 2)* The verbosity used for MLMG solver for space-charge field calculation. Currently MLMG solver looks for verbosity levels from 0-5. A higher number results in more verbose output.

- **amrex.abort_on_out_of_gpu_memory** *(0 or 1; default is 1 for true)* When running on GPUs, memory that does not fit on the device will be automatically swapped to host memory when this option is set to 0. This will cause severe performance drops. Note that even with this set to 1 WarpX will not catch all out-of-memory events yet when operating close to maximum device memory. Please also see the documentation in AMReX.

### 3.2.2 Setting up the field mesh

- **amr.n_cell** *(2 integers in 2D, 3 integers in 3D)* The number of grid points along each direction (on the coarsest level)

- **amr.max_level** *(integer, default: 0)* When using mesh refinement, the number of refinement levels that will be used.

  Use 0 in order to disable mesh refinement. Note: currently, 0 and 1 are supported.

- **amr.ref_ratio** *(integer per refined level, default: 2)* When using mesh refinement, this is the refinement ratio per level. With this option, all directions are fined by the same ratio.

  Note: in development; currently, 2 is supported.

- **amr.ref_ratio_vect** *(3 integers for x,y,z per refined level)* When using mesh refinement, this can be used to set the refinement ratio per direction and level, relative to the previous level.

  Example: for three levels, a value of 2 2 4 8 8 16 refines the first level by 2-fold in x and y and 4-fold in z compared to the coarsest level (level 0/mother grid); compared to the first level, the second level is refined 8-fold in x and y and 16-fold in z.

  Note: in development; currently allowed value: 2 2 2.

- **geometry.dims** *(string)* The dimensions of the simulation geometry. Supported values are 1, 2, 3, RZ. For 3, a cartesian geometry of x, y, z is modeled. For 2, the axes are x and z and all physics in y is assumed to be
translation symmetric. For 1, the only axis is z and the dimensions x and y are translation symmetric. For RZ, we apply an azimuthal mode decomposition, with warpX.n_rz_azimuthal_modes providing further control.

Note that this value has to match the WarpX_DIMS compile-time option. If you installed WarpX from a package manager, then pick the right executable by name.

- **geometry.n_rz_azimuthal_modes (integer; 1 by default)** When using the RZ version, this is the number of azimuthal modes. The default is 1, which corresponds to a perfectly axisymmetric simulation.

- **geometry.prob_lo and geometry.prob_hi (2 floats in 2D, 3 floats in 3D; in meters)** The extent of the full simulation box. This box is rectangular, and thus its extent is given here by the coordinates of the lower corner (geometry.prob_lo) and upper corner (geometry.prob_hi). The first axis of the coordinates is x (or r with cylindrical) and the last is z.

- **warpX.do_moving_window (integer; 0 by default)** Whether to use a moving window for the simulation

- **warpX.moving_window_dir (either x, y or z)** The direction of the moving window.

- **warpX.moving_window_v (float)** The speed of moving window, in units of the speed of light (i.e. use 1.0 for a moving window that moves exactly at the speed of light)

- **warpX.start_moving_window_step (integer; 0 by default)** The timestep at which the moving window starts.

- **warpX.end_moving_window_step (integer; default is -1 for false)** The timestep at which the moving window ends.

- **warpX.fine_tag_lo and warpX.fine_tag_hi (2 floats in 2D, 3 floats in 3D; in meters) optional** When using static mesh refinement with 1 level, the extent of the refined patch. This patch is rectangular, and thus its extent is given here by the coordinates of the lower corner (warpX.fine_tag_lo) and upper corner (warpX.fine_tag_hi).

- **warpX.refine_plasma (integer) optional (default 0)** Increase the number of macro-particles that are injected “ahead” of a mesh refinement patch in a moving window simulation.

  Note: in development; only works with static mesh-refinement, specific to moving window plasma injection, and requires a single refined level.

- **warpX.n_current_deposition_buffer (integer)** When using mesh refinement: the particles that are located inside a refinement patch, but within n_current_deposition_buffer cells of the edge of this patch, will deposit their charge and current to the lower refinement level, instead of depositing to the refinement patch itself. See the mesh-refinement section for more details. If this variable is not explicitly set in the input script, n_current_deposition_buffer is automatically set so as to be large enough to hold the particle shape, on the fine grid

- **warpX.n_field_gather_buffer (integer; 0 by default)** When using mesh refinement: the particles that are located inside a refinement patch, but within n_field_gather_buffer cells of the edge of this patch, will gather the fields from the lower refinement level, instead of gathering the fields from the refinement patch itself. This avoids some of the spurious effects that can occur inside the refinement patch, close to its edge. See the mesh-refinement section for more details. If this variable is not explicitly set in the input script, n_field_gather_buffer is automatically set so that it is one cell larger than n_current_deposition_buffer, on the fine grid.

- **warpX.do_single_precision_comms (integer; 0 by default)** Perform MPI communications for field guard regions in single precision. Only meaningful for WarpX_PRECISION=DOUBLE.

- **particles.deposit_on_main_grid (list of strings)** When using mesh refinement: the particle species whose name are included in the list will deposit their charge/current directly on the main grid (i.e. the coarsest level), even if they are inside a refinement patch.
• **particles.gather_from_main_grid** *(list of strings)* When using mesh refinement: the particle species whose name are included in the list will gather their fields from the main grid (i.e. the coarsest level), even if they are inside a refinement patch.

### 3.2.3 Domain Boundary Conditions

• **boundary.field_lo and boundary.field_hi** *(2 strings for 2D, 3 strings for 3D, pml by default)*

Boundary conditions applied to fields at the lower and upper domain boundaries. Options are:

- **Periodic**: This option can be used to set periodic domain boundaries. Note that if the fields for lo in a certain dimension are set to periodic, then the corresponding upper boundary must also be set to periodic. If particle boundaries are not specified in the input file, then particles boundaries by default will be set to periodic. If particles boundaries are specified, then they must be set to periodic corresponding to the periodic field boundaries.

- **pml** (default): This option can be used to add Perfectly Matched Layers (PML) around the simulation domain. See the *PML theory section* for more details. Additional pml algorithms can be explored using the parameters warpx.do_pml_in_domain, warpx.do_particles_in_pml, and warpx.do_pml_j_damping.

- **absorbing_silver_mueller**: This option can be used to set the Silver-Mueller absorbing boundary conditions. These boundary conditions are simpler and less computationally expensive than the pml, but are also less effective at absorbing the field. They only work with the Yee Maxwell solver.

- **damped**: This is the recommended option in the moving direction when using the spectral solver with moving window (currently only supported along z). This boundary condition applies a damping factor to the electric and magnetic fields in the outer half of the guard cells, using a sine squared profile. As the spectral solver is by nature periodic, the damping prevents fields from wrapping around to the other end of the domain when the periodicity is not desired. This boundary condition is only valid when using the spectral solver.

- **pec**: This option can be used to set a Perfect Electric Conductor at the simulation boundary. For the electromagnetic solve, at PEC, the tangential electric field and the normal magnetic field are set to 0. This boundary can be used to model a dielectric or metallic surface. In the guard-cell region, the tangential electric field is set equal and opposite to the respective field component in the mirror location across the PEC boundary, and the normal electric field is set equal to the field component in the mirror location in the domain across the PEC boundary. Similarly, the tangential (and normal) magnetic field components are set equal (and opposite) to the respective magnetic field components in the mirror locations across the PEC boundary. Note that PEC boundary is invalid at r=0 for the RZ solver. Please use **none** option. This boundary condition does not work with the spectral solver. If an electrostatic field solve is used the boundary potentials can also be set through **boundary.potential_lo_x/y/z** and **boundary.potential_hi_x/y/z** (default 0).

- **none**: No boundary condition is applied to the fields with the electromagnetic solver. This option must be used for the RZ-solver at r=0. If the electrostatic solver is used, a Neumann boundary condition (with gradient equal to 0) will be applied on the specified boundary.

• **boundary.particle_lo and boundary.particle_hi** *(2 strings for 2D, 3 strings for 3D, absorbing by default)*

Options are: * **Absorbing**: Particles leaving the boundary will be deleted.

- **Periodic**: Particles leaving the boundary will re-enter from the opposite boundary. The field boundary condition must be consistently set to periodic and both lower and upper boundaries must be periodic.

- **Reflecting**: Particles leaving the boundary are reflected from the boundary back into the domain. When **boundary.reflect_all_velocities** is false, the sign of only the normal velocity is changed, otherwise the sign of all velocities are changed.
• `boundary.reflect_all_velocities (bool) optional (default false)` For a reflecting boundary condition, this flags whether the sign of only the normal velocity is changed or all velocities.

### 3.2.4 Additional PML parameters

- **warpx.pml_ncell (int; default: 10)** The depth of the PML, in number of cells.
- **do_similar_dm_pml (int; default: 1)** Whether or not to use an amrex::DistributionMapping for the PML grids that is similar to the mother grids, meaning that the mapping will be computed to minimize the communication costs between the PML and the mother grids.
- **warpx.pml_delta (int; default: 10)** The characteristic depth, in number of cells, over which the absorption coefficients of the PML increases.
- **warpx.do_pml_in_domain (int; default: 0)** Whether to create the PML inside the simulation area or outside. If inside, it allows the user to propagate particles in PML and to use extended PML.
- **warpx.pml_has_particles (int; default: 0)** Whether to propagate particles in PML or not. Can only be done if PML are in simulation domain, i.e. if `warpx.do_pml_in_domain = 1`.
- **warpx.do_pml_j_damping (int; default: 0)** Whether to damp current in PML. Can only be used if particles are propagated in PML, i.e. if `warpx.pml_has_particles = 1`.
- **warpx.do_pml_dive_cleaning (bool; default: 1)** Whether to use divergence cleaning for E in the PML region. The value must match `warpx.do_pml_divb_cleaning` (either both false or both true). This option seems to be necessary in order to avoid strong Nyquist instabilities in 3D simulations with the PSATD solver, open boundary conditions and PML in all directions. 2D simulations and 3D simulations with open boundary conditions and PML only in one direction might run well even without divergence cleaning. This option is implemented only for the PSATD solver.
- **warpx.do_pml_divb_cleaning (bool; default: 1)** Whether to use divergence cleaning for B in the PML region. The value must match `warpx.do_pml_dive_cleaning` (either both false or both true). This option seems to be necessary in order to avoid strong Nyquist instabilities in 3D simulations with the PSATD solver, open boundary conditions and PML in all directions. 2D simulations and 3D simulations with open boundary conditions and PML only in one direction might run well even without divergence cleaning.

### 3.2.5 Embedded Boundary Conditions

- **warpx.eb_implicit_function (string)** A function of \(x, y, z\) that defines the surface of the embedded boundary. That surface lies where the function value is 0; the physics simulation area is where the function value is negative; the interior of the embedded boundary is where the function value is positive.
- **warpx.eb_potential(x,y,z,t) (string)** Only used when `warpx.do_electrostatic=labframe`. Gives the value of the electric potential at the surface of the embedded boundary, as a function of \(x, y, z\) and time. This function is also evaluated inside the embedded boundary. For this reason, it is important to define this function in such a way that it is constant inside the embedded boundary.

### 3.2. Input Parameters
3.2.6 Distribution across MPI ranks and parallelization

- **warpx.numprocs** *(2 ints for 2D, 3 ints for 3D) optional (default none)* This optional parameter can be used to control the domain decomposition on the coarsest level. The domain will be chopped into the exact number of pieces in each dimension as specified by this parameter. If it’s not specified, the domain decomposition will be determined by the parameters that will be discussed below. If specified, the product of the numbers must be equal to the number of MPI processes.

- **amr.max_grid_size** *(integer) optional (default 128)* Maximum allowable size of each subdomain (expressed in number of grid points, in each direction). Each subdomain has its own ghost cells, and can be handled by a different MPI rank; several OpenMP threads can work simultaneously on the same subdomain.

  If max_grid_size is such that the total number of subdomains is larger that the number of MPI ranks used, than some MPI ranks will handle several subdomains, thereby providing additional flexibility for load balancing.

  When using mesh refinement, this number applies to the subdomains of the coarsest level, but also to any of the finer level.

- **algo.load_balance_intervals** *(string) optional (default 0)* Using the Intervals parser syntax, this string defines the timesteps at which WarpX should try to redistribute the work across MPI ranks, in order to have better load balancing. Use 0 to disable load_balancing.

  When performing load balancing, WarpX measures the wall time for computational parts of the PIC cycle. It then uses this data to decide how to redistribute the subdomains across MPI ranks. (Each subdomain is unchanged, but its owner is changed in order to have better performance.) This relies on each MPI rank handling several (in fact many) subdomains (see max_grid_size).

- **algo.load_balance_efficiency_ratio_threshold** *(float) optional (default 1.1)* Controls whether to adopt a proposed distribution mapping computed during a load balance. If the the ratio of the proposed to current distribution mapping efficiency (i.e., average cost per MPI process; efficiency is a number in the range [0, 1]) is greater than the threshold value, the proposed distribution mapping is adopted. The suggested range of values is algo.load_balance_efficiency_ratio_threshold >= 1, which ensures that the new distribution mapping is adopted only if doing so would improve the load balance efficiency. The higher the threshold value, the more conservative is the criterion for adoption of a proposed distribution; for example, with algo.load_balance_efficiency_ratio_threshold = 1, the proposed distribution is adopted any time the proposed distribution improves load balancing; if instead algo.load_balance_efficiency_ratio_threshold = 2, the proposed distribution is adopted only if doing so would yield a 100% to the load balance efficiency (with this threshold value, if the current efficiency is 0.45, the new distribution would only be adopted if the proposed efficiency were greater than 0.9).

- **algo.load_balance_with_sfc** *(0 or 1) optional (default 0)* If this is 1: use a Space-Filling Curve (SFC) algorithm in order to perform load-balancing of the simulation. If this is 0: the Knapsack algorithm is used instead.

- **algo.load_balance_knapsack_factor** *(float) optional (default 1.24)* Controls the maximum number of boxes that can be assigned to a rank during load balance when using the ‘knapsack’ policy for update of the distribution mapping: the maximum is load_balance_knapsack_factor*(average number of boxes per rank). For example, if there are 4 boxes per rank and load_balance_knapsack_factor=2, no more than 8 boxes can be assigned to any rank.

- **algo.load_balance_costs_update** *(heuristic or timers or gpuclock) optional (default timers)* If this is heuristic: load balance costs are updated according to a measure of particles and cells assigned to each box of the domain. The cost \( c \) is computed as

\[
c = n_{\text{particle}} \cdot w_{\text{particle}} + n_{\text{cell}} \cdot w_{\text{cell}},
\]
where $n_{\text{particle}}$ is the number of particles on the box, $w_{\text{particle}}$ is the particle cost weight factor (controlled by `algo.costs_heuristic_particles_wt`), $n_{\text{cell}}$ is the number of cells on the box, and $w_{\text{cell}}$ is the cell cost weight factor (controlled by `algo.costs_heuristic_cells_wt`).

If this is `timers`: costs are updated according to in-code timers.

If this is `gpuclock`: [requires to compile with option `-DWarpX_GPUCLOCK=ON`] costs are measured as (max-over-threads) time spent in current deposition routine (only applies when running on GPUs).

- `algo.costs_heuristic_particles_wt (float) optional` Particle weight factor used in Heuristic strategy for costs update; if running on GPU, the particle weight is set to a value determined from single-GPU tests on Summit, depending on the choice of solver (FDTD or PSATD) and order of the particle shape. If running on CPU, the default value is $0.9$.

- `algo.costs_heuristic_cells_wt (float) optional` Cell weight factor used in Heuristic strategy for costs update; if running on GPU, the cell weight is set to a value determined from single-GPU tests on Summit, depending on the choice of solver (FDTD or PSATD) and order of the particle shape. If running on CPU, the default value is $0.1$.

- `warpx.do_dynamic_scheduling (0 or 1) optional (default 1)` Whether to activate OpenMP dynamic scheduling.

- `warpx.safe_guard_cells (0 or 1) optional (default 0)` For developers: run in safe mode, exchanging more guard cells, and more often in the PIC loop (for debugging).

### 3.2.7 Math parser and user-defined constants

WarpX uses AMReX’s math parser that reads expressions in the input file. It can be used in all input parameters that consist of one or more integers or floats. Integer input expecting boolean, 0 or 1, are not parsed. Note that when multiple values are expected, the expressions are space delimited. For integer input values, the expressions are evaluated as real numbers and the final result rounded to the nearest integer. See this section of the AMReX documentation for a complete list of functions supported by the math parser.

### WarpX constants

WarpX provides a few pre-defined constants, that can be used for any parameter that consists of one or more floats.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_e</td>
<td>elementary charge</td>
</tr>
<tr>
<td>m_e</td>
<td>electron mass</td>
</tr>
<tr>
<td>m_p</td>
<td>proton mass</td>
</tr>
<tr>
<td>m_u</td>
<td>unified atomic mass unit (Dalton)</td>
</tr>
<tr>
<td>epsilon0</td>
<td>vacuum permittivity</td>
</tr>
<tr>
<td>mu0</td>
<td>vacuum permeability</td>
</tr>
<tr>
<td>clight</td>
<td>speed of light</td>
</tr>
<tr>
<td>kb</td>
<td>Boltzmann’s constant (J/K)</td>
</tr>
<tr>
<td>pi</td>
<td>math constant pi</td>
</tr>
</tbody>
</table>

See `Source/Utils/WarpXConst.H` for the values.
User-defined constants

Users can define their own constants in the input file. These constants can be used for any parameter that consists of one or more integers or floats. User-defined constant names can contain only letters, numbers and the character __. The name of each constant has to begin with a letter. The following names are used by WarpX, and cannot be used as user-defined constants: x, y, z, X, Y, t. The values of the constants can include the predefined WarpX constants listed above as well as other user-defined constants. For example:

- my_constants.a0 = 3.0
- my_constants.z_plateau = 150.e-6
- my_constants.n0 = 1.e22
- my_constants.wp = sqrt(n0*q_e**2/(epsilon0*m_e))

Coordinates

Besides, for profiles that depend on spatial coordinates (the plasma momentum distribution or the laser field, see below Particle initialization and Laser initialization), the parser will interpret some variables as spatial coordinates. These are specified in the input parameter, i.e., density_function(x,y,z) and field_function(X,Y,t).

The parser reads python-style expressions between double quotes, for instance "a0*x**2 * (1-y*1.e2) * (x>0)" is a valid expression where a0 is a user-defined constant (see above) and x and y are spatial coordinates. The names are case sensitive. The factor (x>0) is 1 where x>0 and 0 where x<=0. It allows the user to define functions by intervals. Alternatively the expression above can be written as if(x>0, a0*x**2 * (1-y*1.e2), 0).

3.2.8 Particle initialization

- particles.species_names (strings, separated by spaces) The name of each species. This is then used in the rest of the input deck; in this documentation we use <species_name> as a placeholder.
- particles.use_fdtd.nci_corr (0 or 1) optional (default 0) Whether to activate the FDTD Numerical Cherenkov Instability corrector. Not currently available in the RZ configuration.
- particles.rigid_injected_species (strings, separated by spaces) List of species injected using the rigid injection method. The rigid injection method is useful when injecting a relativistic particle beam, in boosted-frame simulation; see the input-output section for more details. For species injected using this method, particles are translated along the +z axis with constant velocity as long as their z coordinate verifies z<zinject_plane. When z>zinject_plane, particles are pushed in a standard way, using the specified pusher. (see the parameter <species_name>.zinject_plane below)
- particles.do_tiling (bool) optional (default false if WarpX is compiled for GPUs, true otherwise) Controls whether tiling (‘cache blocking’) transformation is used for particles. Tiling should be on when using OpenMP and off when using GPUs.
- <species_name>.species_type (string) optional (default unspecified) Type of physical species. Currently, the accepted species are "electron", "positron", "photon", "hydrogen" (or equivalently "proton"), "helium" (or equivalently "alpha"), "boron", "carbon", "oxygen", "nitrogen", "argon", "copper" and "xenon". Either this or both mass and charge have to be specified.
- <species_name>.charge (float) optional (default NaN) The charge of one physical particle of this species. If species_type is specified, the charge will be set to the physical value and charge is optional. When <species_name>.do_field_ionization = 1, the physical particle charge is equal to ionization_initial_level * charge, so latter parameter should be equal to q_e (which is defined in WarpX as the elementary charge in coulombs).
• `<species_name>.mass (float) optional (default NaN)` The mass of one physical particle of this species. If `species_type` is specified, the mass will be set to the physical value and mass is optional.

• `<species_name>.xmin,ymin,zmin and <species_name>.xmax,ymax,zmax (float) optional (default unlimited)` When `<species_name>.xmin` and `<species_name>.xmax` are set, they delimit the region within which particles are injected. If periodic boundary conditions are used in direction `i`, then the default (i.e. if the range is not specified) range will be the simulation box, `[geometry.prob_hi[i], geometry.prob_lo[i]]`.

• `<species_name>.injection_style (string; default: none)` Determines how the (macro-)particles will be injected in the simulation. The number of particles per cell is always given with respect to the coarsest level (level 0/mother grid), even if particles are immediately assigned to a refined patch.

The options are:

- **NUniformPerCell**: injection with a fixed number of evenly-spaced particles per cell. This requires the additional parameter `<species_name>.num_particles_per_cell_each_dim`.

- **NRandomPerCell**: injection with a fixed number of randomly-distributed particles per cell. This requires the additional parameter `<species_name>.num_particles_per_cell`.

- **SingleParticle**: Inject a single macroparticle. This requires the additional parameters: `<species_name>.single_particle_pos (3 doubles, particle 3D position [meter])`, `<species_name>.single_particle_vel (3 doubles, particle 3D normalized momentum, i.e. $\gamma \beta$)`, `<species_name>.single_particle_weight (double, macroparticle weight, i.e. number of physical particles it represents)`.

- **MultipleParticles**: Inject multiple macroparticles. This requires the additional parameters: `<species_name>.multiple_particles_pos_x (list of doubles, X positions of the particles [meter])`, `<species_name>.multiple_particles_pos_y (list of doubles, Y positions of the particles [meter])`, `<species_name>.multiple_particles_pos_z (list of doubles, Z positions of the particles [meter])`, `<species_name>.multiple_particles_vel_x (list of doubles, X normalized momenta of the particles, i.e. $\gamma \beta_x$)`, `<species_name>.multiple_particles_vel_y (list of doubles, Y normalized momenta of the particles, i.e. $\gamma \beta_y$)`, `<species_name>.multiple_particles_vel_z (list of doubles, Z normalized momenta of the particles, i.e. $\gamma \beta_z$)`, `<species_name>.multiple_particles_weight (list of doubles, macroparticle weights, i.e. number of physical particles each represents)`.

- **gaussian_beam**: Inject particle beam with gaussian distribution in space in all directions. This requires additional parameters: `<species_name>.q_tot (beam charge) optional (default is q_tot=0)`, `<species_name>.npart (number of particles in the beam)`, `<species_name>.x/y/z_m (average position in x/y/z)`, `<species_name>.x/y/z_rms (standard deviation in x/y/z)`, `<species_name>.x/y/z_cut (optional, particles with abs(x-x_m) > x_cut*x_rms are not injected, same for y and z)`, `<species_name>.q_tot` is the charge of the un-cut beam, so that cutting the distribution is likely to result in a lower total charge), and optional argument `<species_name>.do_symmetrize (whether to symmetrize the beam in the x and y directions)`.

- **external_file**: Inject macroparticles with properties (mass, charge, position, and momentum - $\gamma \beta mc$) read from an external openPMD file. With it users can specify the additional arguments: `<species_name>.injection_file (string) openPMD file name and `<species_name>.q_tot (double) optional (default is q_tot=0 and no re-scaling is done, weight=q_p) when specified it is used to re-scale the weight of externally loaded N physical particles, each of charge q_p, to inject macroparticles of weight=q_p*N`. `<species_name>.charge (double)` optional (default is read from openPMD file) when set this will be the charge of the physical particle represented by the injected macroparticles. `<species_name>.mass (double) optional (default is read from openPMD file) when set this will be the charge of the physical particle represented by the injected macroparticles. `<species_name>.z_shift (double) optional (default is no shift) when set
this value will be added to the longitudinal, z, position of the particles. The external file must include
the species openPMD::Record labeled position and momentum (double arrays), with dimensionality
and units set via openPMD::setUnitDimension and setUnitSI. If the external file also contains
openPMD::Records for mass and charge (constant double scalars) then the species will use
these, unless overwritten in the input file (see "species_name".mass, "species_name".charge
or "species_name".species_type). The external_file option is currently implemented for
2D, 3D and RZ geometries, with record components in the cartesian coordinates (x,y,z) for 3D and
RZ, and (x,z) for 2D. For more information on the openPMD format and how to build WarpX with
it, please visit the install section.

- NFluxPerCell: Continuously inject a flux of macroparticles from a planar surface. The density
specified by the density profile is interpreted to have the units of #/m^2/s. This requires the additional parameters: "species_name".surface_flux_pos (double, location of the injection plane [meter]) "species_name".flux_normal_axis (x, y, or z for 3D, x or z for 2D, or r, t, or z for RZ. When flux_normal_axis is r or t, the x and y components of the user-specified momentum distribution are interpreted as the r and t components respectively) "species_name".flux_direction (-1 or +1, direction of flux relative to the plane) "species_name".num_particles_per_cell (double) "species_name".flux_tmin (double, Optional time at which the flux will be turned on. Ignored when negative.) "species_name".flux_tmax (double, Optional time at which the flux will be turned off. Ignored when negative.)

- none: Do not inject macro-particles (for example, in a simulation that starts with neutral, ionizable
atoms, one may want to create the electrons species – where ionized electrons can be stored later on –
without injecting electron macro-particles).

- <species_name>.num_particles_per_cell_each_dim (3 integers in 3D and RZ, 2 integers in 2D)
  With the NUniformPerCell injection style, this specifies the number of particles along each axis within a
cell. Note that for RZ, the three axis are radius, theta, and z and that the recommended number of particles
per theta is at least two times the number of azimuthal modes requested. (It is recommended to do a
convergence scan of the number of particles per theta)

- <species_name>.random_theta (bool) optional (default 1) When using RZ geometry, whether to random-
ize the azimuthal position of particles. This is used when "species_name".injection_style =
NUniformPerCell.

- <species_name>.do_splitting (bool) optional (default 0) Split particles of the species when crossing the
boundary from a lower resolution domain to a higher resolution domain.

Currently implemented on CPU only.

- <species_name>.do_continuous_injection (0 or 1) Whether to inject particles during the simulation,
and not only at initialization. This can be required with a moving window and/or when running in a boosted
frame.

- <species_name>.initialize_self_fields (0 or 1) Whether to calculate the space-charge fields associ-
ated with this species at the beginning of the simulation. The fields are calculated for the mean gamma of
the species.

- <species_name>.self_fields_required_precision (float, default: 1.e-11) The relative precision with
which the initial space-charge fields should be calculated. More specifically, the initial space-charge fields
are computed with an iterative Multi-Level Multi-Grid (MLMG) solver. For highly-relativistic beams, this
solver can fail to reach the default precision within a reasonable time; in that case, users can set a relaxed
precision requirement through self_fields_required_precision.

- <species_name>.self_fields_absolute_tolerance (float, default: 0.0) The absolute tolerance with
which the space-charge fields should be calculated in units of V/m^2. More specifically, the acceptable
residual with which the solution can be considered converged. In general this should be left as the default,
but in cases where the simulation state changes very little between steps it can occur that the initial guess
for the MLMG solver is so close to the converged value that it fails to improve that solution sufficiently to reach the self_fields_required_precision value.

- `<species_name>.self_fields_max_iters` *(integer, default: 200)* Maximum number of iterations used for MLMG solver for initial space-charge fields calculation. In case if MLMG converges but fails to reach the desired self_fields_required_precision, this parameter may be increased.

- `<species_name>.profile` *(string)* Density profile for this species. The options are:
  - constant: Constant density profile within the box, or between `<species_name>.xmin` and `<species_name>.xmax` (and same in all directions). This requires additional parameter `<species_name>.density`, i.e., the plasma density in $m^{-3}$.
  - predefined: Predefined density profile. This requires additional parameters `<species_name>.predefined_profile_name` and `<species_name>.predefined_profile_params`. Currently, only a parabolic channel density profile is implemented.
  - parse_density_function: the density is given by a function in the input file. It requires additional argument `<species_name>.density_function(x,y,z)`, which is a mathematical expression for the density of the species, e.g., `electrons.density_function(x,y,z) = "n0+n0*x^2*1.e12"` where $n0$ is a user-defined constant, see above. WARNING: where `density_function(x,y,z)` is close to zero, particles will still be injected between `xmin` and `xmax` etc., with a null weight. This is undesirable because it results in useless computing. To avoid this, see option density_min below.

- `<species_name>.density_min` *(float)* optional (default 0.) Minimum plasma density. No particle is injected where the density is below this value.

- `<species_name>.density_max` *(float)* optional (default infinity) Maximum plasma density. The density at each point is the minimum between the value given in the profile, and density_max.

- `<species_name>.radially_weighted` *(bool)* optional (default true) Whether particle's weight is varied with their radius. This only applies to cylindrical geometry. The only valid value is true.
  - predefined: use one of WarpX predefined plasma profiles. It requires additional arguments `<species_name>.predefined_profile_name` and `<species_name>.predefined_profile_params` (see below).

- `<species_name>.momentum_distribution_type` *(string)* Distribution of the normalized momentum ($u=p/mc$) for this species. The options are:
  - at_rest: Particles are initialized with zero momentum.
  - constant: constant momentum profile. This can be controlled with the additional parameters `<species_name>.ux`, `<species_name>.uy` and `<species_name>.uz`, the normalized momenta in the x, y and z direction respectively, which are all 0. by default.
  - gaussian: gaussian momentum distribution in all 3 directions. This can be controlled with the additional arguments for the average momenta along each direction `<species_name>.ux_m`, `<species_name>.uy_m` and `<species_name>.uz_m` as well as standard deviations along each direction `<species_name>.ux_th`, `<species_name>.uy_th` and `<species_name>.uz_th`. These 6 parameters are all 0. by default.
  - gaussianflux: Gaussian momentum flux distribution, which is Gaussian in the plane and v^*Gaussian normal to the plane. It can only be used when injection_style = NFluxPerCell. This can be controlled with the additional arguments to specify the plane's orientation, `<species_name>.flux_normal_axis` and `<species_name>.flux_direction`, for the average momenta along each direction `<species_name>.ux_m`, `<species_name>.uy_m` and `<species_name>.uz_m` as well as standard deviations along each direction `<species_name>.ux_th`, `<species_name>.uy_th` and `<species_name>.uz_th`. Note that the average momenta normal to the plane is not used. `<species_name>.ux_m`, `<species_name>.uy_m`, `<species_name>.uz_m`, `<species_name>.ux_th`, `<species_name>.uy_th` and `<species_name>.uz_th` are all 0. by default.
maxwell_boltzmann: Maxwell-Boltzmann distribution that takes a dimensionless temperature parameter \( \theta \) as an input, where \( \theta = \frac{k_B T}{m c^2} \), "math: T is the temperature in Kelvin, \( k_B \) is the Boltzmann constant, \( c \) is the speed of light, and \( m \) is the mass of the species. Theta is specified by a combination of \(<\text{species_name}.theta_distribution_type, \text{species_name}.theta, and <\text{species_name}.theta_function(x,y,z) (see below). For values of \( \theta > 0.01 \), errors due to ignored relativistic terms exceed 1%. Temperatures less than zero are not allowed. The plasma can be initialized to move at a bulk velocity \( \beta = v/c \). The speed is specified by the parameters <\text{species_name}.beta_distribution_type, \text{species_name}.beta, and <\text{species_name}.beta_function(x,y,z) (see below). \( \beta \) can be positive or negative and is limited to the range \( -1 < \beta < 1 \). The direction of the velocity field is given by <\text{species_name}.bulk_vel_dir = (+/-) 'x', 'y', 'z', and must be the same across the domain. Please leave no whitespace between the sign and the character on input. A direction without a sign will be treated as positive. The MB distribution is initialized in the drifting frame by sampling three Gaussian distributions in each dimension using, the Box Mueller method, and then the distribution is transformed to the simulation frame using the flipping method. The flipping method can be found in Zenitani 2015 section III. B. (Phys. Plasmas 22, 042116). By default, \( \beta \) is equal to 0. and bulk_vel_dir is +x.

Note that though the particles may move at relativistic speeds in the simulation frame, they are not relativistic in the drift frame. This is as opposed to the Maxwell Juttner setting, which initializes particles with relativistic momentums in their drifting frame.

maxwell_juttner: Maxwell-Juttner distribution for high temperature plasma that takes a dimensionless temperature parameter \( \theta \) as an input, where \( \theta = \frac{k_B T}{m c^2} \), "math: T is the temperature in Kelvin, \( k_B \) is the Boltzmann constant, \( m \) is the mass of the species. Theta is specified by a combination of <\text{species_name}.theta_distribution_type, \text{species_name}.theta, and <\text{species_name}.theta_function(x,y,z) (see below). The Sobol method used to generate the distribution will not terminate for \( \theta \lesssim 0.1 \), and the code will abort if it encounters a temperature below that threshold. The Maxwell-Boltzmann distribution is recommended for temperatures in the range \( 0.01 < \theta < 0.1 \). Errors due to relativistic effects can be expected to approximately between 1% and 10%. The plasma can be initialized to move at a bulk velocity \( \beta = v/c \). The speed is specified by the parameters <\text{species_name}.beta_distribution_type, \text{species_name}.beta, and <\text{species_name}.beta_function(x,y,z) (see below). \( \beta \) can be positive or negative and is limited to the range \( -1 < \beta < 1 \). The direction of the velocity field is given by <\text{species_name}.bulk_vel_dir = (+/-) 'x', 'y', 'z', and must be the same across the domain. Please leave no whitespace between the sign and the character on input. A direction without a sign will be treated as positive. The MJ distribution will be initialized in the moving frame using the Sobol method, and then the distribution will be transformed to the simulation frame using the flipping method. Both the Sobol and the flipping method can be found in Zenitani 2015 (Phys. Plasmas 22, 042116). By default, \( \beta \) is equal to 0. and bulk_vel_dir is +x.

Please take notice that particles initialized with this setting can be relativistic in two ways. In the simulation frame, they can drift with a relativistic speed \( \beta \). Then, in the drifting frame they are still moving with relativistic speeds due to high temperature. This is as opposed to the Maxwell Boltzmann setting, which initializes non-relativistic plasma in their relativistic drifting frame.

radial_expansion: momentum depends on the radial coordinate linearly. This can be controlled with additional parameter u_over_r which is the slope (0. by default).

parse_momentum_function: the momentum is given by a function in the input file. It requires additional arguments <\text{species_name}.momentum_function_ux(x,y,z), <\text{species_name}.momentum_function_uy(x,y,z) and <\text{species_name}.momentum_function_uz(x,y,z), which gives the distribution of each component of the momentum as a function of space.

• <\text{species_name}.theta_distribution_type (string) optional (default constant) Only read if <\text{species_name}.momentum_distribution_type is maxwell_boltzmann or maxwell_juttner. See documentation for these distributions (above) for constraints on values of theta. Temperatures less than zero are not allowed.
- If constant, use a constant temperature, given by the required float parameter `<species_name>`.theta.

- If parser, use a spatially-dependent analytic parser function, given by the required parameter `<species_name>`.theta_function(x,y,z).

• `<species_name>`.beta_distribution_type (string) optional (default constant) Only read if `<species_name>`.momentum_distribution_type is maxwell_boltzmann or maxwell_juttner. See documentation for these distributions (above) for constraints on values of beta.

- If constant, use a constant speed, given by the required float parameter `<species_name>`.beta.

- If parser, use a spatially-dependent analytic parser function, given by the required parameter `<species_name>`.beta_function(x,y,z).

• `<species_name>`.zinject_plane (float) Only read if `<species_name>` is in particles. rigid_injected_species. Injection plane when using the rigid injection method. See particles. rigid_injected_species above.

• `<species_name>`.rigid_advance (bool) Only read if `<species_name>` is in particles. rigid_injected_species.

  - If false, each particle is advanced with its own velocity vz until it reaches zinject_plane.

  - If true, each particle is advanced with the average speed of the species vzbar until it reaches zinject_plane.

• `<species_name>`.predefined_profile_name (string) Only read if `<species_name>`.profile is predefined.

  - If parabolic_channel, the plasma profile is a parabolic profile with cosine-like ramps at the beginning and the end of the profile. The density is given by

    \[ n = n_0 n(x,y) n(z - z_0) \]

    with

    \[ n(x,y) = 1 + 4 \frac{x^2 + y^2}{k_p^2 R_c^4} \]

    where \( k_p \) is the plasma wavenumber associated with density \( n_0 \). Here, with \( z_0 \) as the start of the plasma, \( n(z - z_0) \) is a cosine-like up-ramp from 0 to \( L_{ramp,up} \), constant to 1 from \( L_{ramp,up} \) to \( L_{ramp,up} + L_{plateau} \) and a cosine-like down-ramp from \( L_{ramp,up} + L_{plateau} \) to \( L_{ramp,up} + L_{plateau} + L_{ramp,down} \). All parameters are given in predefined_profile_params.

• `<species_name>`.predefined_profile_params (list of float) Parameters for the predefined profiles.

  - If `species_name`.predefined_profile_name is parabolic_channel, predefined_profile_params contains a space-separated list of the following parameters, in this order: \( z_0 L_{ramp,up} L_{plateau} L_{ramp,down} R_c n_0 \)

• `<species_name>`.do_backward_propagation (bool) Inject a backward-propagating beam to reduce the effect of charge-separation fields when running in the boosted frame. See examples.

• `<species_name>`.split_type (int) optional (default 0) Splitting technique. When 0, particles are split along the simulation axes (4 particles in 2D, 6 particles in 3D). When 1, particles are split along the diagonals (4 particles in 2D, 8 particles in 3D).

• `<species_name>`.do_not_deposit (0 or 1 optional; default 0) If 1 is given, both charge deposition and current deposition will not be done, thus that species does not contribute to the fields.

• `<species_name>`.do_not_gather (0 or 1 optional; default 0) If 1 is given, field gather from grids will not be done, thus that species will not be affected by the field on grids.
- `<species_name>.do_not_push (θ or l optional; default 0)` If θ is given, this species will not be pushed by any pusher during the simulation.

- `<species_name>.addIntegerAttributes (list of string)` User-defined integer particle attribute for species, `species_name`. These integer attributes will be initialized with user-defined functions when the particles are generated. If the user-defined integer attribute is `<int_attrib_name>` then the following required parameter must be specified to initialize the attribute. * `<species_name>.attribute.<int_attrib_name>(x,y,z,ux,uy,uz,t)` denotes `int_attrib_name` as a function of the physical positions in the unit of meter. `ux`, `uy`, `uz` represent the particle velocities in the unit of `γν/𝑐`, where `γ` is the Lorentz factor, `ν/𝑐` is the particle velocity normalized by the speed of light. E.g. If `electrons.addIntegerAttributes = upstream` and `electrons.upstream(x,y,z,ux,uy,uz,t) = (x>0) * 1` is provided then, an integer attribute `upstream` is added to all electron particles and when these particles are generated, the particles with position less than 0 are assigned a value of 1.

- `<species_name>.addRealAttributes (list of string)` User-defined real particle attribute for species, `species_name`. These real attributes will be initialized with user-defined functions when the particles are generated. If the user-defined real attribute is `<real_attrib_name>` then the following required parameter must be specified to initialize the attribute. * `<species_name>.attribute.<real_attrib_name>(x,y,z,ux,uy,uz,t)` denotes `real_attrib_name` as a function of the physical positions in the unit of meter. `ux`, `uy`, `uz` represent the particle velocities in the unit of `γν/𝑐`, where `γ` is the Lorentz factor, `ν/𝑐` is the particle velocity normalized by the speed of light.

- `<species>.save_particles_at_xlo/ylo/zlo, <species>.save_particles_at_xhi/yhi/zhi and <species>.save_particles_at_eb (optional)` If θ particles of this species will be copied to the scraped particle buffer for the specified boundary if they leave the simulation domain in the specified direction. If `USE_EB=TRUE` the `save_particles_at_eb` flag can be set to 1 to also save particle data for the particles of this species that impact the embedded boundary. The scraped particle buffer can be used to track particle fluxes out of the simulation but is currently only accessible via the Python interface. The function `get_particle_boundary_buffer`, found in the `picmi.Simulation` class as `sim.extension.get_particle_boundary_buffer()`, can be used to access the scraped particle buffer. An entry is included for every particle in the buffer of the timestep at which the particle was scraped. This can be accessed by passing the argument `comp_name="step_scraped"` to the above mentioned function.

Note: Currently the scraped particle buffer relies on the user to access the data in the buffer for processing and periodically clear the buffer. The buffer will grow unbounded as particles are scraped and therefore could lead to memory issues if not periodically cleared. To clear the buffer call `warpx_clearParticleBoundaryBuffer()`.

- `<species>.do_back_transformed_diagnostics (0 or l optional, default l)` Only used when `warpx.do_back_transformed_diagnostics=1`. When running in a boosted frame, whether or not to plot back-transformed diagnostics for this species.

- `warpx.serialize_initial_conditions (0 or l optional, default 0)` Serialize the initial conditions for reproducible testing. Mainly whether or not to use OpenMP threading for particle initialization.

- `<species>.do_field_ionization (0 or l optional, default 0)` Do field ionization for this species (using the ADK theory).

- `<species>.physical_element (string)` Only read if `do_field_ionization = 1`. Symbol of chemical element for this species. Example: for Helium, use `physical_element = He`. Elements up to atomic number `Z=86` (Radon) are supported, let us know if you need higher `Z`.

- `<species>.ionization_product_species (string)` Only read if `do_field_ionization = 1`. Name of species in which ionized electrons are stored. This species must be created as a regular species in the input file (in
particular, it must be in particles.species_names).

- `<species>.ionization_initial_level (int) optional (default 0)` Only read if `do_field_ionization = 1`. Initial ionization level of the species (must be smaller than the atomic number of chemical element given in physical_element).

- `<species>.do_classical_radiation_reaction (int) optional (default 0)` Enables Radiation Reaction (or Radiation Friction) for the species. Species must be either electrons or positrons. Boris pusher must be used for the simulation.

- `<species>.do_qed_quantum_sync (int) optional (default 0)` Enables Quantum synchrotron emission for this species. Quantum synchrotron lookup table should be either generated or loaded from disk to enable this process (see “Lookup tables for QED modules” section below). `<species>` must be either an electron or a positron species. **This feature requires to compile with QED=TRUE**

- `<species>.do_qed_breit_wheeler (int) optional (default 0)` Enables non-linear Breit-Wheeler process for this species. Breit-Wheeler lookup table should be either generated or loaded from disk to enable this process (see “Lookup tables for QED modules” section below). `<species>` must be a photon species. **This feature requires to compile with QED=TRUE**

- `<species>.qed_quantum_sync_phot_product_species (string)` If an electron or a positron species has the Quantum synchrotron process, a photon product species must be specified (the name of an existing photon species must be provided) **This feature requires to compile with QED=TRUE**

- `<species>.qed_breit_wheeler_ele_product_species (string)` If a photon species has the Breit-Wheeler process, an electron product species must be specified (the name of an existing electron species must be provided) **This feature requires to compile with QED=TRUE**

- `<species>.qed_breit_wheeler_pos_product_species (string)` If a photon species has the Breit-Wheeler process, a positron product species must be specified (the name of an existing positron species must be provided) **This feature requires to compile with QED=TRUE**

- `<species>.do_resampling (0 or 1) optional (default 0)` If 1 resampling is performed for this species. This means that the number of macroparticles will be reduced at specific timesteps while preserving the distribution function as much as possible (in particular the weight of the remaining particles will be increased on average). This can be useful in situations with continuous creation of particles (e.g. with ionization or with QED effects). At least one resampling trigger (see below) must be specified to actually perform resampling.

- `<species>.resampling_algorithm (string) optional (default leveling_thinning)` The algorithm used for resampling. Currently there is only one option, which is already set by default:
  
  - leveling_thinning This algorithm is defined in Muraviev et al., arXiv:2006.08593 (2020). It has two parameters:
    
    * `<species>.resampling_algorithm_target_ratio (float) optional (default 1.5)` This roughly corresponds to the ratio between the number of particles before and after resampling.
    
    * `<species>.resampling_algorithm_min_ppc (int) optional (default 1)` Resampling is not performed in cells with a number of macroparticles strictly smaller than this parameter.

- `<species>.resampling_trigger_intervals (string) optional (default 0)` Using the *Intervals* parser syntax, this string defines timesteps at which resampling is performed.

- `<species>.resampling_trigger_max_avg_ppc (float) optional (default infinity)` Resampling is performed everytime the number of macroparticles per cell of the species averaged over the whole simulation domain exceeds this parameter.
3.2.9 Laser initialization

- **lasers.names (list of string)** Name of each laser. This is then used in the rest of the input deck; in this documentation we use `<laser_name>` as a placeholder. The parameters below must be provided for each laser pulse.

- **<laser_name>.position (3 floats in 3D and 2D; in meters)** The coordinates of one of the point of the antenna that will emit the laser. The plane of the antenna is entirely defined by `<laser_name>.position` and `<laser_name>.direction`.

  `<laser_name>.position` also corresponds to the origin of the coordinates system for the laser transverse profile. For instance, for a Gaussian laser profile, the peak of intensity will be at the position given by `<laser_name>.position`. This variable can thus be used to shift the position of the laser pulse transversally.

  **Note:** In 2D, `<laser_name>.position` is still given by 3 numbers, but the second number is ignored.

When running a **boosted-frame simulation**, provide the value of `<laser_name>.position` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame.

- **<laser_name>.polarization (3 floats in 3D and 2D)** The coordinates of a vector that points in the direction of polarization of the laser. The norm of this vector is unimportant, only its direction matters.

  **Note:** Even in 2D, all the 3 components of this vectors are important (i.e. the polarization can be orthogonal to the plane of the simulation).

- **<laser_name>.direction (3 floats in 3D)** The coordinates of a vector that points in the propagation direction of the laser. The norm of this vector is unimportant, only its direction matters.

  The plane of the antenna that will emit the laser is orthogonal to this vector.

  **Warning:** When running **boosted-frame simulations**, `<laser_name>.direction` should be parallel to `warpx.boost_direction`, for now.

- **<laser_name>.e_max (float; in V/m)** Peak amplitude of the laser field.

  For a laser with a wavelength $\lambda = 0.8 \, \mu m$, the peak amplitude is related to $a_0$ by:

  $E_{\text{max}} = a_0 \frac{2\pi n_e c^2}{e \lambda} = a_0 \times (4.0 \cdot 10^{12} \, V.m^{-1})$

  When running a **boosted-frame simulation**, provide the value of `<laser_name>.e_max` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame.

- **<laser_name>.a0 (float; dimensionless)** Peak normalized amplitude of the laser field (given in the lab frame, just as e_max above). See the description of `<laser_name>.e_max` for the conversion between a0 and e_max. Exactly one of a0 and e_max must be specified.

- **<laser_name>.wavelength (float; in meters)** The wavelength of the laser in vacuum.

  When running a **boosted-frame simulation**, provide the value of `<laser_name>.wavelength` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame.
• `<laser_name>.profile` *(string)* The spatio-temporal shape of the laser. The options that are currently implemented are:
  - "Gaussian": The transverse and longitudinal profiles are Gaussian.
  - "Harris": The transverse profile is Gaussian, but the longitudinal profile is given by the Harris function (see `<laser_name>.profile_duration` for more details)
  - "parse_field_function": the laser electric field is given by a function in the input file. It requires additional argument `<laser_name>.field_function(X,Y,t)`, which is a mathematical expression, e.g. `<laser_name>.field_function(X,Y,t) = "a0*X**2 * (X>0) * cos(omega0*t)"` where `a0` and `omega0` are user-defined constant, see above. The profile passed here is the full profile, not only the laser envelope. `t` is time and `X` and `Y` are coordinates orthogonal to `<laser_name>.direction` (not necessarily the x and y coordinates of the simulation). All parameters above are required, but none of the parameters below are used when `<laser_name>.parse_field_function=1`. Even though `<laser_name>.wavelength` and `<laser_name>.e_max` should be included in the laser function, they still have to be specified as they are used for numerical purposes.
  - "from_txye_file": the electric field of the laser is read from an external binary file whose format is explained below. It requires to provide the name of the binary file setting the additional parameter `<laser_name>.txye_file_name` *(string)*. It accepts an optional parameter `<laser_name>.time_chunk_size` *(int)*. This allows to read only time_chunk_size timesteps from the binary file. New timesteps are read as soon as they are needed. The default value is automatically set to the number of timesteps contained in the binary file (i.e. only one read is performed at the beginning of the simulation). It also accepts the optional parameter `<laser_name>.delay` *(float; in seconds)*, which allows delaying (delay > 0) or anticipating (delay < 0) the laser by the specified amount of time. The external binary file should provide E(x,y,t) on a rectangular (but non necessarily uniform) grid. The code performs a bi-linear (in 2D) or tri-linear (in 3D) interpolation to set the field values. x,y,t are meant to be in S.I. units, while the field value is meant to be multiplied by `<laser_name>.e_max` (i.e. in most cases the maximum of abs(E(x,y,t)) should be 1, so that the maximum field intensity can be set straightforwardly with `<laser_name>.e_max`). The binary file has to respect the following format:

  * flag to indicate if the grid is uniform or not (1 byte, 0 means non-uniform, !=0 means uniform)
  * np, number of timesteps `(uint32_t, must be >=2)`
  * nx, number of points along x `(uint32_t, must be >=2)`
  * ny, number of points along y `(uint32_t, must be 1 for 2D simulations and >=2 for 3D simulations)`
  * timesteps `(double[2] if grid is uniform, double[np] otherwise)`
  * x_coords `(double[2] if grid is uniform, double[nx] otherwise)`
  * field_data `(double[nt * nx * ny], with nt being the slowest coordinate)`.

  A file at this format can be generated from Python, see an example at Examples/Modules/laser_injection_from_file

• `<laser_name>.profile_t_peak` *(float; in seconds)* The time at which the laser reaches its peak intensity, at the position given by `<laser_name>.position` (only used for the "gaussian" profile)

  When running a boosted-frame simulation, provide the value of `<laser_name>.profile_t_peak` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame.

• `<laser_name>.profile_duration` *(float; in seconds)* The duration of the laser pulse, defined as $\tau$ below:

  - For the "gaussian" profile:
\[ E(x, t) \propto \exp \left( -\frac{(t - t_{\text{peak}})^2}{\tau^2} \right) \]

Note that \( \tau \) relates to the full width at half maximum (FWHM) of intensity, which is closer to pulse length measurements in experiments, as \( \tau = \text{FWHM}_1/\sqrt{2\ln(2)} \approx \text{FWHM}_1/1.174 \).

- For the "harris" profile:

\[ E(x, t) \propto \frac{1}{32} \left[ 10 - 15 \cos \left( \frac{2\pi t}{\tau} \right) + 6 \cos \left( \frac{4\pi t}{\tau} \right) - \cos \left( \frac{6\pi t}{\tau} \right) \right] \Theta(\tau - t) \]

When running a boosted-frame simulation, provide the value of \(<laser_name>.profile_duration\) in the laboratory frame, and use \texttt{warpx.gamma_boost} to automatically perform the conversion to the boosted frame.

- \(<laser_name>.profile_waist (float ; in meters)\) The waist of the transverse Gaussian laser profile, defined as \( w_0 \):

\[ E(x, t) \propto \exp \left( -\frac{x^2}{w_0^2} \right) \]

- \(<laser_name>.profile_focal_distance (float ; in meters)\) The distance from laser_position to the focal plane. (where the distance is defined along the direction given by \(<laser_name>.direction).\)

Use a negative number for a defocussing laser instead of a focussing laser.

When running a boosted-frame simulation, provide the value of \(<laser_name>.profile_focal_distance\) in the laboratory frame, and use \texttt{warpx.gamma_boost} to automatically perform the conversion to the boosted frame.

- \(<laser_name>.phi0 (float ; in radians) optional (default 0.)\) The Carrier Envelope Phase, i.e. the phase of the laser oscillation, at the position where the laser envelope is maximum (only used for the "gaussian" profile)

- \(<laser_name>.stc_direction (3 floats) optional (default 1. 0. 0.)\) Direction of laser spatio-temporal couplings. See definition in Akturk et al., Opt Express, vol 12, no 19 (2004).

- \(<laser_name>.zeta (float ; in meters.seconds) optional (default 0.)\) Spatial chirp at focus in direction \(<laser_name>.stc_direction.\) See definition in Akturk et al., Opt Express, vol 12, no 19 (2004).

- \(<laser_name>.beta (float ; in seconds) optional (default 0.)\) Angular dispersion (or angular chirp) at focus in direction \(<laser_name>.stc_direction.\) See definition in Akturk et al., Opt Express, vol 12, no 19 (2004).

- \(<laser_name>.phi2 (float ; in seconds**2) optional (default 0.)\) Temporal chirp at focus. See definition in Akturk et al., Opt Express, vol 12, no 19 (2004).

- \(<laser_name>.do_continuous_injection (0 or 1) optional (default 0).\) Whether or not to use continuous injection. If the antenna starts outside of the simulation domain but enters it at some point (due to moving window or moving antenna in the boosted frame), use this so that the laser antenna is injected when it reaches the box boundary. If running in a boosted frame, this requires the boost direction, moving window direction and laser propagation direction to be along \( z \). If not running in a boosted frame, this requires the moving window and laser propagation directions to be the same (\( x, y \) or \( z \))

- \(<laser_name>.min_particles_per_mode (int) optional (default 4)\) When using the RZ version, this specifies the minimum number of particles per angular mode. The laser particles are loaded into radial spokes, with the number of spokes given by \texttt{min_particles_per_mode*(warpx.n_rz_azimuthal_modes-1)}.\)

- \texttt{warpx.num_mirrors (int) optional (default 0)} Users can input perfect mirror condition inside the simulation domain. The number of mirrors is given by \texttt{warpx.num_mirrors.} The mirrors are orthogonal to the \( z \) direction. The following parameters are required when \texttt{warpx.num_mirrors} is \( >0 \).

- \texttt{warpx.mirror_z (list of float) required if warpx.num_mirrors>0} \( z \) location of the front of the mirrors.
• \texttt{warpx.mirror\_z\_width} (list of float) required if \texttt{warpx.num\_mirrors}>0 \quad \text{z width of the mirrors.}

• \texttt{warpx.mirror\_z\_npoints} (list of int) required if \texttt{warpx.num\_mirrors}>0 \quad \text{In the boosted frame, depending on \texttt{gamma\_boost}, \texttt{warpx.mirror\_z\_width} can be smaller than the cell size, so that the mirror would not work. This parameter is the minimum number of points for the mirror. If \texttt{mirror\_z\_width} < \texttt{dz/cell\_size}, the upper bound of the mirror is increased so that it contains at least \texttt{mirror\_z\_npoints}.}

3.2.10 External fields

• \texttt{warpx.B\_ext\_grid\_init\_style} (string) optional (default is “default”) \quad \text{This parameter determines the type of initialization for the external magnetic field. The “default” style initializes the external magnetic field \((B_x,B_y,B_z)\) to \((0.0, 0.0, 0.0)\). The string can be set to “constant” if a constant magnetic field is required to be set at initialization. If set to “constant”, then an additional parameter, namely, \texttt{warpx.B\_external\_grid\_must\_be\_specified} must be specified. If set to \texttt{parse\_B\_ext\_grid\_function}, then a mathematical expression can be used to initialize the external magnetic field on the grid. It requires additional parameters in the input file, namely, \texttt{warpx.Bx\_external\_grid\_function(x,y,z)}, \texttt{warpx.By\_external\_grid\_function(x,y,z)}, \texttt{warpx.Bz\_external\_grid\_function(x,y,z)} to initialize the external magnetic field for each of the three components on the grid. Constants required in the expression can be set using \texttt{my\_constants}. For example, if \texttt{warpx.Bx\_external\_grid\_function(x,y,z)} \(= Bo*x + delta*(y + z)\) then the constants \(Bo\) and \(delta\) required in the above equation can be set using \texttt{my\_constants.Bo} and \texttt{my\_constants.delta} in the input file. For a two-dimensional simulation, it is assumed that the first dimension is \(x\) and the second dimension is \(z\), and the value of \(y\) is set to zero. Note that the current implementation of the parser for external B-field does not work with RZ and the code will abort with an error message.}

• \texttt{warpx.E\_ext\_grid\_init\_style} (string) optional (default is “default”) \quad \text{This parameter determines the type of initialization for the external electric field. The “default” style initializes the external electric field \((E_x,E_y,E_z)\) to \((0.0, 0.0, 0.0)\). The string can be set to “constant” if a constant electric field is required to be set at initialization. If set to “constant”, then an additional parameter, namely, \texttt{warpx.E\_external\_grid\_must\_be\_specified} must be specified in the input file. If set to \texttt{parse\_E\_ext\_grid\_function}, then a mathematical expression can be used to initialize the external electric field on the grid. It required additional parameters in the input file, namely, \texttt{warpx.Ex\_external\_grid\_function(x,y,z)}, \texttt{warpx.Ey\_external\_grid\_function(x,y,z)}, \texttt{warpx.Ez\_external\_grid\_function(x,y,z)} to initialize the external electric field for each of the three components on the grid. Constants required in the expression can be set using \texttt{my\_constants}. For example, if \texttt{warpx.Ex\_external\_grid\_function(x,y,z)} \(= Eo*x + delta*(y + z)\) then the constants \(Eo\) and \(delta\) required in the above equation can be set using \texttt{my\_constants.Eo} and \texttt{my\_constants.delta} in the input file. For a two-dimensional simulation, it is assumed that the first dimension is \(x\) and the second dimension is \(z\), and the value of \(y\) is set to zero. Note that the current implementation of the parser for external E-field does not work with RZ and the code will abort with an error message.}

• \texttt{warpx.E\_external\_grid} & \texttt{warpx.B\_external\_grid} (list of 3 floats) required when \texttt{warpx.E\_ext\_grid\_init\_style}="constant" and when \texttt{warpx.B\_ext\_grid\_init\_style}="constant", respectively. \text{External uniform and constant electrostatic and magnetostatic field added to the grid at initialization. Use with caution as these fields are used for the field solver. In particular, do not use any other boundary condition than periodic.}

• \texttt{particles.E\_ext\_particle\_init\_style} & \texttt{particles.B\_ext\_particle\_init\_style} (string) optional (default “none”) \quad \text{These parameters determine the type of the external electric and magnetic fields respectively that are applied directly to the particles at every timestep. The field values are specified in the lab frame. With the default none style, no field is applied. Possible values are constant, \texttt{parse\_E\_ext\_particle\_function} or \texttt{parse\_B\_ext\_particle\_function}, or \texttt{repeated\_plasma\_lens}.

  - constant: a constant field is applied, given by the input parameters \texttt{particles.E\_external\_particle} or \texttt{particles.B\_external\_particle}, which are lists of the field components.

3.2. Input Parameters
- parse_E_ext_particle_function or parse_B_ext_particle_function: the field is specified as an analytic expression that is a function of space \((x,y,z)\) and time \((t)\), relative to the lab frame. The E-field is specified by the input parameters:

  * particles.Ex_external_particle_function\((x,y,z,t)\)
  * particles.Ey_external_particle_function\((x,y,z,t)\)
  * particles.Ez_external_particle_function\((x,y,z,t)\)

The B-field is specified by the input parameters:

  * particles.Bx_external_particle_function\((x,y,z,t)\)
  * particles.By_external_particle_function\((x,y,z,t)\)
  * particles.Bz_external_particle_function\((x,y,z,t)\)

Note that the position is defined in Cartesian coordinates, as a function of \((x,y,z)\), even for RZ.

- repeated_plasma_lens: apply a series of plasma lenses. The properties of the lenses are defined in the lab frame by the input parameters:

  * repeated_plasma_lens_period, the period length of the repeat, a single float number,
  * repeated_plasma_lens_starts, the start of each lens relative to the period, an array of floats,
  * repeated_plasma_lens_lengths, the length of each lens, an array of floats,
  * repeated_plasma_lens_strengths_E, the electric focusing strength of each lens, an array of floats, when particles.E_ext_particle_init_style is set to repeated_plasma_lens.
  * repeated_plasma_lens_strengths_B, the magnetic focusing strength of each lens, an array of floats, when particles.B_ext_particle_init_style is set to repeated_plasma_lens.

The applied field is uniform longitudinally (along \(z\)) with a hard edge, where residence corrections are used for more accurate field calculation. On the time step when a particle enters or leaves each lens, the field applied is scaled by the fraction of the time step spent within the lens. The fields are of the form

\[
E_x = \text{strength} \cdot x, \quad E_y = \text{strength} \cdot y, \quad \text{and} \quad E_z = 0,
\]

and

\[
B_x = 0, \quad B_y = -\text{strength} \cdot x, \quad \text{and} \quad B_z = 0.
\]

### 3.2.11 Collision initialization

WarpX provides a relativistic elastic Monte Carlo binary Coulomb collision model, following the algorithm given by Perez et al. (Phys. Plasmas 19, 083104, 2012). When the RZ mode is used, `warpx.n_rz_azimuthal_modes` must be set to 1 at the moment, since the current implementation of the collision module assumes axisymmetry. A non-relativistic Monte Carlo treatment for particles colliding with a neutral, uniform background gas is also available. The implementation follows the so-called null collision strategy discussed for example in Birdsall (IEEE Transactions on Plasma Science, vol. 19, no. 2, pp. 65-85, 1991).

- **collisions.collision_names** (*strings, separated by spaces*) The name of each collision type. This is then used in the rest of the input deck; in this documentation we use `<collision_name>` as a placeholder.

- **<collision_name>.type** (*string*) optional The type of collision. The types implemented are `pairwisecoulomb` for pairwise Coulomb collisions and `background_mcc` for collisions between particles and a neutral background. If not specified, it defaults to `pairwisecoulomb`.

- **<collision_name>.species** (*strings*) If using `pairwisecoulomb` type this should be the names of two species, between which the collision will be considered. The number of provided `<collision_name>.species` should match the number of collision names, i.e. `collisions.collision_names`. If using `background_mcc` type this should be the name of the species for which collisions will be included. Only one species name should be given.
• `<collision_name>`.CoulombLog (float) optional Only for pairwisecoulomb. A provided fixed Coulomb logarithm of the collision type `<collision_name>`. For example, a typical Coulomb logarithm has a form of \(\ln(\lambda_D/R)\), where \(\lambda_D\) is the Debye length, \(R \approx 1.4A^{1/3}\) is the effective Coulombic radius of the nucleus, \(A\) is the mass number. If this is not provided, or if a non-positive value is provided, a Coulomb logarithm will be computed automatically according to the algorithm in Perez et al. (Phys. Plasmas 19, 083104, 2012).

• `<collision_name>`.ndt (int) optional Execute collision every # time steps. The default value is 1.

• `<collision_name>`.background_density (float) Only for background_mcc. The density of the neutral background gas in \(m^{-3}\). Can also provide `<collision_name>`.background_density(x,y,z,t) using the parser initialization style for spatially and temporally varying density. If a function is used for the background density, the input parameter `<collision_name>`.max_background_density must also be provided to calculate the maximum collision probability.

• `<collision_name>`.background_temperature (float) Only for background_mcc. The temperature of the neutral background gas in Kelvin. Can also provide `<collision_name>`.background_temperature(x, y, z, t) using the parser initialization style for spatially and temporally varying temperature.

• `<collision_name>`.background_mass (float) optional Only for background_mcc. The mass of the background gas in kg. If not given the mass of the colliding species will be used unless ionization is included in which case the mass of the product species will be used.

• `<collision_name>`.scattering_processes (strings separated by spaces) Only for background_mcc. The MCC scattering processes that should be included. Available options are elastic, back & charge_exchange for ions and elastic, excitationX & ionization for electrons. The elastic option uses hard-sphere scattering, with a differential cross section that is independent of angle. With charge_exchange, the ion velocity is replaced with the neutral velocity, chosen from a Maxwellian based on the value of `<collision_name>`.background_temperature. Multiple excitation events can be included for electrons corresponding to excitation to different levels, the X above can be changed to a unique identifier for each excitation process. For each scattering process specified a path to a cross-section data file must also be given. We use <scattering_process> as a placeholder going forward.

• `<collision_name>`.<scattering_process>_cross_section (string) Only for background_mcc. Path to the file containing cross-section data for the given scattering processes. The cross-section file must have exactly 2 columns of data, the first containing equally spaced energies in eV and the second the corresponding cross-section in \(m^2\).

• `<collision_name>`.<scattering_process>_energy (float) Only for background_mcc. If the scattering process is either excitationX or ionization the energy cost of that process must be given in eV.

• `<collision_name>`.ionization_species (float) Only for background_mcc. If the scattering process is ionization the produced species must also be given. For example if argon properties is used for the background gas, a species of argon ions should be specified here.

### 3.2.12 Numerics and algorithms

• `warpx.cfl` (float) The ratio between the actual timestep that is used in the simulation and the Courant-Friedrichs-Lewy (CFL) limit. (e.g. for `warpx.cfl=1`, the timestep will be exactly equal to the CFL limit.)

• `warpx.use_filter` (0 or 1; default: 1, except for RZ FDTD) Whether to smooth the charge and currents on the mesh, after depositing them from the macro-particles. This uses a bilinear filter (see the filtering section). The default is 1 in all cases, except for simulations in RZ geometry using the FDTD solver. With the RZ PSATD solver, the filtering is done in \(k\)-space.
Warning: Known bug: filter currently not working with FDTD solver in RZ geometry (see https://github.com/ECP-WarpX/WarpX/issues/1943).

- `warpx.filter_npass_each_dir (3 int) optional (default 1 1 1)` Number of passes along each direction for the bilinear filter. In 2D simulations, only the first two values are read.

- `warpx.use_filter_compensation (0 or 1; default: 0)` Whether to add compensation when applying filtering. This is only supported with the RZ spectral solver.

- `algo.current_deposition (string, optional)` This parameter selects the algorithm for the deposition of the current density. Available options are: `direct`, `esirkepov`, and `vay`. The default choice is `esirkepov` for FDTD maxwell solvers and `direct` for standard or Galilean PSATD solver (that is, with `algo.maxwell_solver = psatd`).

  1. **direct**  
     The current density is deposited as described in the section *Current deposition*. This deposition scheme does not conserve charge.

  2. **esirkepov**  
     The current density is deposited as described in *(Esirkepov, CPC, 2001)*. This deposition scheme guarantees charge conservation for shape factors of arbitrary order.

  3. **vay**  
     The current density is deposited as described in *(Vay et al, 2013)* (see section *Current deposition* for more details). This option guarantees charge conservation only when used in combination with `psatd.periodic_single_box_fft=1`, that is, only for periodic single-box simulations with global FFTs without guard cells. The implementation for domain decomposition with local FFTs over guard cells is planned but not yet completed.

- `algo.charge_deposition (string, optional)` The algorithm for the charge density deposition. Available options are:

  - **standard**: standard charge deposition algorithm, described in the *particle-in-cell theory section*.

- `algo.field_gathering (string, optional)` The algorithm for field gathering. Available options are:

  - **energy-conserving**: gathers directly from the grid points (either staggered or nodal grid-points depending on `warpx.do_nodal`).

  - **momentum-conserving**: first average the fields from the grid points to the nodes, and then gather from the nodes.

  If `algo.field_gathering` is not specified, the default is `energy-conserving`. If `warpx.do_nodal` is true, then `energy-conserving` and `momentum-conserving` are equivalent.

- `algo.particle_pusher (string, optional)` The algorithm for the particle pusher. Available options are:

  - **boris**: Boris pusher.

  - **vay**: Vay pusher (see Vay, Phys. Plasmas (2008))

  - **higuera**: Higuera-Cary pusher (see Higuera and Cary, Phys. Plasmas (2017))

  If `algo.particle_pusher` is not specified, `boris` is the default.

- `algo.particle_shape (integer; 1, 2, or 3)` The order of the shape factors (splines) for the macro-particles along all spatial directions: 1 for linear, 2 for quadratic, 3 for cubic. Low-order shape factors result in faster simulations, but may lead to more noisy results. High-order shape factors are computationally more expensive, but may increase the overall accuracy of the results. For production runs it is generally safer to use high-order shape factors, such as cubic order.
Note that this input parameter is not optional and must always be set in all input files provided that there is at least one particle species (set in input as `particles.species_names`) or one laser species (set in input as `lasers.names`) in the simulation. No default value is provided automatically.

- **algo.maxwell_solver (string, optional)** The algorithm for the Maxwell field solver. Available options are:
  - `yee`: Yee FDTD solver.
  - `ckc`: (not available in RZ geometry) Cole-Karkkainen solver with Cowan coefficients (see Cowan, PRSTAB 16 (2013))
  - `psatd`: Pseudo-spectral solver (see theory)

  If `algo.maxwell_solver` is not specified, `yee` is the default.

- **algo.em_solver_medium (string, optional)** The medium for evaluating the Maxwell solver. Available options are:
  - `vacuum`: vacuum properties are used in the Maxwell solver.
  - `macroscopic`: macroscopic Maxwell equation is evaluated. If this option is selected, then the corresponding properties of the medium must be provided using `macroscopic.sigma`, `macroscopic.epsilon`, and `macroscopic.mu` for each case where the initialization style is constant. Otherwise if the initialization style uses the parser, `macroscopic.sigma_function(x,y,z)`, `macroscopic.epsilon_function(x,y,z)` and/or `macroscopic.mu_function(x,y,z)` must be provided using the parser initialization style for spatially varying macroscopic properties.

  If `algo.em_solver_medium` is not specified, `vacuum` is the default.

- **algo.macroscopic_sigma_method (string, optional)** The algorithm for updating electric field when `algo.em_solver_medium` is macroscopic. Available options are:
  - `backwardeuler`: is a fully-implicit, first-order in time scheme for E-update (default).
  - `laxwendroff`: is the semi-implicit, second order in time scheme for E-update.

  Comparing the two methods, Lax-Wendroff is more prone to developing oscillations and requires a smaller timestep for stability. On the other hand, Backward Euler is more robust but it is first-order accurate in time compared to the second-order Lax-Wendroff method.

- **macroscopic.sigma_function(x,y,z), macroscopic.epsilon_function(x,y,z), macroscopic.mu_function(x,y,z)** To initialize spatially varying conductivity, permittivity, and permeability, respectively, using a mathematical function in the input. Constants required in the mathematical expression can be set using `my_constants`. These parameters are parsed if `algo.em_solver_medium=macroscopic`.

- **macroscopic.sigma, macroscopic.epsilon, macroscopic.mu (double)** To initialize a constant conductivity, permittivity, and permeability of the computational medium, respectively. The default values are the corresponding values in vacuum.

- **interpolation.galerkin_scheme (0 or 1)** Whether to use a Galerkin scheme when gathering fields to particles. When set to 1, the interpolation orders used for field-gathering are reduced for certain field components along certain directions. For example, $E_z$ is gathered using `algo.particle_shape` along $(x,y)$ and `algo.particle_shape - 1` along $z$. See equations (21)-(23) of (Godfrey and Vay, 2013) and associated references for details. Defaults to 1 unless `warpx.do_nodal = 1` and/or `algo.field_gathering = momentum-conserving`.
**Warning:** The default behavior should not normally be changed. At present, this parameter is intended mainly for testing and development purposes.

- **interpolation.field_centering_nox, interpolation.field_centering_noy, interpolation.field_centering_noz**
  
  The order of interpolation used with staggered grids (warpx.do_nodal = 0) and momentum-conserving field gathering (algo.field_gathering = momentum-conserving) to interpolate the electric and magnetic fields from the cell centers to the cell nodes, before gathering the fields from the cell nodes to the particle positions. High-order interpolation (order 8 in each direction, at least) is necessary to ensure stability in typical LWFA boosted-frame simulations using the Galilean PSATD or comoving PSATD schemes. This finite-order interpolation is used only when the PSATD solver is used for Maxwell’s equations. With the FDTD solver, basic linear interpolation is used instead.

- **interpolation.current_centering_nox, interpolation.current_centering_noy, interpolation.current_centering_noz**
  
  The order of interpolation used to center the currents from nodal to staggered grids (if warpx.do_current_centering = 1), before pushing the Maxwell fields on staggered grids. This finite-order interpolation is used only when the PSATD solver is used for Maxwell’s equations. With the FDTD solver, basic linear interpolation is used instead.

- **warpx.do_current_centering (0 or 1; default: 0)**
  
  If true, the current is deposited on a nodal grid and then centered to a staggered grid (Yee grid), using finite-order interpolation. If warpx.do_nodal = 1, the Maxwell fields are pushed on a nodal grid, it is not necessary to center the currents to a staggered grid, and we set therefore warpx.do_current_centering = 0 automatically, overwriting the user-defined input.

- **warpx.do_dive_cleaning (0 or 1; default: 0)**
  
  Whether to use modified Maxwell equations that progressively eliminate the error in $\nabla (E) - \rho$. This can be useful when using a current deposition algorithm which is not strictly charge-conserving, or when using mesh refinement. These modified Maxwell equations will cause the error to propagate (at the speed of light) to the boundaries of the simulation domain, where it can be absorbed.

- **warpx.do_nodal (0 or 1; default: 0)**
  
  Whether to use a nodal grid (i.e. all fields are defined at the same points in space) or a staggered grid (i.e. Yee grid; different fields are defined at different points in space)

- **warpx.do_subcycling (0 or 1; default: 0)**
  
  Whether or not to use sub-cycling. Different refinement levels have a different cell size, which results in different Courant–Friedrichs–Lewy (CFL) limits for the time step. By default, when using mesh refinement, the same time step is used for all levels. This time step is taken as the CFL limit of the finest level. Hence, for coarser levels, the timestep is only a fraction of the CFL limit for this level, which may lead to numerical artifacts. With sub-cycling, each level evolves with its own time step, set to its own CFL limit. In practice, it means that when level 0 performs one iteration, level 1 performs two iterations. Currently, this option is only supported when amr.max_level = 1. More information can be found at https://ieeexplore.ieee.org/document/8659392.

- **warpx.do_multi_J (0 or 1; default: 0)**
  
  Whether to use the multi-J algorithm, where current deposition and field update are performed multiple times within each time step. The number of sub-steps is determined by the input parameter warpx.do_multi_J_n_depositions. Unlike sub-cycling, field gathering is performed only once per time step, as in regular PIC cycles. When warpx.do_multi_J = 1, we perform linear interpolation of two distinct currents deposited at the beginning and the end of the time step, instead of using one single current deposited at half time. For simulations with strong numerical Cherenkov instability (NCI), it is recommended to use the multi-J algorithm in combination with psatd.do_time_averaging = 1.

- **warpx.do_multi_J_n_depositions (integer)**
  
  Number of sub-steps to use with the multi-J algorithm, when warpx.do_multi_J = 1. Note that this input parameter is not optional and must always be set in all input files where warpx.do_multi_J = 1. No default value is provided automatically.

- **psatd.nox, psatd.noy, psatd.noz (integer) optional (default 16 for all)**
  
  The order of accuracy of the spatial derivatives, when using the code compiled with a PSATD solver. If psatd.periodic_single_box_fft is used, these can be set to inf for infinite-order PSATD.
• **psatd.nx_guard**, `psatd.ny_guard`, `psatd.nz_guard` (integer) optional The number of guard cells to use with PSATD solver. If not set by users, these values are calculated automatically and determined empirically and would be equal the order of the solver for nodal grid, and half the order of the solver for staggered.

• **psatd.periodic_single_box_fft (0 or 1; default: 0)** If true, this will not incorporate the guard cells into the box over which FFTs are performed. This is only valid when WarpX is run with periodic boundaries and a single box. In this case, using `psatd.periodic_single_box_fft` is equivalent to using a global FFT over the whole domain. Therefore, all the approximations that are usually made when using local FFTs with guard cells (for problems with multiple boxes) become exact in the case of the periodic, single-box FFT without guard cells.

• **psatd.current_correction (0 or 1; default: 0)** If true, a current correction scheme in Fourier space is applied in order to guarantee charge conservation. If `psatd.v_galilean` is zero, the spectral solver used is the standard PSATD scheme described in (Vay et al, JCP 243, 2013) and the current correction reads

\[
\hat{J}\frac{n+1}{2}\text{correct} = \hat{J}\frac{n+1}{2} - \left( k \cdot \hat{J}\frac{n+1}{2} - i \hat{\rho}\frac{n+1}{2} - \hat{\rho}\frac{n}{2} \right) \frac{k^2}{\Omega}
\]

If `psatd.v_galilean` is non-zero, the spectral solver used is the Galilean PSATD scheme described in (Lehe et al, PRE 94, 2016) and the current correction reads

\[
\hat{J}\frac{n+1}{2}\text{correct} = \hat{J}\frac{n+1}{2} - \left( k \cdot \hat{J}\frac{n+1}{2} - (k \cdot v_G) \hat{\rho}\frac{n+1}{2} - \hat{\rho}\frac{n}{2} \right) \frac{k^2}{\Omega}
\]

where \( \theta = \exp(i k \cdot v_G \Delta t/2) \).

This option is currently implemented only for the standard PSATD and Galilean PSATD schemes, while it is not yet available for the averaged Galilean PSATD scheme (activated by the input parameter `psatd.do_time_averaging`).

This option guarantees charge conservation only when used in combination with `psatd.periodic_single_box_fft=1`, namely for periodic single-box simulations with global FFTs without guard cells. The implementation for domain decomposition with local FFTs over guard cells is planned but not yet completed.

• **psatd.update_with_rho (0 or 1)** If true, the update equation for the electric field is expressed in terms of both the current density and the charge density, namely \( \hat{J}\frac{n+1}{2}, \hat{\rho}\frac{n}{2}, \hat{\rho}\frac{n+1}{2} \). If false, instead, the update equation for the electric field is expressed in terms of the current density \( \hat{J}\frac{n+1}{2} \) only. If charge is expected to be conserved (by setting, for example, `psatd.current_correction=1`), then the two formulations are expected to be equivalent.

This option is currently implemented only for the standard PSATD and Galilean PSATD schemes, while it is not yet available for the averaged Galilean PSATD scheme (activated by the input parameter `psatd.do_time_averaging`).

If `psatd.v_galilean` is zero, the spectral solver used is the standard PSATD scheme described in (Vay et al, JCP 243, 2013):

1. if `psatd.update_with_rho=0`, the update equation for the electric field reads

\[
\hat{E}\frac{n+1}{2} = C \hat{E}\frac{n}{2} + \frac{S}{k} \times \hat{B}\frac{n}{2} - \frac{S}{\epsilon_0 k} \hat{J}\frac{n+1}{2}
\]

\[
+ \frac{1-C}{\Omega k^2} (k \cdot \hat{E}\frac{n}{2} k) + \frac{1}{\epsilon_0 k^2} \left( \frac{S}{\epsilon k} - \Delta t \right) (k \cdot \hat{J}\frac{n+1}{2}) k
\]

2. if `psatd.update_with_rho=1`, the update equation for the electric field reads

### 3.2. Input Parameters
\[
E^{n+1} = C E^n + i \frac{\varepsilon_0 c}{k} k \times \dot{B}^n - \frac{S}{\varepsilon_0 c k} \dot{\boldsymbol{J}}^{n+1/2} \\
+ i \frac{1}{\varepsilon_0 k^2} \left( C - \frac{S}{c k} \frac{1}{\Delta t} \right) \dot{\rho}^n k - i \frac{1}{\varepsilon_0 k^2} \left( 1 - \frac{S}{c k} \frac{1}{\Delta t} \right) \dot{\rho}^{n+1} k
\]

The coefficients \( C \) and \( S \) are defined in (Vay et al, JCP 243, 2013).

If \texttt{psatd.v\_galilean} is non-zero, the spectral solver used is the Galilean PSATD scheme described in (Lehe et al, PRE 94, 2016):

1. if \texttt{psatd.update\_with\_rho=0}, the update equation for the electric field reads

\[
E^{n+1} = \theta^2 C \hat{E}^n + i \frac{\theta^2 S \varepsilon_0 c}{k} k \times \hat{B}^n + \frac{i \nu \theta \chi_1 - \theta^2 S}{\varepsilon_0 c k} \dot{\boldsymbol{J}}^{n+1/2} \\
+ \theta^2 \frac{\chi_2 - \chi_3}{k^2} (\mathbf{k} \cdot \hat{E}^n) k + i \frac{\chi_2}{\varepsilon_0 c k^3 \nu} \frac{1}{\Delta t} \frac{k \cdot \dot{J}^{n+1/2}}{k} k
\]

2. if \texttt{psatd.update\_with\_rho=1}, the update equation for the electric field reads

\[
E^{n+1} = \theta^2 C \hat{E}^n + i \frac{\theta^2 S \varepsilon_0 c}{k} k \times \hat{B}^n + \frac{i \nu \theta \chi_1 - \theta^2 S}{\varepsilon_0 c k} \dot{\boldsymbol{J}}^{n+1/2} \\
+ i \frac{\theta^2 \chi_3}{\varepsilon_0 k^2} \dot{\rho}^n k - i \frac{\chi_2}{\varepsilon_0 k^2} \dot{\rho}^{n+1} k
\]

The coefficients \( C, \theta, \nu, \chi_1, \chi_2, \) and \( \chi_3 \) are defined in (Lehe et al, PRE 94, 2016).

The default value for \texttt{psatd.update\_with\_rho} is 1 if \texttt{psatd.v\_galilean} is non-zero and 0 otherwise.

Note that the update with and without rho is also supported in RZ geometry.

- \texttt{psatd.v\_galilean (3 floats, in units of the speed of light; default 0.0.0.0.)} Defines the galilean velocity. Non-zero \texttt{v\_galilean} activates Galilean algorithm, which suppresses the Numerical Cherenkov instability in boosted-frame simulation. This requires the code to be compiled with \texttt{USE_PSATD = TRUE}. (see the sub-section Numerical Stability and alternate formulation in a Galilean frame in the \texttt{theory section}). It also requires the use of the \texttt{direct} current deposition option \texttt{algo.current\_deposition = direct} (does not work with Esirkepov algorithm).

- \texttt{psatd.v\_comoving (3 floating-point values, in units of the speed of light; default 0.0.0.)} Defines the comoving velocity in the comoving PSATD scheme. A non-zero comoving velocity selects the comoving PSATD algorithm, which suppresses the numerical Cherenkov instability (NCI) in boosted-frame simulations, under certain assumptions. This option requires that WarpX is compiled with \texttt{USE_PSATD = TRUE}. It also requires the use of direct current deposition (\texttt{algo.current\_deposition = direct}) and has not been neither implemented nor tested with other current deposition schemes.

- \texttt{psatd.do\_time\_averaging (0 or 1; default: 0)} Whether to use an averaged Galilean PSATD algorithm or standard Galilean PSATD.

- \texttt{warpx.override\_sync\_intervals (string) optional (default I)} Using the \texttt{Intervals parser} syntax, this string defines the timesteps at which synchronization of sources (\texttt{rho} and \texttt{J}) and fields (\texttt{E} and \texttt{B}) on grid nodes at box boundaries is performed. Since the grid nodes at the interface between two neighbor boxes are duplicated in both boxes, an instability can occur if they have too different values. This option makes sure that they are synchronized periodically. Note that if Perfectly Matched Layers (PML) are used, synchronization of the \texttt{E} and \texttt{B} fields is performed at every timestep regardless of this parameter.

- \texttt{warpx.use\_hybrid\_QED (bool; default: 0)} Will use the Hybrid QED Maxwell solver when pushing fields: a QED correction is added to the field solver to solve non-linear Maxwell’s equations, according to [Quantum Electrodynamics vacuum polarization solver, P. Carneiro et al., ArXiv 2016]. Note that this option can only be used with the PSATD build. Furthermore, \texttt{warpx.do\_nodal} must be set to \texttt{I} which is not its default value.
• **warpx.quantum_xi** (*float; default: 1.3050122.e-52*) Overwrites the actual quantum parameter used in Maxwell’s QED equations. Assigning a value here will make the simulation unphysical, but will allow QED effects to become more apparent. Note that this option will only have an effect if the `warpx.use_Hybrid_QED` flag is also triggered.

• **warpx.do_device_synchronize** (*bool*) optional (default 1) When running in an accelerated platform, whether to call a `amrex::Gpu::synchronize()` around profiling regions. This allows the profiler to give meaningful timers, but (hardly) slows down the simulation.

• **warpx.sort_intervals** (*string*) optional (defaults: -1 on CPU; 4 on GPU) Using the *Intervals parser* syntax, this string defines the timesteps at which particles are sorted by bin. If <=0, do not sort particles. It is turned on on GPUs for performance reasons (to improve memory locality).

• **warpx.sort_bin_size** (list of *int*) optional (default 1 1 1) If `sort_intervals` is activated particles are sorted in bins of `sort_bin_size` cells. In 2D, only the first two elements are read.

### 3.2.13 Diagnostics and output

#### In-situ visualization

WarpX has three types of diagnostics: FullDiagnostics consist in dumps of fields and particles at given iterations, BackTransformedDiagnostics are used when running a simulation in a boosted frame, to reconstruct output data to the lab frame, and ReducedDiags allow the user to compute some reduced quantity (particle temperature, max of a field) and write a small amount of data to text files. Similar to what is done for physical species, WarpX has a class Diagnostics that allows users to initialize different diagnostics, each of them with different fields, resolution and period. This currently applies to standard diagnostics, but should be extended to back-transformed diagnostics and reduced diagnostics (and others) in a near future.

#### Full Diagnostics

FullDiagnostics consist in dumps of fields and particles at given iterations. Similar to what is done for physical species, WarpX has a class Diagnostics that allows users to initialize different diagnostics, each of them with different fields, resolution and period. The user specifies the number of diagnostics and the name of each of them, and then specifies options for each of them separately. Note that some parameter (those that do not start with a `<diag_name>` prefix) apply to all diagnostics. This should be changed in the future. In-situ capabilities can be used by turning on Sensei or Ascent (provided they are installed) through the output format, see below.

• **diagnostics.enable** (*0 or 1*, optional, default 1) Whether to enable or disable diagnostics. This flag overwrites all other diagnostics input parameters.

• **diagnostics.diags_names** (list of *string* optional, default empty) Name of each diagnostics. example: `diagnostics.diags_names = diag1 my_second_diag`.

• `<diag_name>.intervals` (*string*) Using the *Intervals parser* syntax, this string defines the timesteps at which data is dumped. Use a negative number or 0 to disable data dumping. example: `diag1.intervals = 10, 20:25:1`. Note that by default the last timestep is dumped regardless of this parameter. This can be changed using the parameter `<diag_name>.dump_last_timestep` described below.

• `<diag_name>.dump_last_timestep` (*bool* optional, default 1) If this is 1, the last timestep is dumped regardless of `<diag_name>.period`.

• `<diag_name>.diag_type` (*string*) Type of diagnostics. Full and BackTransformed example: `diag1.diag_type = Full` or `diag1.diag_type = BackTransformed`.

• `<diag_name>.format` (*string* optional, default `plotfile`) Flush format. Possible values are:
  - `plotfile` for native AMReX format.
- checkpoint for a checkpoint file, only works with <diag_name>.diag_type = Full.
- openpmd for OpenPMD format openPMD. Requires to build WarpX with USE_OPENPMD=TRUE (see instructions).
- ascent for in-situ visualization using Ascent.
- sensei for in-situ visualization using Sensei.

example: diag1.format = openpmd.

- <diag_name>.sensei_config (string) Only read if <diag_name>.format = sensei. Points to the SEN-SEI XML file which selects and configures the desired back end.

- <diag_name>.sensei_pin_mesh (integer; 0 by default) Only read if <diag_name>.format = sensei. When 1 lower left corner of the mesh is pinned to 0.,0.,0.

- <diag_name>.openpmd_backend (bp, h5 or json) optional, only used if <diag_name>.format = openpmd I/O backend for openPMD data dumps. bp is the ADIOS I/O library, h5 is the HDF5 format, and json is a simple text format. json only works with serial/single-rank jobs. When WarpX is compiled with openPMD support, the first available backend in the order given above is taken. Note that when using BackTransformed diagnostic type, the openpmd format supports only h5 backend for both species and fields, while bp backend can be used for visualizing fields, but not particles. The code will abort if bp is selected for particle output.

- <diag_name>.openpmd_encoding (optional, v (variable based), f (file based) or g (group based) ) only read if <diag_name>.format = openpmd openPMD file output encoding. File based: one file per timestep (slower), group/variable based: one file for all steps (faster). variable based is an experimental feature with ADIOS2 and not supported for back-transformed diagnostics. Default: f (full diagnostics)

- <diag_name>.adios2_operator.type (zfp, blosc) optional, ADIOS2 I/O operator type for openPMD data dumps.

- <diag_name>.adios2_operator.parameters.* optional, ADIOS2 I/O operator parameters for openPMD data dumps.

A typical example for ADIOS2 output using lossless compression with blosc using the zstd compressor and 6 CPU treads per MPI Rank (e.g. for a GPU run with spare CPU resources):

```plaintext
<diag_name>.adios2_operator.type = blosc
<diag_name>.adios2_operator.parameters.compressor = zstd
<diag_name>.adios2_operator.parameters.clevel = 1
<diag_name>.adios2_operator.parameters.doshuffle = BLOSC_BITSHUFFLE
<diag_name>.adios2_operator.parameters.threshold = 2048
<diag_name>.adios2_operator.parameters.nthreads = 6 # per MPI rank (and thus per GPU)
```

or for the lossy ZFP compressor using very strong compression per scalar:

```plaintext
<diag_name>.adios2_operator.type = zfp
<diag_name>.adios2_operator.parameters.precision = 3
```

- <diag_name>.adios2_engine.type (bp4, sst, ssc, dataman) optional, ADIOS2 Engine type for openPMD data dumps. See full list of engines at ADIOS2 readthedocs

- <diag_name>.adios2_engine.parameters.* optional, ADIOS2 Engine parameters for openPMD data dumps.

An example for parameters for the BP engine are setting the number of writers (NumAggregators), transparently redirecting data to burst buffers etc. A detailed list of engine-specific parameters are available at the official ADIOS2 documentation
• `<diag_name>.fields_to_plot` (list of strings, optional) Fields written to output. Possible values: Ex Ey Ez Bx By Bz jx jy jz part_per_cell rho phi F part_per_grid divE divB and rho_<species_name>, where `<species_name>` must match the name of one of the available particle species. Note that phi will only be written out when do_electrostatic=labframe. Default is `<diag_name>.fields_to_plot = Ex Ey Ez Bx By Bz jx jy jz`. Note that the fields are averaged on the cell centers before they are written to file.

• `<diag_name>.plot_raw_fields` (0 or 1) optional (default 0) By default, the fields written in the plot files are averaged on the cell centers. When `<diag_name>.plot_raw_fields = 1`, then the raw (i.e. non-averaged) fields are also saved in the output files. Only works with `<diag_name>.format = plotfile`. See this section in the yt documentation for more details on how to view raw fields.

• `<diag_name>.plot_raw_fields_guards` (0 or 1) optional (default 0) Only used when `<diag_name>.plot_raw_fields = 1`. Whether to include the guard cells in the output of the raw fields. Only works with `<diag_name>.format = plotfile`.

• `<diag_name>.coarsening_ratio` (list of int) optional (default 1 1 1) Reduce size of the field output by this ratio in each dimension. (This is done by averaging the field over 1 or 2 points along each direction, depending on the staggering). If `blocking_factor` and `max_grid_size` are used for the domain decomposition, as detailed in the parallelization section, coarsening_ratio should be an integer divisor of `blocking_factor`. If `warpx.numprocs` is used instead, the total number of cells in a given dimension must be a multiple of the coarsening_ratio multiplied by `numprocs` in that dimension.

• `<diag_name>.file_prefix` (string) optional (default diags/<diag_name>) Root for output file names. Supports sub-directories.

• `<diag_name>.file_min_digits` (int) optional (default 6) The minimum number of digits used for the iteration number appended to the diagnostic file names.

• `<diag_name>.diag_lo` (list float, 1 per dimension) optional (default -infinity -infinity -infinity) Lower corner of the output fields (if smaller than `warpx.dom_lo`, then set to `warpx.dom_lo`). Currently, when the diag_lo is different from `warpx.dom_lo`, particle output is disabled.

• `<diag_name>.diag_hi` (list float, 1 per dimension) optional (default +infinity +infinity +infinity) Higher corner of the output fields (if larger than `warpx.dom_hi`, then set to `warpx.dom_hi`). Currently, when the diag_hi is different from `warpx.dom_hi`, particle output is disabled.

• `<diag_name>.write_species` (0 or 1) optional (default 1) Whether to write species output or not. For checkpoint format, always set this parameter to 1.

• `<diag_name>.<species_name>.variables` (list of strings separated by spaces, optional) List of particle quantities to write to output. Choices are w for the particle weight and ux uy uz for the particle momenta. By default, all particle quantities are written. If `<diag_name>.<species_name>.variables = none`, no particle data are written, except for particle positions, which are always included.

• `<diag_name>.<species_name>.random_fraction` (float) optional If provided `<diag_name>.<species_name>.random_fraction = a`, only a fraction of the particle data of this species will be dumped randomly in diag `<diag_name>`, i.e. if rand() < a, this particle will be dumped, where rand() denotes a random number generator. The value a provided should be between 0 and 1.

• `<diag_name>.<species_name>.uniform_stride` (int) optional If provided `<diag_name>.<species_name>.uniform_stride = n`, every n particle of this species will be dumped, selected uniformly. The value provided should be an integer greater than or equal to 0.
• `<diag_name>.<species_name>.plot_filter_function(t,x,y,z,ux,uy,uz) (string)` optional Users can provide an expression returning a boolean for whether a particle is dumped (the exact test is whether the return value is > 0.5). `t` represents the physical time in seconds during the simulation. `x, y, z` represent particle positions in the unit of meter. `ux, uy, uz` represent particle velocities in the unit of $\gamma v/c$, where $\gamma$ is the Lorentz factor, $v/c$ is the particle velocity normalized by the speed of light. E.g. If provided `(x>0.0)*(uz<10.0)` only those particles located at positions $x$ greater than 0, and those having velocity $uz$ less than 10, will be dumped.

• `amrex.async_out (0 or 1)` optional (default 0) Whether to use asynchronous IO when writing plotfiles. This only has an effect when using the AMReX plotfile format. Please see the data analysis section for more information.

• `amrex.async_out_nfiles (int)` optional (default 64) The maximum number of files to write to when using asynchronous IO. To use asynchronous IO with more than `amrex.async_out_nfiles` MPI ranks, WarpX must be configured with `-DWarpX_MPI_THREAD_MULTIPLE=ON`. Please see the data analysis section for more information.

**BackTransformed Diagnostics (with support for Plotfile/openPMD output)**

BackTransformed diag type are used when running a simulation in a boosted frame, to reconstruct output data to the lab frame. This option can be set using `<diag_name>.diag_type = BackTransformed`. Additional options for this diagnostic include:

• `<diag_name>.num_snapshots_lab (integer)` Only used when `<diag_name>.diag_type` is BackTransformed. The number of lab-frame snapshots that will be written.

• `<diag_name>.dt_snapshots_lab (float, in seconds)` Only used when `<diag_name>.diag_type` is BackTransformed. The time interval inbetween the lab-frame snapshots (where this time interval is expressed in the laboratory frame).

• `<diag_name>.dz_snapshots_lab (float, in meters)` Only used when `<diag_name>.diag_type` is BackTransformed. Distance between the lab-frame snapshots (expressed in the laboratory frame). `dt_snapshots_lab` is then computed by `dt_snapshots_lab = dz_snapshots_lab/c`. Either `dt_snapshots_lab` or `dz_snapshots_lab` is required.

• `<diag_name>.buffer_size (integer)` Only used when `<diag_name>.diag_type` is BackTransformed. The default size of the back transformed diagnostic buffers used to generate lab-frame data is 256. That is, when the multifab with lab-frame data has 256 z-slices, the data will be flushed out. However, if many lab-frame snapshots are required for diagnostics and visualization, the GPU may run out of memory with many large boxes with a size of 256 in the z-direction. This input parameter can then be used to set a smaller buffer-size, preferably multiples of 8, such that, a large number of lab-frame snapshot data can be generated without running out of gpu memory. The downside to using a small buffer size, is that the I/O time may increase due to frequent flushes of the lab-frame data. The other option is to keep the default value for buffer size and use slices to reduce the memory footprint and maintain optimum I/O performance.

**Back-Transformed Diagnostics (legacy output)**

BackTransformedDiagnostics are used when running a simulation in a boosted frame, to reconstruct output data to the lab frame, and

• `warpx.do_back_transformed_diagnostics (0 or 1)` Whether to use the back-transformed diagnostics (i.e. diagnostics that perform on-the-fly conversion to the laboratory frame, when running boosted-frame simulations)

• `warpx.lab_data_directory (string)` The directory in which to save the lab frame data when using the back-transformed diagnostics. If not specified, the default is `is lab_frame_data`.  

94 Chapter 3. Usage
• **warpx.num_snapshots_lab** *(integer)* Only used when `warpx.do_back_transformed_diagnostics` is 1. The number of lab-frame snapshots that will be written.

• **warpx.dt_snapshots_lab** *(float, in seconds)* Only used when `warpx.do_back_transformed_diagnostics` is 1. The time interval inbetween the lab-frame snapshots (where this time interval is expressed in the laboratory frame).

• **warpx.dz_snapshots_lab** *(float, in meters)* Only used when `warpx.do_back_transformed_diagnostics` is 1. Distance between the lab-frame snapshots (expressed in the laboratory frame). `dt_snapshots_lab` is then computed by `dt_snapshots_lab = dz_snapshots_lab/c`. Either `dt_snapshots_lab` or `dz_snapshots_lab` is required.

• **warpx.do_back_transformed_fields** *(0 or 1)* Whether to use the back-transformed diagnostics for the fields.

• **warpx.back_transformed_diag_fields** *(space-separated list of string)* Which fields to dumped in back-transformed diagnostics. Choices are `'Ex'`, `'Ey'`, `'Ez'`, `'Bx'`, `'By'`, `'Bz'`, `'jx'`, `'jy'`, `'jz'` and `'rho'`. Example: `warpx.back_transformed_diag_fields = Ex Ez By`. By default, all fields are dumped.

• **warpx.buffer_size** *(integer)* The default size of the back-transformed diagnostic buffers used to generate lab-frame data is 256. That is, when the multifab with lab-frame data has 256 z-slices, the data will be flushed out. However, if many lab-frame snapshots are required for diagnostics and visualization, the GPU may run out of memory with many large boxes with a size of 256 in the z-direction. This input parameter can then be used to set a smaller buffer-size, preferably multiples of 8, such that, a large number of lab-frame snapshot data can be generated without running out of gpu memory. The downside to using a small buffer size, is that the I/O time may increase due to frequent flushes of the lab-frame data. The other option is to keep the default value for buffer size and use slices to reduce the memory footprint and maintain optimum I/O performance.

• **slice.num_slice_snapshots_lab** *(integer)* Only used when `warpx.do_back_transformed_diagnostics` is 1. The number of back-transformed field and particle data that will be written for the reduced domain defined by `slice.dom_lo` and `slice.dom_hi`. Note that the ‘slice’ is a reduced diagnostic which could be 1D, 2D, or 3D, aligned with the co-ordinate axes. These slices can be visualized using `read_raw_data.py` and the HDF5 format can be visualized using the h5py library. Please see the documentation on visualization for further details.

• **slice.dt_slice_snapshots_lab** *(float, in seconds)* Only used when `warpx.do_back_transformed_diagnostics` is 1. The time interval between the back-transformed reduced diagnostics (where this time interval is expressed in the laboratory frame).

• **slice.particle_slice_width_lab** *(float, in meters)* Only used when `warpx.do_back_transformed_diagnostics` is 1 and `slice.num_slice_snapshots_lab` is non-zero. Particles are copied from the full back-transformed diagnostic to the reduced slice diagnostic if there are within the user-defined width from the slice region defined by `slice.dom_lo` and `slice.dom_hi`.

### Reduced Diagnostics

ReducedDiags allow the user to compute some reduced quantity (particle temperature, max of a field) and write a small amount of data to text files.

• **warpx.reduced_diags_names** *(strings, separated by spaces)* The names given by the user of simple reduced diagnostics. Also the names of the output `.txt` files. This reduced diagnostics aims to produce simple outputs of the time history of some physical quantities. If `warpx.reduced_diags_names` is not provided in the input file, no reduced diagnostics will be done. This is then used in the rest of the input deck; in this documentation we use `<reduced_diags_name>` as a placeholder.

• **<reduced_diags_name>.type** *(string)* The type of reduced diagnostics associated with this `<reduced_diags_name>`. For example, ParticleEnergy, FieldEnergy, etc. All available types...
are described below in detail. For all reduced diagnostics, the first and the second columns in the output file are the time step and the corresponding physical time in seconds, respectively.

- **ParticleEnergy** This type computes the total and mean relativistic particle kinetic energy among all species:

\[
E_p = \sum_{i=1}^{N} w_i \left( \sqrt{|p_i|^2 c^2 + m_0^2 c^4} - m_0 c^2 \right)
\]

where \( p_i \) is the relativistic momentum of the \( i \)-th particle, \( c \) is the speed of light, \( m_0 \) is the rest mass, \( N \) is the number of particles, and \( w_i \) is the weight of the \( i \)-th particle.

The output columns are the total energy of all species, the total energy per species, the total mean energy \( E_p / \sum_i w_i \) of all species, and the total mean energy per species.

- **ParticleMomentum** This type computes the total and mean relativistic particle momentum among all species:

\[
P_p = \sum_{i=1}^{N} w_i p_i
\]

where \( p_i \) is the relativistic momentum of the \( i \)-th particle, \( N \) is the number of particles, and \( w_i \) is the weight of the \( i \)-th particle.

The output columns are the components of the total momentum of all species, the total momentum per species, the total mean momentum \( P_p / \sum_i w_i \) of all species, and the total mean momentum per species.

- **FieldEnergy** This type computes the electromagnetic field energy

\[
E_f = \frac{1}{2} \sum_{\text{cells}} \left( \varepsilon_0 |E|^2 + \frac{|B|^2}{\mu_0} \right) \Delta V
\]

where \( E \) is the electric field, \( B \) is the magnetic field, \( \varepsilon_0 \) is the vacuum permittivity, \( \mu_0 \) is the vacuum permeability, \( \Delta V \) is the cell volume (or cell area in 2D), and the sum is over all cells.

The output columns are the total field energy \( E_f \), the \( E \) field energy, and the \( B \) field energy, at each mesh refinement level.

- **FieldMomentum** This type computes the electromagnetic field momentum

\[
P_f = \varepsilon_0 \sum_{\text{cells}} (E \times B) \Delta V
\]

where \( E \) is the electric field, \( B \) is the magnetic field, \( \varepsilon_0 \) is the vacuum permittivity, \( \Delta V \) is the cell volume (or cell area in 2D), and the sum is over all cells.

The output columns are the components of the total field momentum \( P_f \) at each mesh refinement level.

Note that the fields are not averaged on the cell centers before their energy is computed.

- **FieldMaximum** This type computes the maximum value of each component of the electric and magnetic fields and of the norm of the electric and magnetic field vectors. Measuring maximum fields in a plasma might be very noisy in PIC, use this instead for analysis of scenarios such as an electromagnetic wave propagating in vacuum.

The output columns are the maximum value of the \( E_x \) field, the maximum value of the \( E_y \) field, the maximum value of the \( E_z \) field, the maximum value of the norm \( |E| \) of the electric field, the maximum value of the \( B_x \) field, the maximum value of the \( B_y \) field, the maximum value of the \( B_z \) field.
$B_z$ field and the maximum value of the norm $|B|$ of the magnetic field, at mesh refinement levels from 0 to $n$.

Note that the fields are averaged on the cell centers before their maximum values are computed.

- **FieldProbe** This type computes the value of each component of the electric and magnetic fields and of the Poynting vector (a measure of electromagnetic flux) at points in the domain.

  Multiple geometries for point probes can be specified via `<reduced_diags_name>.
  probe_geometry = ...`:

  * Point (default): a single point
  * Line: a line of points with equal spacing
  * Plane: a plane of points with equal spacing

  **Point**: The point where the fields are measured is specified through the input parameters `<reduced_diags_name>.
  x_probe`, `<reduced_diags_name>.
  y_probe` and `<reduced_diags_name>.
  z_probe`.

  **Line**: probe a 1 dimensional line of points to create a line detector. Initial input parameters `x_probe`, `y_probe`, and `z_probe` designate one end of the line detector, while the far end is specified via `<reduced_diags_name>.
  x1_probe`, `<reduced_diags_name>.
  y1_probe`, `<reduced_diags_name>.
  z1_probe`. Additionally, `<reduced_diags_name>.resolution` must be defined to give the number of detector points along the line (equally spaced) to probe.

  **Plane**: probe a 2 dimensional plane of points to create a square plane detector. Initial input parameters `x_probe`, `y Probe`, and `z_probe` designate the center of the detector. The detector plane is normal to a vector specified by `<reduced_diags_name>.
  target_normal_x`, `<reduced_diags_name>.
  target_normal_y`, and `<reduced_diags_name>.
  target_normal_z`. Note that it is not necessary to specify the target_normal vector in a 2D simulation (the only supported normal is in y). The top of the plane is perpendicular to an “up” vector denoted by `<reduced_diags_name>.
  target_up_x`, `<reduced_diags_name>.
  target_up_y`, and `<reduced_diags_name>.
  target_up_z`. The detector has a square radius to be determined by `<reduced_diags_name>.detector_radius`. Similarly to the line detector, the plane detector requires a resolution `<reduced_diags_name>.
  resolution`, which denotes the number of detector particles along each side of the square detector.

  The output columns are the value of the $E_x$ field, the value of the $E_y$ field, the value of the $E_z$ field, the value of the $B_x$ field, the value of the $B_y$ field, the value of the $B_z$ field and the value of the Poynting Vector $|S|$ of the electromagnetic fields, at mesh refinement levels from 0 to $n$, at point $(x, y, z)$.

  Note: the norms are always interpolated to the measurement point before they are written to file. The electromagnetic field components are interpolated to the measurement point by default, but can be saved as non-averaged by setting `<reduced_diags_name>.
  raw_fields = true`, in which case the raw fields for the cell containing the measurement point are saved. The interpolation order can be set by specifying `<reduced_diags_name>.
  interp_order`, otherwise it is set to 1. Integrated electric and magnetic field components can instead be obtained by specifying `<reduced_diags_name>.
  integrate == true`.

  **Warning**: The FieldProbe reduced diagnostic does not yet add a Lorentz back transformation for boosted frame simulations. Thus, it records field data in the boosted frame, not (yet) in the lab frame.

- **RhoMaximum** This type computes the maximum and minimum values of the total charge density as well as the maximum absolute value of the charge density of each charged species. Please be aware
that measuring maximum charge densities might be very noisy in PIC simulations.

The output columns are the maximum value of the \( \rho \) field, the minimum value of the \( \rho \) field, the maximum value of the absolute \( |\rho| \) field of each charged species.

Note that the charge densities are averaged on the cell centers before their maximum values are computed.

- **FieldReduction** This type computes an arbitrary reduction of the positions and the electromagnetic fields.

  * `<reduced_diags_name>.reduced_function(x,y,z,Ex,Ey,Ez,Bx,By,Bz)` *(string)* An analytic function to be reduced must be provided, using the math parser.

  * `<reduced_diags_name>.reduction_type` *(string)* The type of reduction to be performed. It must be either Maximum, Minimum or Integral. Integral computes the spatial integral of the function defined in the parser by summing its value on all grid points and multiplying the result by the volume of a cell. Please be also aware that measuring maximum quantities might be very noisy in PIC simulations.

  The only output column is the reduced value.

  Note that the fields are averaged on the cell centers before the reduction is performed.

- **ParticleNumber** This type computes the total number of macroparticles and of physical particles (i.e. the sum of their weights) in the whole simulation domain (for each species and summed over all species). It can be useful in particular for simulations with creation (ionization, QED processes) or removal (resampling) of particles.

  The output columns are total number of macroparticles summed over all species, total number of macroparticles of each species, sum of the particles’ weight summed over all species, sum of the particles’ weight of each species.

- **BeamRelevant** This type computes properties of a particle beam relevant for particle accelerators, like position, momentum, emittance, etc.

  `<reduced_diags_name>.species` must be provided, such that the diagnostics are done for this (beam-like) species only.

  The output columns (for 3D-XYZ) are the following, where the average is done over the whole species (typical usage: the particle beam is in a separate species):

  [1], [2], [3]: The mean values of beam positions (m) \( \langle x \rangle, \langle y \rangle, \langle z \rangle \).

  [4], [5], [6]: The mean values of beam relativistic momenta (kg m/s) \( \langle p_x \rangle, \langle p_y \rangle, \langle p_z \rangle \).

  [7]: The mean Lorentz factor \( \langle \gamma \rangle \).

  [8], [9], [10]: The RMS values of beam positions (m) \( \delta_x = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}, \delta_y = \sqrt{\langle (y - \langle y \rangle)^2 \rangle}, \delta_z = \sqrt{\langle (z - \langle z \rangle)^2 \rangle} \).

  [11], [12], [13]: The RMS values of beam relativistic momenta (kg m/s) \( \delta_{px} = \sqrt{\langle (p_x - \langle p_x \rangle)^2 \rangle}, \delta_{py} = \sqrt{\langle (p_y - \langle p_y \rangle)^2 \rangle}, \delta_{pz} = \sqrt{\langle (p_z - \langle p_z \rangle)^2 \rangle} \).

  [14]: The RMS value of the Lorentz factor \( \sqrt{\langle (\gamma - \langle \gamma \rangle)^2 \rangle} \).

  [15], [16], [17]: beam projected transverse RMS normalized emittance (m) \( \epsilon_x = \frac{1}{mc} \sqrt{\delta_x^2 \delta_{px}^2 - \left\langle (x - \langle x \rangle)(p_x - \langle p_x \rangle) \right\rangle^2}, \epsilon_y = \frac{1}{mc} \sqrt{\delta_y^2 \delta_{py}^2 - \left\langle (y - \langle y \rangle)(p_y - \langle p_y \rangle) \right\rangle^2}, \epsilon_z = \frac{1}{mc} \sqrt{\delta_z^2 \delta_{pz}^2 - \left\langle (z - \langle z \rangle)(p_z - \langle p_z \rangle) \right\rangle^2} \).

  [18]: The charge of the beam (C).
For 2D-XZ, \( \langle y \rangle, \delta_y, \) and \( \epsilon_y \) will not be outputed.

- **LoadBalanceCosts** This type computes the cost, used in load balancing, for each box on the domain. The cost \( c \) is computed as

\[
c = n_{\text{particle}} \cdot w_{\text{particle}} + n_{\text{cell}} \cdot w_{\text{cell}},
\]

where \( n_{\text{particle}} \) is the number of particles on the box, \( w_{\text{particle}} \) is the particle cost weight factor (controlled by `algo.costs_heuristic_particles_wt`), \( n_{\text{cell}} \) is the number of cells on the box, and \( w_{\text{cell}} \) is the cell cost weight factor (controlled by `algo.costs_heuristic_cells_wt`).

- **LoadBalanceEfficiency** This type computes the load balance efficiency, given the present costs and distribution mapping. Load balance efficiency is computed as the mean cost over all ranks, divided by the maximum cost over all ranks. Until costs are recorded, load balance efficiency is output as -1; at earliest, the load balance efficiency can be output starting at step 2, since costs are not recorded until step 1.

- **ParticleHistogram** This type computes a user defined particle histogram.

- `<reduced_diags_name>.species (string)` A species name must be provided, such that the diagnostics are done for this species.

- `<reduced_diags_name>.histogram_function(t,x,y,z,ux,uy,uz) (string)` A histogram function must be provided. \( t \) represents the physical time in seconds during the simulation. \( x, y, z \) represent particle positions in the unit of meter. \( ux, uy, uz \) represent the particle velocities in the unit of \( \gamma v/c \), where \( \gamma \) is the Lorentz factor, \( v/c \) is the particle velocity normalized by the speed of light. E.g. \( x \) produces the position (density) distribution in \( x \). \( ux \) produces the velocity distribution in \( x \), \( \sqrt{ux^2+uy^2+uz^2} \) produces the speed distribution. The default value of the histogram without normalization is \( f = \sum_{i=1}^{N} w_i \), where \( \sum_{i=1}^{N} \) is the sum over \( N \) particles in that bin, \( w_i \) denotes the weight of the \( i \)th particle.

- `<reduced_diags_name>.bin_number (int > 0)` This is the number of bins used for the histogram.

- `<reduced_diags_name>.bin_max (float)` This is the maximum value of the bins.

- `<reduced_diags_name>.bin_min (float)` This is the minimum value of the bins.

- `<reduced_diags_name>.normalization (optional)` This provides options to normalize the histogram:

  - `unity_particle_weight` uses unity particle weight to compute the histogram, such that the values of the histogram are the number of counted macroparticles in that bin, i.e. \( f = \sum_{i=1}^{N} 1 \), \( N \) is the number of particles in that bin.

  - `max_to_unity` will normalize the histogram such that its maximum value is one.

  - `area_to_unity` will normalize the histogram such that the area under the histogram is one, so the histogram is also the probability density function.

  If nothing is provided, the macroparticle weight will be used to compute the histogram, and no normalization will be done.

- `<reduced_diags_name>.filter_function(t,x,y,z,ux,uy,uz) (string) optional` Users can provide an expression returning a boolean for whether a particle is taken into account when calculating the histogram (the exact test is whether the return value is \( > 0.5 \)). \( t \) represents the physical time in seconds during the simulation. \( x, y, z \) represent particle positions in the unit of meter. \( ux, uy, uz \) represent particle velocities in the unit of \( \gamma v/c \), \( \gamma v/c \) is the particle velocity normalized by the speed of light.
where $\gamma$ is the Lorentz factor, $v/c$ is the particle velocity normalized by the speed of light. E.g. if provided $(x>0.0)*(u_\perp<10.0)$ only those particles located at positions $x$ greater than 0, and those having velocity $u_\perp$ less than 10, will be taken into account when calculating the histogram.

The output columns are values of the 1st bin, the 2nd bin, ..., the nth bin. An example input file and a loading python script of using the histogram reduced diagnostics are given in Examples/Tests/initial_distribution/.

- **ParticleExtrema** This type computes the minimum and maximum values of particle position, momentum, gamma, weight, and the $\chi$ parameter for QED species.
  
  `<reduced_diags_name>.species` must be provided, such that the diagnostics are done for this species only.

  The output columns are minimum and maximum position $x$, $y$, $z$; minimum and maximum momentum $p_x$, $p_y$, $p_z$; minimum and maximum gamma $\gamma$; minimum and maximum weight $w$; minimum and maximum $\chi$.

  Note that when the QED parameter $\chi$ is computed, field gather is carried out at every output, so the time of the diagnostic may be long depending on the simulation size.

- `<reduced_diags_name>.intervals` *(string)* Using the *Intervals Parser* syntax, this string defines the timesteps at which reduced diagnostics are written to file.

- `<reduced_diags_name>.path` *(string)* optional (default ./diags/reducedfiles/) The path that the output file will be stored.

- `<reduced_diags_name>.extension` *(string)* optional (default *txt*) The extension of the output file.

- `<reduced_diags_name>.separator` *(string)* optional (default a *whitespace*) The separator between row values in the output file. The default separator is a whitespace.

### 3.2.14 Lookup tables and other settings for QED modules

Lookup tables store pre-computed values for functions used by the QED modules. **This feature requires to compile with QED=TRUE (and also with QED_TABLE_GEN=TRUE for table generation)**

- **qed_bw.lookup_table_mode** *(string)* There are three options to prepare the lookup table required by the Breit-Wheeler module:

  - *builtin*: a built-in table is used (Warning: the table gives reasonable results but its resolution is quite low).

  - *generate*: a new table is generated. This option requires Boost math library (version $\geq$ 1.66) and to compile with QED_TABLE_GEN=TRUE. All the following parameters must be specified (table 1 is used to evolve the optical depth of the photons, while table 2 is used for pair generation):

    - `qed_bw.tab_ddnt_chimin` *(float)*: minimum chi parameter for lookup table 1 (used for the evolution of the optical depth of the photons)
    - `qed_bw.tab_ddnt_chimax` *(float)*: maximum chi parameter for lookup table 1
    - `qed_bw.tab_ddnt_howmany` *(int)*: number of points to be used for lookup table 1
    - `qed_bw.tab_pair_chimin` *(float)*: minimum chi parameter for lookup table 2 (used for pair generation)
    - `qed_bw.tab_pair_chimax` *(float)*: maximum chi parameter for lookup table 2
    - `qed_bw.tab_pair_howmany` *(int)*: number of points to be used for chi axis in lookup table 2
* ` qed_bw.tab_pair_frac_how_many (int)`: number of points to be used for the second axis in lookup table 2 (the second axis is the ratio between the quantum parameter of the less energetic particle of the pair and the quantum parameter of the photon).

* ` qed_bw.save_table_in (string)`: where to save the lookup table
  - load: a lookup table is loaded from a pre-generated binary file. The following parameter must be specified:
    * ` qed_bw.load_table_from (string)`: name of the lookup table file to read from.

**` qed_qs.lookup_table_mode (string)`:** There are three options to prepare the lookup table required by the Quantum Synchrotron module:

  - builtin: a built-in table is used (Warning: the table gives reasonable results but its resolution is quite low).
  - generate: a new table is generated. This option requires Boost math library (version >= 1.66) and to compile with QED_TABLE_GEN=TRUE. All the following parameters must be specified (table 1 is used to evolve the optical depth of the particles, while table 2 is used for photon emission):
    * ` qed_qs.tab_dndt_chi_min (float)`: minimum chi parameter for lookup table 1 (used for the evolution of the optical depth of electrons and positrons)
    * ` qed_qs.tab_dndt_chi_max (float)`: maximum chi parameter for lookup table 1
    * ` qed_qs.tab_dndt_how_many (int)`: number of points to be used for lookup table 1
    * ` qed_qs.tab_em_chi_min (float)`: minimum chi parameter for lookup table 2 (used for photon emission)
    * ` qed_qs.tab_em_chi_max (float)`: maximum chi parameter for lookup table 2
    * ` qed_qs.tab_em_chi_how_many (int)`: number of points to be used for chi axis in lookup table 2
    * ` qed_qs.tab_em_frac_how_many (int)`: number of points to be used for the second axis in lookup table 2 (the second axis is the ratio between the quantum parameter of the photon and the quantum parameter of the charged particle).
    * ` qed_qs.tab_em_frac_min (float)`: minimum value to be considered for the second axis of lookup table 2
    * ` qed_bw.save_table_in (string)`: where to save the lookup table
  - load: a lookup table is loaded from a pre-generated binary file. The following parameter must be specified:
    * ` qed_qs.load_table_from (string)`: name of the lookup table file to read from.

**` qed_bw.chi_min (float)`:** minimum chi parameter to be considered by the Breit-Wheeler engine (suggested value: 0.01)

**` qed_qs.chi_min (float)`:** minimum chi parameter to be considered by the Quantum Synchrotron engine (suggested value: 0.001)

**` qed_qs.photon_creation_energy_threshold (float)`:** optional (default 2) Energy threshold for photon particle creation in $\gamma m_e c^2$ units.

**` warpx.do_qed_schwinger (bool)`:** optional (default 0) If this is 1, Schwinger electron-positron pairs can be generated in vacuum in the cells where the EM field is high enough. Activating the Schwinger process requires the code to be compiled with QED=TRUE and PICSAR. If ` warpx.do_qed_schwinger = 1`, Schwinger product species must be specified with ` qed_schwinger.ele_product_species ` and ` qed_schwinger.pos_product_species `. Schwinger process requires either ` warpx.do_nodal=1 ` or

---

### 3.2. Input Parameters

---
algo.field_gathering=momentum-conserving (so that different field components are computed at the same location in the grid) and does not currently support mesh refinement, cylindrical coordinates or single precision.

- **qed_schwinger.ele_product_species** *(string)* If Schwinger process is activated, an electron product species must be specified (the name of an existing electron species must be provided).

- **qed_schwinger.pos_product_species** *(string)* If Schwinger process is activated, a positron product species must be specified (the name of an existing positron species must be provided).

- **qed_schwinger.y_size** *(float; in meters)* If Schwinger process is activated with \( \text{DIM}=2D \), a transverse size must be specified. It is used to convert the pair production rate per unit volume into an actual number of created particles. This value should correspond to the typical transverse extent for which the EM field has a very high value (e.g. the beam waist for a focused laser beam).

- **qed_schwinger.xmin, ymin, zmin** and **qed_schwinger.xmax, ymax, zmax** *(float)* optional (default unlimited)
  When **qed_schwinger.xmin** and **qed_schwinger.xmax** are set, they delimit the region within which Schwinger pairs can be created. The same is applicable in the other directions.

- **qed_schwinger.threshold_poisson_gaussian** *(integer)* optional (default 25)
  If the expected number of physical pairs created in a cell at a given timestep is smaller than this threshold, a Poisson distribution is used to draw the actual number of physical pairs created. Otherwise a Gaussian distribution is used. Note that, regardless of this parameter, the number of macroparticles created is at most one per cell per timestep per species (with a weight corresponding to the number of physical pairs created).

### 3.2.15 Checkpoints and restart

WarpX supports checkpoints/restart via AMReX. The checkpoint capability can be turned with regular diagnostics: `<diag_name>.format = checkpoint`.

- **amr.restart** *(string)* Name of the checkpoint file to restart from. Returns an error if the folder does not exist or if it is not properly formatted.

### 3.2.16 Intervals parser

WarpX can parse time step interval expressions of the form `start:stop:period`, e.g. `1:2:3, 4::, 5:6, :, ::10`. A comma is used as a separator between groups of intervals, which we call slices. The resulting time steps are the union set of all given slices. White spaces are ignored. A single slice can have 0, 1 or 2 colons :, just as numpy slices, but with inclusive upper bound for stop.

- For 0 colon the given value is the period

- For 1 colon the given string is of the type `start:stop`

- For 2 colons the given string is of the type `start:stop:period`

Any value that is not given is set to default. Default is 0 for the start, `std::numeric_limits<int>::max()` for the stop and 1 for the period. For the 1 and 2 colon syntax, actually having values in the string is optional (this means that `::5`, `100::10` and `100:` are all valid syntaxes).

All values can be expressions that will be parsed in the same way as other integer input parameters.

**Examples**

- **something_intervals = 50** -> do something at timesteps 0, 50, 100, 150, etc. (equivalent to **something_intervals = ::50**)

- **something_intervals = 300:600:100** -> do something at timesteps 300, 400, 500 and 600.

- **something_intervals = 300::50** -> do something at timesteps 300, 350, 400, 450, etc.

• `something_intervals = :` or `something_intervals = ::` -> do something at every timestep.


This is essentially the python slicing syntax except that the stop is inclusive (0:100 contains 100) and that no colon means that the given value is the period.

Note that if a given period is zero or negative, the corresponding slice is disregarded. For example, `something_intervals = -1` deactivates `something` and `something_intervals = ::-1,100:1000:25` is equivalent to `something_intervals = 100:1000:25`.

3.3 Python (PICMI)

WarpX uses the PICMI standard for its Python input files. See PICMI information and source code.

WarpX can be run in one of two modes. It can run as a preprocessor, using the Python input file to generate an input file to be used by the C++ version, or it can be run directly from Python.

In either mode, if using a virtual environment, be sure to activate it before compiling and running WarpX.

Example input files can be found in the examples section. The examples support running in both modes by commenting and uncommenting the appropriate lines.

WarpX requires Python version 3.6 or newer.

3.3.1 Using Python input as a preprocessor

In this case, only the pure Python version needs to be installed, as described here.

In order to run a new simulation:

• Create a new directory, where the simulation will be run.

• Add a Python script in the directory.

The input file should have the line like `sim.write_input_file(file_name = 'inputs_from_PICMI')` which runs the preprocessor, generating the AMReX inputs file.

• Run the script with Python:

```python
python <python_script>
```

where `<python_script>` is the name of the script. This creates the WarpX input file that you can run as normal with the WarpX executable.
3.3.2 Running WarpX directly from Python

For this, a full Python installation of WarpX is required, as described in the install documentation (developers).

In order to run a new simulation:

- Create a new directory, where the simulation will be run.
- Add a Python script in the directory.

The input file should have the line `sim.step()` which runs the simulation.

- Run the script with Python:

  ```bash
  mpirun -np <n_ranks> python <python_script>
  ```

  where `<n_ranks>` is the number of MPI ranks used, and `<python_script>` is the name of the script.

3.4 Examples

This section allows you to download input files that correspond to different physical situations.

We provide two kinds of inputs:

- AMReX inputs files, with parameters described here,
- PICMI python input files, with parameters described here.

For a complete list of all example input files, have a look at our Examples/ directory. It contains folders and subfolders with self-describing names that you can try. All these input files are automatically tested, so they should always be up-to-date.

3.4.1 Beam-driven electron acceleration

AMReX inputs:

- 2D case
- 2D case in boosted frame
- 3D case in boosted frame

PICMI:

- Without mesh refinement
- With mesh refinement

3.4.2 Laser-driven electron acceleration

AMReX inputs:

- 1D case
- 2D case
- 2D case in boosted frame
- 3D case
- RZ case
PICMI (Python) scripts:

- 1D case
- 2D case with mesh refinement
- 3D case
- RZ case

### 3.4.3 Plasma mirror

2D case

### 3.4.4 Laser-ion acceleration

2D case

**Note:** The resolution of this 2D case is extremely low by default. You will need a computing cluster for adequate resolution of the target density, see comments in the input file.

### 3.4.5 Uniform plasma

2D case 3D case

### 3.4.6 Capacitive discharge

The Monte-Carlo collision (MCC) model can be used to simulate electron and ion collisions with a neutral background gas. In particular this can be used to study capacitive discharges between parallel plates. The implementation has been tested against the benchmark results from Turner et al. in *Phys. Plasmas* 20, 013507, 2013. The figure below shows a comparison of the ion density as calculated in WarpX (in June 2021) compared to the literature results (which can be found [here](#)).

![Benchmark of WarpX PIC-MCC implementation](#)

An input file to reproduce the benchmark calculations is linked below. To run a given case `-n`, from 1 to 4, execute:

```bash
```

3.4. Examples
python3 PICMI_inputs_1d.py -n 1

Once the simulation completes an output file `avg_ion_density.npy` will be created which can be compared to the literature results as in the plot above. Running case 1 on 4 processors takes roughly 20 minutes to complete.

- input file

**Note:** This example needs additional calibration data for cross sections. Download this data alongside your inputs file and update the paths in the inputs file:

```
git clone https://github.com/ECP-WarpX/warpx-data.git
```

### 3.4.7 Test cases

PICMI (Python) test cases included that can be used as a reference:

- Gaussian beam
- Langmuir plasma wave test in 3d
- Langmuir plasma wave test in RZ
- Langmuir plasma wave test in 2D

### 3.4.8 Manipulating fields via Python

An example of using Python to access the simulation charge density, solve the Poisson equation (using superLU) and write the resulting electrostatic potential back to the simulation is given in the input file below. This example uses the `fields.py` module included in the `pywarpx` library.

- Direct Poisson solver example

An example of initializing the fields by accessing their data through Python, advancing the simulation for a chosen number of time steps, and plotting the fields again through Python. The simulation runs with 128 regular cells, 8 guard cells, and 10 PML cells, in each direction. Moreover, it uses div(E) and div(B) cleaning both in the regular grid and in the PML and initializes all available electromagnetic fields (E,B,F,G) identically.

- Unit pulse with PML

### 3.5 In-Depth: PWFA

As described in the *Introduction*, one of the key applications of the WarpX exascale computing platform is in modelling future, compact and economic plasma-based accelerators. In this section we describe the simulation setup of a realistic *electron beam driven plasma wakefield accelerator* (PWFA) configuration. For illustration purposes the setup can be explored with *WarpX* using the example input file `PWFA`.

The simulation setup consists of 4 particle species: drive beam (driver), witness beam (beam), plasma electrons (plasma_e), and plasma ions (plasma_p). The species physical parameters are summarized in the following table.
<table>
<thead>
<tr>
<th>Species</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver</td>
<td>$\gamma = 48923; N = 2 \times 10^8; z = 4.0 \text{ m}; x = 2.0 \text{ m}$</td>
</tr>
<tr>
<td>beam</td>
<td>$\gamma = 48923; N = 6 \times 10^5; z = 1.0 \text{ mm}; x = 0.5 \text{ m}$</td>
</tr>
<tr>
<td>plasma_e</td>
<td>$n = 1 \times 10^23 \text{ m}^{-3}; w = 70 \text{ m}; lr = 8 \text{ mm}; L = 200 \text{ mm}$</td>
</tr>
<tr>
<td>plasma_p</td>
<td>$n = 1 \times 10^23 \text{ m}^{-3}; w = 70 \text{ m}; lr = 8 \text{ mm}; L = 200 \text{ mm}$</td>
</tr>
</tbody>
</table>

Where $\gamma$ is the beam relativistic Lorentz factor, $N$ is the number of particles, and $x$, $y$, $z$ are the beam widths (root-mean-squares of particle positions) in the transverse ($x,y$) and longitudinal directions.

The plasma, of total length $L$, has a density profile that consists of a $lr$ long linear up-ramp, ranging from 0 to peak value $n$, is uniform within a transverse width of $w$ and after the up-ramp.

With this configuration the driver excites a nonlinear plasma wake and drives the bubble depleted of plasma electrons where the beam accelerates, as can be seen in Fig. [fig:PWFA].

![Image](image_url)

**Fig. 3.1: [fig:PWFA]** Plot of the driver (blue), beam (red) and plasma_e (green) electron macroparticle distribution at the time step 1000 of the example simulation. These are overlapping the 2D plot of the longitudinal electric field showing the accelerating/deccelerating (red/blue) regions of the plasma bubble.

Listed below are the key arguments and best-practices relevant for choosing the pwfa simulation parameters used in the example.
### 3.5.1 2D Geometry

2D cartesian (with longitudinal direction z and transverse x) geometry simulations can give valuable physical and numerical insight into the simulation requirements and evolution. At the same time it is much less time consuming than the full 3D cartesian or cylindrical geometries.

### 3.5.2 Finite Difference Time Domain

For standard plasma wakefield configurations, it is possible to model the physics correctly using the Particle-In-Cell (PIC) Finite Difference Time Domain (FDTD) algorithms (*Particle-in-Cell Method*). If the simulation contains localised extremely high intensity fields, however, numerical instabilities might arise, such as the numerical Cherenkov instability (*Moving window and optimal Lorentz boosted frame*). In that case, it is recommended to use the Pseudo Spectral Analytical Time Domain (PSATD) or the Pseudo-Spectral Time-Domain (PSTD) algorithms. In the example we are describing, it is sufficient to use FDTD.

### 3.5.3 Cole-Karkkainen solver with Cowan coefficients

There are two FDTD Maxwell field solvers that compute the field push implemented in WarpX: the Yee and Cole-Karkkainen solver with Cowan coefficients (CKC) solvers. The later includes a modification that allows the numerical dispersion of light in vacuum to be exact, and that is why we choose CKC for the example.

### 3.5.4 Lorentz boosted frame

WarpX simulations can be done in the laboratory or Lorentz-boosted frames. In the laboratory frame, there is typically no need to model the plasma ions species, since they are mainly stationary during the short time scales associated with the motion of plasma electrons. In the boosted frame, that argument is no longer valid, as ions have relativistic velocities. The boosted frame still results in a substantial reduction to the simulation computational cost.

**Note:** Even if the simulations uses the boosted frame, most of its input file parameters are defined in respect to the laboratory frame.

We recommend that you design your numerical setup so that the width of the box is not significantly narrower than the distance from 0 to its right edge (done, for example, by setting the right edge equal to 0).

### 3.5.5 Moving window

To avoid having to simulate the whole 0.2 mm of plasma with the high resolution that is required to model the beam and plasma interaction correctly, we use the moving window. In this way we define a simulation box (grid) with a fixed size that travels at the speed-of-light ($c$), i.e. follows the beam.

**Note:** When using moving window the option of continuous injection needs to be active for all particles initialized outside of the simulation box.
3.5.6 Resolution

Longitudinal and transverse resolutions (i.e. number and dimensions of the PIC grid cells) should be chosen to accurately describe the physical processes taking place in the simulation. Convergence scans, where resolution in both directions is gradually increased, should be used to determine the optimal configuration. Multiple cells per beam length and width are recommended (our illustrative example resolution is coarse).

Note: To avoid spurious effects, in the boosted frame, we consider the constraint that the transverse cell size should be larger than the transverse one. Translating this condition to the cell transverse ($d_x$) and longitudinal dimensions ($d_z$) in the laboratory frame leads to: $d_x > (d_z(1 + \beta_b)\gamma_b)$, where $\beta_b$ is the boosted frame velocity in units of $c$.

3.5.7 Time step

The time step (\(dt\)) is used to iterate over the main PIC loop and is computed by WarpX differently depending on the Maxwell field FDTD solvers used:

- For Yee is equal to the CFL parameter chosen in the input file (Input Parameters) times the Courant–Friedrichs–Lewy condition (CFL) that follows the analytical expression in Particle-in-Cell Method
- For CKC is equal to CFL times the minimum between the boosted frame cell dimensions

where CFL is chosen to be below unity and set an optimal trade-off between making the simulation faster and avoiding NCI and other spurious effects.

3.5.8 Duration of the simulation

To determine the total number of time steps of the simulation, we could either set the \(<\text{zmax_plasma_to_compute_max_step}>\) parameter to the end of the plasma (\(z_{\text{end}}\)), or compute it using:

- boosted frame edge of the simulation box, corner = \(l_c/(1 - \beta_b)\gamma_b)\)
- time of interaction in the boosted frame, \(T = \frac{z_{\text{end}}/\gamma_b - \text{corner}}{c(1 + \beta_b)}\)
- total number of iterations, \(i_{\text{max}} = T/dt\)

where \(l_c\) is the position of the left edge of the simulation box (in respect to propagation direction).

3.5.9 Plotfiles and snapshots

WarpX allows the data to be stored in different formats, such as plotfiles (following the yt guidelines), hdf5 and openPMD (following its standard). In the example, we are dumping plotfiles with boosted frame information on the simulation particles and fields. We are also requesting back transformed diagnostics that transform that information back to the laboratory frame. The diagnostics results are analysed and stored in snapshots at each time step and so it is best to make sure that the run does not end before filling the final snapshot.
3.5.10 Maximum grid size and blocking factor

These parameters are carefully chosen to improve the code parallelization, load-balancing and performance (Input Parameters) for each numerical configuration. They define the smallest and largest number of cells that can be contained in each simulation box and are carefully defined in the AMReX documentation.

3.6 Workflows

This section collects typical user workflows and best practices for WarpX.

3.6.1 Parallelization in WarpX

When running a simulation, the domain is split into independent rectangular sub-domains (called grids). This is the way AMReX, a core component of WarpX, handles parallelization and/or mesh refinement. Furthermore, this decomposition makes load balancing possible: each MPI rank typically computes a few grids, and a rank with a lot of work can transfer one or several grids to their neighbors.

A user does not specify this decomposition explicitly. Instead, the user gives hints to the code, and the actual decomposition is determined at runtime, depending on the parallelization. The main user-defined parameters are `amr.max_grid_size` and `amr.blocking_factor`.

**AMReX max_grid_size and blocking_factor**

- `amr.max_grid_size` is the maximum number of cells per grid along each direction (default `amr.max_grid_size=32` in 3D).
- `amr.blocking_factor`: is the minimum number of cells per grid along each direction (default `amr.blocking_factor=8`). Note that both the domain (at each level) and `max_grid_size` must be divisible by `blocking_factor`.

**Note:** You can use the parameters above if you want the same number of cells in all directions. Or you can set `amr.max_grid_size_x`, `amr.max_grid_size_y` and `amr.max_grid_size_z`; `amr.blocking_factor_x`, `amr.blocking_factor_y` and `amr.blocking_factor_z` to different numbers of cells.

The total number of grids is determined using those two restrictions and the number of ranks used to run the simulation. You can visit AMReX documentation for more information on the two parameters.

These parameters can have a dramatic impact on the code performance. Each grid in the decomposition is surrounded by guard cells, thus increasing the amount of data, computation and communication. Hence having a too small `max_grid_size`, may ruin the code performance.

On the other hand, a too-large `max_grid_size` is likely to result in a single grid per MPI rank, thus preventing load balancing. By setting these two parameters, the user wants to give some flexibility to the code while avoiding pathological behaviors.

For more information on this decomposition, see the Gridding and Load Balancing page on AMReX documentation. For specific information on the dynamic load balancer used in WarpX, visit the Load Balancing page on the AMReX documentation.

The best values for these parameters strongly depends on a number of parameters, among which numerical parameters:

- Algorithms used (Maxwell/spectral field solver, filters, order of the particle shape factor)
• Number of guard cells (that depends on the particle shape factor and the type and order of the Maxwell solver, the filters used, etc.)
• Number of particles per cell, and the number of species

and MPI decomposition and computer architecture used for the run:
• GPU or CPU
• Number of OpenMP threads
• Amount of high-bandwidth memory.

Because these parameters put additional constraints on the domain size for a simulation, it can be cumbersome to calculate the number of cells and the physical size of the computational domain for a given resolution. This Python script does it automatically.

When using the RZ spectral solver, the values of `amr.max_grid_size` and `amr.blocking_factor` are constrained since the solver requires that the full radial extent be within a each block. For the radial values, any input is ignored and the max grid size and blocking factor are both set equal to the number of radial cells. For the longitudinal values, the blocking factor has a minimum size of 8, allowing the computational domain of each block to be large enough relative to the guard cells for reasonable performance, but the max grid size and blocking factor must also be small enough so that there will be at least one block per processor. If max grid size and/or blocking factor are too large, they will be silently reduced as needed. If there are too many processors so that there is not enough blocks for the number processors, WarpX will abort.

### 3.6.2 Profiling the code

Profiling allows us to find the bottle-necks of the code as it is currently implemented. Bottle-necks are the parts of the code that may delay the simulation, making it more computationally expensive. Once found, we can update the related code sections and improve its efficiency. Profiling tools can also be used to check how load balanced the simulation is, i.e. if the work is well distributed across all MPI ranks used. Load balancing can be activated in WarpX by setting input parameters, see the parallelization input parameter section.

#### Profiling with AMReX's built-in profiling tools

By default, WarpX uses the AMReX baseline tool, the TINYPROFILER, to evaluate the time information for different parts of the code (functions) between the different MPI ranks. The results, timers, are stored into four tables in the standard output, stdout, that are located below the simulation steps information and above the warnings regarding unused input file parameters (if there were any).

The timers are displayed in tables for which the columns correspond to:

- name of the function
- number of times it is called in total
- minimum of time spent exclusively/inclusively in it, between all ranks
- average of time, between all ranks
- maximum time, between all ranks
- maximum percentage of time spent, across all ranks

If the simulation is well load balanced the minimum, average and maximum times should be identical.

The top two tables refer to the complete simulation information. The bottom two are related to the Evolve() section of the code (where each time step is computed).
Each set of two timers show the exclusive, top, and inclusive, bottom, information depending on whether the time spent in nested sections of the codes are included.

For more detailed information please visit the AMReX profiling documentation.

Note: When creating performance-related issues on the WarpX GitHub repo, please include Tiny Profiler tables (besides the usual issue description, input file and submission script), or (even better) the whole standard output.

3.6.3 Debugging the code

Sometimes, the code does not give you the result that you are expecting. This can be due to a variety of reasons, from misunderstandings or changes in the input parameters, system specific quirks, or bugs. You might also want to debug your code as you implement new features in WarpX during development.

This section gives a step-by-step guidance on how to systematically check what might be going wrong.

Debugging Workflow

Try the following steps to debug a simulation:

1. Check the output text file, usually called output.txt: are there warnings or errors present?
2. On an HPC system, look for the job output and error files, usually called WarpX.e... and WarpX.o.... Read long messages from the top and follow potential guidance.
3. If your simulation already created output data files: Check if they look reasonable before the problem occurred; are the initial conditions of the simulation as you expected? Do you spot numerical artifacts or instabilities that could point to missing resolution or unexpected/incompatible numerical parameters?
4. Did the job output files indicate a crash? Check the Backtrace.<mpirank> files for the location of the code that triggered the crash. Backtraces are read from bottom (high-level) to top (most specific line that crashed).
5. In case of a crash, Backtraces can be more detailed if you re-compile with debug flags: for example, try compiling with -DCMAKE_BUILD_TYPE=RelWithDebInfo (some slowdown) or even -DCMAKE_BUILD_TYPE=Debug (this will make the simulation way slower) and rerun.
6. If debug builds are too costly, try instead compiling with -DAMReX_ASSERTIONS=ON to activate more checks and rerun.
7. If the problem looks like a memory violation, this could be from an invalid field or particle index access. Try compiling with -DAMReX_BOUND_CHECK=ON (this will make the simulation very slow), and rerun.
8. If the problem looks like a random memory might be used, try initializing memory with signaling Not-a-Number (NaN) values through the runtime option fab.init_snan = 1. Further useful runtime options are amrex.fpe_trap_invalid, amrex.fpe_trap_zero and amrex.fpe_trap_overflow (see details in the AMReX link below).
9. On Nvidia GPUs, if you suspect the problem might be a race condition due to a missing host / device synchronization, set the environment variable export CUDA_LAUNCH_BLOCKING=1 and rerun.
10. Consider simplifying your input options and re-adding more options after having found a working baseline.

For more information, see also the AMReX Debugging Manual.

Last but not least: the community of WarpX developers and users can help if you get stuck. Collect your above findings, describe where and what you are running and how you installed the code, describe the issue you are seeing with details and input files used and what you already tried. Can you reproduce the problem with a smaller setup (less parallelism and/or less resolution)? Report these details in a WarpX GitHub issue.
### Debuggers

See the AMReX debugger section on additional runtime parameters to

- disable backtraces
- rethrow exceptions
- avoid AMReX-level signal handling

You will need to set those runtime options to work directly with debuggers.

#### 3.6.4 Run LibEnsemble on WarpX

LibEnsemble is a library to coordinate the concurrent evaluation of dynamic ensembles of calculations. While a WarpX simulation can provide insight in some physics, it remains a single point evaluation in the space of parameters. If you have a simulation ready for use, but would like to (i) scan over some input parameters uniformly for, e.g., a tolerance study, or (ii) have a random evaluation of the space of input parameters within a given span or (iii) tune some input parameters to optimize an output parameter, e.g., beam emittance, energy spread, etc., LibEnsemble provides these capabilities and will take care of tasks monitoring with fault tolerance on multiple platforms (LibEnsemble targets modern HPC platforms like Summit).

Scripts to run LibEnsemble on WarpX simulations can be found in `WarpX/Tools/LibEnsemble/`. This documentation does not aim at giving a training on LibEnsemble, so please refer to the LibEnsemble documentation for technical details.

#### WarpX example problem for LibEnsemble study

The WarpX example is built on a 2D input file (so that 1 simulation take < 1 min) of a 2-stage laser-wakefield simulation in a boosted frame. It aims at optimizing emittance preservation in the coupling between two consecutive plasma accelerator stages. Each stage accelerates an externally-injected electron beam, which charge has been tuned to show a decent amount of Ez field flattening due to longitudinal beam loading. Each stage has a parabolic transverse profile to guide the laser pulse, and a uniform longitudinal profile with cos-shape ramps and the entrance and at the exit. A fresh laser pulse is introduced at the entrance of each stage and deleted at the exit. The beam transverse distribution is matched to the first stage and the beam is injected at the beginning of the plateau of the first stage, so that the emittance is conserved in the first stage. The two stages are separated by a few-cm gap, and a focusing lens is placed in-between. Note that this is a very low resolution simulation to show an example, so it is not close to numerical convergence.

In this example, we allow LibEnsemble to tune four input parameters:

- Length of the downramp of the first stage
- Longitudinal position of the focusing lens (between the two stages)
- Strength of the focusing lens
- Length of the downramp of the second stage

The output parameter that LibEnsemble minimizes is the beam emittance at the exit of the second stage, while making sure the charge loss is small.

The scripts provided can run on a local machine or on the Summit supercomputer at OLCF. Two options are available: random sampling of parameter space or optimization on the output parameter. For the latter, we are using the Asynchronously Parallel Optimization Solver for finding Multiple Minima APOSMM method provided by LibEnsemble.
Install LibEnsemble

Besides a working WarpX executable, you have to install libEnsemble and its dependencies.
You can either install all packages via conda (recommended),

```
conda install -c conda-forge libensemble matplotlib numpy scipy yt
```

or try to install the same dependencies via pip (pick one or the other; note our installation details on Summit):

```
libensemble
matplotlib
nlopt
numpy
pytest
scipy
yt
```

What's in Tools/LibEnsemble?

See the LibEnsemble User Guide for an overview of LibEnsemble concepts. In a nutshell, a user needs to define

- A generator function `gen_f` that will generate inputs for the simulation, which can be done uniformly, randomly or using an optimizer. The generator output, i.e., the simulation input, is called 'x'. The generator is provided by LibEnsemble. When the generator is an optimizer, it takes the simulation output called 'f' as an input.

- A simulation function `sim_f` that will take 'x' as an input and return a single output value 'f'. In our example, `sim_f` modifies the WarpX input file depending on 'x', launches a WarpX simulation and reads the simulation output plotfile to extract 'f'.

- An allocator function `alloc_f` that will feed available workers with tasks. This is provided by LibEnsemble.

The files in `Tools/LibEnsemble/` are:

- `run_libensemble_on_warpx.py` This is the main LibEnsemble script. It imports `gen_f` and `alloc_f` from LibEnsemble, `sim_f` from file `warpx_simf.py` (see below), defines dictionaries for parameters of each of these objects (`gen_specs` includes lower and upper bound of each element in the input array 'x', `alloc_specs` and `sim_specs` respectively) and runs LibEnsemble.

- `warpx_simf.py` defines the `sim_f` function called `run_warpx`:

  ```python
  warpx_simf.run_warpx(H, persis_info, sim_specs, libE_info)
  ```

  This function runs a WarpX simulation and returns quantity 'f' as well as other physical quantities measured in the run for convenience. Status check is done periodically on the simulation, provided by LibEnsemble.

- `sim/inputs` WarpX input file. Some of its parameters are modified in `run_warpx`.

- `write_sim_input.py` (util) update one WarpX input file depending on values in 'x' for this run.

  ```python
  write_sim_input.write_sim_input(input_file, x_values)
  ```

  This function modifies input_file to replace the values of some parameters by values provided in x_values.

(continues on next page)
Parameters
----------

**input_file** : WarpX input file.

**x_values** : Specific value of input x for this run.

**read_sim_output.py** (util) Read WarpX plotfile and return 'f'.

```python
read_sim_output.read_sim_output(workdir)
```

Return optimizing quantity 'f' and other parameters for convenience.

Parameters
----------

**workdir** : Path to directory where the simulation ran.

**plot_results.py** (util) Read LibEnsemble output files .npy and .pickle and plot output 'f' (and other output, just for convenience) as a function of input from all simulations.

**all_machine_specs.py** (util) Dictionaries of machine-specific parameters. For convenience, the maximum number of WarpX runs is defined there.

**summit_submit_mproc.sh** Submission script for LibEnsemble+WarpX on Summit. Make sure to edit this file and add your project ID for allocations.

### Run the example

On Summit or for a local run, LibEnsemble can run with a Random Number Generator (easiest, good as a first try) or with an optimizer (requires python package nlopt). This is set by the variable generator_type in `run_libensemble_on_warpx.py`. We hereafter assume that all Python modules required are installed and that a WarpX executable is available.

### Run locally

Adjust the `local_specs` dictionary in `all_machine_specs.py` to fix the path to the WarpX executable (and optionally change the number of cores and OpenMP threads), and run

```bash
python run_libensemble_on_warpx.py --comms local --nworkers 3
```

This is adapted to a 4-core machine, as it will use:

- 1 process to run LibEnsemble
- 1 process (among the 3 workers) to run the generator
- 2 processes to run 2 concurrent simulations

3.6. Workflows 115
Run on Summit at OLCF

- cp -r $HOME/warpx/Tools/LibEnsemble/* sim_directory
- modify run_libensemble_on_warpx.py to have machine = 'summit'
- modify all_machine_specs.py to put the right path to the WarpX executable
- modify summit_submit_mproc.sh to set LIBE_PLOTS to false and set the project ID
- bsub summit_submit_mproc.sh

3.6.5 Visualizing a distribution mapping

WarpX provides via reduced diagnostics an output LoadBalanceCosts, which allows for visualization of a simulation’s distribution mapping and computational costs. Here we demonstrate the workflow for generating this data and using it to plot distribution mappings and load balance costs.

Generating the data

To generate ‘Load Balance Costs’ reduced diagnostics output, WarpX should be run with the following lines added to the input file (the name of the reduced diagnostics file, LBC, and interval in steps to output reduced diagnostics data, 100, may be changed as needed):

```python
warpx.reduced_diags_names = LBC
LBC.type = LoadBalanceCosts
LBC.intervals = 100
```

The line `warpx.reduced_diags_names = LBC` sets the name of the reduced diagnostics output file to `LBC`. The next line `LBC.type = LoadBalanceCosts` tells WarpX that the reduced diagnostics is a `LoadBalanceCosts` diagnostic, and instructs WarpX to record costs and rank layouts. The final line, `LBC.intervals = 100`, controls the interval for output of this reduced diagnostic’s data.

Loading and plotting the data

After generating data (called LBC_knapsack.txt and LBC_sfc.txt in the example below), the following Python code, along with a helper class in plot_distribution_mapping.py can be used to read the data:

```python
# Math
import numpy as np
import random

# Plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import ListedColormap
from mpl_toolkits.axes_grid1 import make_axes_locatable

# Data handling
import plot_distribution_mapping as pdm
```

(continues on next page)
sim_knapsack = pdm.SimData('LBC_knapsack.txt', # Data directory
                          [2800]       # Files to process
                          )
sim_sfc = pdm.SimData('LBC_sfc.txt', [2800])

# Set reduced diagnostics data for step 2800
for sim in [sim_knapsack, sim_sfc]: sim(2800)

For 2D data, the following function can be used for visualization of distribution mappings:

```python
# Plotting -- we know beforehand the data is 2D
def plot(sim):
    
    Plot MPI rank layout for a set of `LoadBalanceCosts` reduced diagnostics
    (2D) data.

    Arguments:
sim -- SimData class with data (2D) loaded for desired iteration

    # Make first cmap
cmap = plt.cm.nipy_spectral
cmaplist = [cmap(i) for i in range(cmap.N)][::-1]
unique_ranks = np.unique(sim.rank_arr)
sz = len(unique_ranks)
cmap = mpl.colors.LinearSegmentedColormap.from_list(
    'my_cmap', cmaplist, sz) # create the new map

    # Make cmap from 1 --> 96 then randomize
cmaplist = [cmap(i) for i in range(sz)]
random.Random(6).shuffle(cmaplist)
cmap = mpl.colors.LinearSegmentedColormap.from_list(
    'my_cmap', cmaplist, sz) # create the new map

    # Define the bins and normalize
bounds = np.linspace(0, sz, sz + 1)
norm = mpl.colors.BoundaryNorm(bounds, sz)

    my, mx = sim.rank_arr.shape
xcoord, ycoord = np.linspace(0, mx, mx + 1), np.linspace(0, my, my + 1)
im = plt.pcolormesh(xcoord, ycoord, sim.rank_arr,
                    cmap=cmap, norm=norm)

    # Grid lines
plt.ylabel('$j$')
plt.xlabel('$i$')
plt.minorticks_on()
plt.hlines(ycoord, xcoord[0], xcoord[-1],
          alpha=0.7, linewidth=0.3, color='lightgrey')
plt.vlines(xcoord, ycoord[0], ycoord[-1],
          alpha=0.7, linewidth=0.3, color='lightgrey')
plt.gca().set_aspect('equal')
```

(continues on next page)
# Center rank label

```python
for j in range(my):
    for i in range(mx):
        text = plt.gca().text(i+0.5, j+0.5, int(sim.rank_arr[j][i]),
                               ha="center", va="center",
                               color="w", fontsize=8)
```

# Colorbar

```python
divider = make_axes_locatable(plt.gca())
cax = divider.new_horizontal(size="5%", pad=0.05)
plt.gcf().add_axes(cax)
cb=plt.gcf().colorbar(im, label='rank', cax=cax, orientation="vertical")
minorticks = np.linspace(0, 1, len(unique_ranks) + 1)
```

The function can be used as follows:

```python
fig, axs = plt.subplots(1, 2, figsize=(12, 6))
plt.sca(axs[0])
plt.title('Knapsack')
plot(sim_knapsack)
plt.sca(axs[1])
plt.title('SFC')
plot(sim_sfc)
plt.tight_layout()
```

This generates plots like in [fig:knapsack_sfc_distribution_mapping_2D]:

![Distribution mappings](image)

**Fig. 3.2:** Sample distribution mappings from simulations with knapsack (left) and space-filling curve (right) policies for update of the distribution mapping when load balancing.

Similarly, the computational costs per box can be plotted with the following code:
This generates plots like in [fig:knapsack_sfc_costs_2D]:

Fig. 3.3: Sample computational cost per box from simulations with knapsack (left) and space-filling curve (right) policies for update of the distribution mapping when load balancing.

Loading 3D data works the same as loading 2D data, but this time the cost and rank arrays will be 3 dimensional. Here we load and plot some example 3D data (LBC_3D.txt) from a simulation run on 4 MPI ranks. Particles fill the box from \( k = 0 \) to \( k = 1 \).

```python
sim_3D = pdm.SimData('LBC_3D.txt', [1,2,3])
sim_3D(1)

# Plotting -- we know beforehand the data is 3D
def plot_3D(sim, j0):
    Plot MPI rank layout for a set of 'LoadBalanceCosts' reduced diagnostics (3D) data.
```

(continues on next page)
Arguments:
sim -- SimData class with data (3D) loaded for desired iteration
j0 -- slice along j direction to plot ik slice

# Make first cmap
cmap = plt.cm.viridis
cmaplist = [cmap(i) for i in range(cmap.N)][::-1]
unique_ranks = np.unique(sim.rank_arr)
sz = len(unique_ranks)
cmap = mpl.colors.LinearSegmentedColormap.from_list('my_cmap', cmaplist, sz) # create the new map

# Make cmap from 1 --> 96 then randomize
cmaplist = [cmap(i) for i in range(sz)]
random.Random(6).shuffle(cmaplist)
cmap = mpl.colors.LinearSegmentedColormap.from_list('my_cmap', cmaplist, sz) # create the new map

# Define the bins and normalize
bounds = np.linspace(0, sz, sz + 1)
norm = mpl.colors.BoundaryNorm(bounds, sz)

mz, my, mx = sim.rank_arr.shape
xcoord, ycoord, zcoord = np.linspace(0, mx, mx + 1), np.linspace(0, my, my + 1), np.linspace(0, mz, mz + 1)
im = plt.pcolormesh(xcoord, zcoord, sim.rank_arr[:, j0, :], cmap=cmap, norm=norm)

# Grid lines
plt.ylabel('$k$')
plt.xlabel('$i$')
plt.minorticks_on()
plt.hlines(zcoord, xcoord[0], xcoord[-1], alpha=0.7, linewidth=0.3, color='lightgrey')
plt.vlines(xcoord, zcoord[0], zcoord[-1], alpha=0.7, linewidth=0.3, color='lightgrey')

plt.gca().set_aspect('equal')

# Center rank label
for k in range(mz):
    for i in range(mx):
        text = plt.gca().text(i + 0.5, k + 0.5, int(sim.rank_arr[k][j0][i]), ha='center', va='center', color='red', fontsize=8)

# Colorbar
divider = make_axes_locatable(plt.gca())
cax = divider.new_horizontal(size='5%', pad=0.05)
plt.gcf().add_axes(cax)
cb = plt.colorbar(im, label='rank', cax=cax, orientation='vertical')
ticks = np.linspace(0, 1, len(unique_ranks)+1)
This generates plots like in [fig:distribution_mapping_3D]:

Fig. 3.4: Sample distribution mappings from 3D simulations, visualized as slices in the $i k$ plane along $j$. 

```python
cb.ax.yaxis.set_ticks(ticks)
cb.ax.yaxis.set_ticklabels([0, 1, 2, 3, " "])

fig, axs = plt.subplots(2, 2, figsize=(8, 8))
for j, ax in enumerate(axs.flatten()):
    plt.sca(ax)
    plt.title('j={}'.format(j))
    plot_3D(sim_3D, j)
    plt.tight_layout()
```
3.6.6 Predicting the Number of Guard Cells for PSATD Simulations

When the computational domain is decomposed in parallel subdomains and the pseudo-spectral analytical time-domain (PSATD) method is used to solve Maxwell’s equations (by setting algo.maxwell_solver = psatd in the input file), the number of guard cells used to exchange fields between neighboring subdomains can be chosen based on the extent of the stencil of the leading term in Maxwell’s equations, in Fourier space. A measure of such stencil can be obtained by computing the inverse Fourier transform of the given term along a chosen axis and by averaging the result over the remaining axes in Fourier space. The idea is to look at how quickly such stencils fall off to machine precision, with respect to their extension in units of grid cells, and identify consequently the number of cells after which the stencils will be truncated, with the aim of balancing numerical accuracy and locality. See (Zoni et al., 2021) for reference.

A user can run the Python script Stencil.py, located in ./Tools/DevUtils, in order to compute such stencils and estimate the number of guard cells needed for a given PSATD simulation with domain decomposition. In particular, the script computes the minimum number of guard cells for a given error threshold, that is, the minimum number of guard cells such that the stencil measure is not larger than the error threshold. The user can modify the input parameters set in the main function in order to reproduce the simulation setup. These parameters include: cell size, time step, spectral order, Lorentz boost, whether the PSATD algorithm is based on the Galilean scheme, and error threshold (this is not an input parameter of a WarpX simulation, but rather an empirical error threshold chosen to balance numerical accuracy and locality, as mentioned above).

3.6.7 Archiving

Archiving simulation inputs, scripts and output data is a common need for computational physicists. Here are some popular tools and workflows to make archiving easy.

HPC Systems: HPSS

A very common tape filesystem is HPSS, e.g., on NERSC or OLCF.

- What’s in my archive file system? hsi ls
- Already something in my archive location? hsi ls 2019/cool_campaign/ as usual
- Let’s create a neat directory structure:
  - new directory on the archive: hsi mkdir 2021
  - create sub-dirs per campaign as usual: hsi mkdir 2021/reproduce_paper
- Create an archive of a simulation: htar -cvf 2021/reproduce_paper/sim_042.tar /global/cfs/cdirs/m1234/ahuebl/reproduce_paper/sim_042
  - This copies all files over to the tape filesystem and stores them as a single .tar archive
  - The first argument here will be the new archive .tar file on the archive file system, all following arguments (can be multiple, separated by a space) are locations to directories and files on the parallel file system.
  - Don’t be confused, these tools also create an index .tar.idx file along it; just leave that file be and don’t interact with it
- Restore things:
  - mkdir here_we_restore
  - cd here_we_restore
  - htar -xvf 2021/reproduce_paper/sim_42.tar
* this copies the .tar file back from tape to our parallel filesystem and extracts its content in the current directory

Argument meaning: -c create; -x extract; -v verbose; -f tar filename. That’s it, folks!

**Note:** Sometimes, for large dirs, htar takes a while. You could then consider running it as part of a (single-node/single-cpu) job script.

### Desktops/Laptops:

Even for small simulation runs, it is worth to create data archives. A good location for such an archive might be the cloud storage provided by one’s institution.

Tools like rclone can help with this, e.g., to quickly sync a large amount of directories to a Google Drive.

<table>
<thead>
<tr>
<th>Asynchronous</th>
<th>File</th>
<th>Copies:</th>
<th>Globus</th>
</tr>
</thead>
<tbody>
<tr>
<td>The scientific data service Globus allows to perform large-scale data copies, between HPC centers as well as local computers, with ease and a graphical user interface. Copies can be kicked off asynchronously, often use dedicated internet backbones and are checked when transfers are complete.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many HPC centers also add their archives as a storage endpoint and one can download a client program to add also one’s desktop/laptop.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Scientific Data for Publications

It is good practice to make computational results accessible, scrutinizable and ideally even reusable.

For data artifacts up to approximately 50 GB, consider using free services like Zenodo and Figshare to store supplementary materials of your publications.

For more information, see the open science movement, open data and open access.

**Note:** More information, guidance and templates will be posted here in the future.

### 3.7 FAQ

This section lists frequently asked usage questions.

#### 3.7.1 What is MPI thread support level?

We report this in output on startup together with other information.

That is the MPI support for threaded execution, e.g., with OpenMP or system threads.

We currently only use this for optional, async IO with AMReX plotfiles. In the past, requesting MPI threading support had performance penalties, but we have not seen such anymore on recent systems. Thus, we request it by default but you can overwrite it with a compile time option if it ever becomes needed.
3.7.2 How do I suppress tiny profiler output if I do not care to see it?

Via `AMReX_TINY_PROFILE=OFF` (see: build options and then AMReX build options). We change the default in `cmake/dependencies/AMReX.cmake`.

Note that the tiny profiler adds literally no overhead to the simulation runtime, thus we enable it by default.

3.7.3 What kinds of RZ output do you support?

In RZ, supported detail of RZ output depends on the output format that is configured in the inputs file.

openPMD supports output of the detailed RZ modes and reconstructs representations on-the-fly in post-processing, e.g., in openPMD-viewer or other tools. For some tools, this is in-development.

AMReX plotfiles and other in situ methods output a 2D reconstructed Cartesian slice at $\theta = 0$ by default (and can opt-in to dump raw modes).
CHAPTER
FOUR

DATA ANALYSIS

4.1 Output formats

WarpX can write diagnostics data either in

- plotfile format or in
- openPMD format(s).

Plotfiles are AMReX’ native data format, while openPMD is implemented in popular community formats such as ADIOS and HDF5.

This section describes some of the tools available to visualize the data.

4.1.1 Asynchronous IO

When using the AMReX plotfile format, users can set the amrex.async_out=1 option to perform the IO in a non-blocking fashion, meaning that the simulation will continue to run while an IO thread controls writing the data to disk. This can significantly reduce the overall time spent in IO. This is primarily intended for large runs on supercomputers such as Summit and Cori; depending on the MPI implementation you are using, you may not see a benefit on your workstation.

When writing plotfiles, each rank will write to a separate file, up to some maximum number (by default, 64). This maximum can be adjusted using the amrex.async_out_nfiles inputs parameter. To use asynchronous IO with more than amrex.async_out_nfiles MPI ranks, WarpX must be configured with -DWarpX_MPI_THREAD_MULTIPLE=ON. Please see the building instructions for details.

4.1.2 In Situ Capabilities

WarpX includes so-called reduced diagnostics. Reduced diagnostics create observables on-the-fly, such as energy histograms or particle beam statistics and are easily visualized in post-processing.

In addition, WarpX also has vn-situ visualization capabilities (i.e. visualizing the data directly from the simulation, without dumping data files to disk).
SENSEI is a lightweight framework for in situ data analysis. SENSEI’s data model and API provide uniform access to and run time selection of a diverse set of visualization and analysis back ends including VisIt Libsim, ParaView Catalyst, VTK-m, Ascent, ADIOS, Yt, and Python.

SENSEI uses an XML file to select and configure one or more back ends at run time. Run time selection of the back end via XML means one user can access Catalyst, another Libsim, yet another Python with no changes to the code.

The three major architectural components in SENSEI are data adaptors which present simulation data in SENSEI’s data model, analysis adaptors which present the back end data consumers to the simulation, and bridge code from which the simulation manages adaptors and periodically pushes data through the system. SENSEI comes equipped with a number of analysis adaptors enabling use of popular analysis and visualization libraries such as VisIt Libsim, ParaView Catalyst, Python, and ADIOS to name a few. AMReX contains SENSEI data adaptors and bridge code making it easy to use in AMReX based simulation codes.

SENSEI provides a configurable analysis adaptor which uses an XML file to select and configure one or more back ends at run time. Run time selection of the back end via XML means one user can access Catalyst, another Libsim, yet another Python with no changes to the code. This is depicted in figure Fig. 4.1. On the left side of the figure AMReX produces data, the bridge code pushes the data through the configurable analysis adaptor to the back end that was selected at run time.
Compiling with GNU Make

For codes making use of AMReX’s build system add the following variable to the code’s main GNUmakefile.

```
USE_SENSEI_INSITU = TRUE
```

When set, AMReX’s make files will query environment variables for the lists of compiler and linker flags, include directories, and link libraries. These lists can be quite elaborate when using more sophisticated back ends, and are best set automatically using the `sensei_config` command line tool that should be installed with SENSEI. Prior to invoking make use the following command to set these variables:

```
source sensei_config
```

Typically, the `sensei_config` tool is in the users PATH after loading the desired SENSEI module. After configuring the build environment with `sensei_config`, proceed as usual.

```
make -j4 -f GNUmakefile
```

### ParmParse Configuration

Once an AMReX code has been compiled with SENSEI features enabled, it will need to be enabled and configured at runtime. This is done using ParmParse input file. The supported parameters are described in the following table.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>insitu.int</td>
<td>turns in situ processing on or off and controls how often data is processed.</td>
<td>0</td>
</tr>
<tr>
<td>insitu.start</td>
<td>controls when in situ processing starts.</td>
<td>0</td>
</tr>
<tr>
<td>insitu.config</td>
<td>points to the SENSEI XML file which selects and configures the desired back end.</td>
<td></td>
</tr>
<tr>
<td>insitu.</td>
<td>when the lower left corner of the mesh is pinned to 0.,0.,0.</td>
<td>0</td>
</tr>
<tr>
<td>pin_mesh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A typical use case is to enabled SENSEI by setting `insitu.int` to be greater than 1, and `insitu.config` to point SENSEI to an XML file that selects and configures the desired back end.

```
insitu.int = 2
insitu.config = render_iso_catalyst.xml
```

### Back-end Selection and Configuration

The back end is selected and configured at run time using the SENSEI XML file. The XML sets parameters specific to SENSEI and to the chosen back end. Many of the back ends have sophisticated configuration mechanisms which SENSEI makes use of. For example the following XML configuration was used on NERSC’s Cori with WarpX to render 10 iso surfaces, shown in figure Fig. 4.2, using VisIt Libsim.

```
<sensei>
  <analysis type="libsim" frequency="1" mode="batch"
    session="beam_j_pin.session"
    image-filename="beam_j_pin_%ts" image-width="1200" image-height="900"
    image-format="png" enabled="1"/>
</sensei>
```
The *session* attribute names a session file that contains VisIt specific runtime configuration. The session file is generated using VisIt GUI on a representative dataset. Usually this data set is generated in a low resolution run of the desired simulation.

![Image: Rendering of 10 3D iso-surfaces of j using VisIt libsim. The upper left quadrant has been clipped away to reveal inner structure.](image)

**Fig. 4.2:** Rendering of 10 3D iso-surfaces of j using VisIt libsim. The upper left quadrant has been clipped away to reveal inner structure.

The same run and visualization was repeated using ParaView Catalyst, shown in figure **Fig. 4.3**, by providing the following XML configuration.

```xml
<sensei>
  <analysis type="catalyst" pipeline="pythonscript"
    filename="beam_j.py" enabled="1" />
</sensei>
```

Here the *filename* attribute is used to pass Catalyst a Catalyst specific configuration that was generated using the ParaView GUI on a representative dataset.

The renderings in these runs were configured using a representative dataset which was obtained by running the simulation for a few time steps at a lower spatial resolution. When using VisIt Libsim the following XML configures the VTK writer to write the simulation data in VTK format. At the end of the run a `.visit` file that VisIt can open will be generated.
Fig. 4.3: Rendering of 10 3D iso-surfaces of $j$ using ParaView Catalyst. The upper left quadrant has been clipped away to reveal inner structure.
When using ParaView Catalyst the following XML configures the VTK writer to write the simulation data in VTK format. At the end of the run a .pvd file that ParaView can open will be generated.

```xml
<sensei>
  <analysis type="PosthocIO" mode="visit" writer="xml"
    ghost_array_name="avtGhostZones" output_dir="./"
    enabled="1">
  </analysis>
</sensei>
```

Obtaining SENSEI

SENSEI is hosted on Kitware’s Gitlab site at https://gitlab.kitware.com/sensei/sensei It’s best to checkout the latest release rather than working on the develop branch.

To ease the burden of wrangling backend installs SENSEI provides two platforms with all dependencies pre-installed, a VirtualBox VM, and a NERSC Cori deployment. New users are encouraged to experiment with one of these.

SENSEI VM

The SENSEI VM comes with all of SENSEI’s dependencies and the major backends such as VisIt and ParaView installed. The VM is the easiest way to test things out. It also can be used to see how installs were done and the environment configured.

The SENSEI VM can be downloaded here.

The SENSEI VM uses modules to manage the build and run environment. Load the SENSEI modulefile for the back-end you wish to use. The following table describes the available installs and which back-ends are supported in each.

<table>
<thead>
<tr>
<th>modulefile</th>
<th>back-end(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensei/2.1.1-catalyst-shared</td>
<td>ParaView Catalyst, ADIOS, Python</td>
</tr>
<tr>
<td>sensei/2.1.1-lbsim-shared</td>
<td>VisIt Lbsim, ADIOS, Python</td>
</tr>
<tr>
<td>sensei/2.1.1-vtk-shared</td>
<td>VTK-m, ADIOS, Python</td>
</tr>
</tbody>
</table>
SENSEI installs use modules to manage the build and run environment. Load the SENSEI modulefile for the back-end you wish to use. The following table describes the available installs and which back-ends are supported in each.

<table>
<thead>
<tr>
<th>modulefile</th>
<th>back-end(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensei/2.1.0-catalyst-shared</td>
<td>ParaView Catalyst, ADIOS, Python</td>
</tr>
<tr>
<td>sensei/2.1.0-libsim-shared</td>
<td>VisIt Libsim, ADIOS, Python</td>
</tr>
<tr>
<td>sensei/2.1.0-vtk-shared</td>
<td>VTK-m, ADIOS, Python</td>
</tr>
</tbody>
</table>

To access the SENSEI modulefiles on cori first add the SENSEI install to the search path:

```
module use /usr/common/software/sensei/modulefiles
```

Rendering with VisIt Libsim

This section shows an example of using SENSEI and three different backends on a 3D LPA simulation. The instructions are specifically for NERSC cori, but also work with the SENSEI VM. The primary difference between working through the examples on cori or the VM are that different versions of software are installed.

First, log into cori and clone the git repo's.

```
cd $SCRATCH
mkdir warpx
cd warpx/
git clone https://github.com/ECP-WarpX/WarpX.git WarpX-libsim
git clone https://github.com/AMReX-Codes/amrex
git clone https://github.com/ECP-WarpX/picsar.git
cd WarpX-libsim
vim GNUmakefile
```

Next, edit the makefile to turn the SENSEI features on.

```
USE_SENSEI_INSITU_TRUE
```

Then, load the SENSEI VisIt module, bring SENSEI’s build requirements into the environment, and compile WarpX.

```
module use /usr/common/software/sensei/modulefiles/
module load sensei/2.1.0-libsim-shared
source sensei_config
make -j8
```

Download the WarpX input deck, SENSEI XML configuration and VisIt session files. The inputs file configures WarpX, the xml file configures SENSEI, and the session file configures VisIt. The inputs and xml files are written by hand, while the session file is generated in VisIt gui on a representative data set.

4.1. Output formats
wget https://data.kitware.com/api/v1/item/5c05d48e8d777f2179d22f20/download -O inputs.3d
wget https://data.kitware.com/api/v1/item/5c05d4588d777f2179d22f16/download -O beam_j_pin.xml
wget https://data.kitware.com/api/v1/item/5c05d4588d777f2179d22f0e/download -O beam_j_pin.session

To run the demo, submit an interactive job to the batch queue, and launch WarpX.

salloc -C haswell -N 1 -t 00:30:00 -q debug
./Bin/main3d.gnu.TPROF.MPI.OMP.ex inputs.3d

Rendering with ParaView Catalyst

First, log into cori and clone the git repo's.

cd $SCRATCH
mkdir warpx

cd warpx/

git clone https://github.com/ECP-WarpX/WarpX.git WarpX-catalyst

git clone --branch development https://github.com/AMReX-Codes/amrex

git clone https://github.com/ECP-WarpX/picsar.git

cd WarpX-catalyst
vim GNUmakefile

Next, edit the makefile to turn the SENSEI features on.

USE_SENSEI_INSITU_TRUE

Then, load the SENSEI ParaView module, bring SENSEI's build requirements into the environment, and compile WarpX.

module use /usr/common/software/sensei/modulefiles/
module load sensei/2.1.0-catalyst-shared

source sensei_config
make -j8

Download the WarpX input deck, SENSEI XML configuration and and ParaView session files. The inputs file configures WarpX, the xml file configures SENSEI, and the session file configures ParaView. The inputs and xml files are written by hand, while the session file is generated in ParaView gui on a representative data set.

wget https://data.kitware.com/api/v1/item/5c05b3fd8d777f2179d2067d/download -O inputs.3d
wget https://data.kitware.com/api/v1/item/5c05b3fd8d777f2179d20675/download -O beam_j.xml
wget https://data.kitware.com/api/v1/item/5c05b3fd8d777f2179d2066d/download -O beam_j.py

To run the demo, submit an interactive job to the batch queue, and launch WarpX.

salloc -C haswell -N 1 -t 00:30:00 -q debug
./Bin/main3d.gnu.TPROF.MPI.OMP.ex inputs.3d
In situ Calculation with Python

SENSEI’s Python back-end loads a user provided script file containing callbacks for Initialize, Execute, and Finalize phases of the run. During the execute phase the simulation pushes data through SENSEI. SENSEI forwards this data to the user provided Python function. SENSEI’s MPI communicator is made available to the user’s function via a global variable comm.

Here is a template for the user provided Python code.

```python
# YOUR IMPORTS HERE

# SET DEFAULTS OF GLOBAL VARIABLES THAT INFLUENCE RUNTIME BEHAVIOR HERE

def Initialize():
    """ Initialization code """
    # YOUR CODE HERE
    return

def Execute(dataAdaptor):
    """ Use sensei::DataAdaptor instance passed in dataAdaptor to access and process simulation data """
    # YOUR CODE HERE
    return

def Finalize():
    """ Finalization code """
    # YOUR CODE HERE
    return

Initialize and Finalize are optional and will be called if they are provided. Execute is required. SENSEI’s DataAdaptor API is used to obtain data and metadata from the simulation. Data is through VTK Object’s. In WarpX the vtkOverlappingAMR VTK dataset is used.

The following script shows a simple integration of a scalar quantity over the valid cells of the mesh. The result is saved in a CSV format.

```python
import numpy as np, matplotlib.pyplot as plt from vtk.util.numpy_support import * from vtk import vtkDataObject import sys

# default values of control parameters
array = ''
out_file = ''

def Initialize():
    # rank zero writes the result
    if comm.Get_rank() == 0:
        fn = out_file if out_file else 'integrate_%s.csv' % (array)
        f = open(fn, 'w')
        f.write('# time, %s
' % (array))
        f.close()
    return
```

(continues on next page)
def Execute(adaptor):
    # get the mesh and arrays we need
    dobj = adaptor.GetMesh('mesh', False)
    adaptor.AddArray(dobj, 'mesh', vtkDataObject.CELL, array)
    adaptor.AddGhostCellsArray(dobj, 'mesh')
    time = adaptor.GetDataTime()

    # integrate over the local blocks
    varint = 0.
    it = dobj.NewIterator()
    while not it.IsDoneWithTraversal():
        # get the local data block and its props
        blk = it.GetCurrentDataObject()

        # get the array container
        atts = blk.GetCellData()

        # get the data array
        var = vtk_to_numpy(atts.GetArray(array))

        # get ghost cell mask
        ghost = vtk_to_numpy(atts.GetArray('vtkGhostType'))
        ii = np.where(ghost == 0)[0]

        # integrate over valid cells
        varint = np.sum(var[ii])*np.prod(blk.GetSpacing())
        it.GoToNextItem()

    # reduce integral to rank 0
    varint = comm.reduce(varint, root=0, op=MPI.SUM)

    # rank zero writes the result
    if comm.Get_rank() == 0:
        fn = out_file if out_file else 'integrate_%s.csv' % (array)
        f = open(fn, 'a+')
        f.write('%s, %s
' % (time, varint))
        f.close()
    return

The following XML configures SENSEI’s Python back-end.

```
<analysis type="python" script_file="/integrate.py" enabled="1">
    <initialize_source>
        array="rho"
        out_file="rho.csv"
    </initialize_source>
</analysis>
```

The `script_file` attribute sets the file path to load the user’s Python code from, and the `initialize_source` element contains Python code that controls runtime behavior specific to each user provided script.
In situ Visualization with Ascent

Ascent is a system designed to meet the in-situ visualization and analysis needs of simulation code teams running multi-physics calculations on many-core HPC architectures. It provides rendering runtimes that can leverage many-core CPUs and GPUs to render images of simulation meshes.

Compiling with GNU Make

After building and installing Ascent according to the instructions at Building Ascent, you can enable support for it in WarpX by changing the line

```
USE_ASCENT_INSITU=FALSE
```

in GNU makefile to

```
USE_ASCENT_INSITU=TRUE
```

Furthermore, you must ensure that either the ASCENT_DIR shell environment variable contains the directory where Ascent is installed or you must specify this location when invoking make, i.e.,

```
maker -j 8 USE_ASCENT_INSITU=TRUE ASCENT_DIR=/path/to/ascent/install
```

Inputs File Configuration

Once WarpX has been compiled with Ascent support, it will need to be enabled and configured at runtime. This is done using our usual inputs file (read with amrex::ParmParse). The supported parameters are part of the FullDiagnostics with <diag_name>.format parameter set to ascent.

Visualization/Analysis Pipeline Configuration

Ascent uses the file ascent_actions.yaml to configure analysis and visualization pipelines. Ascent looks for the ascent_actions.yaml file in the current working directory.

For example, the following ascent_actions.yaml file extracts an isosurface of the field Ex for 15 levels and saves the resulting images to levels_<nnnn>.png. Ascent Actions provides an overview over all available analysis and visualization actions.

```
- action: "add_pipelines"
  pipelines:
    p1:
      f1:
        type: "contour"
        params:
          field: "Ex"
          levels: 15

  - action: "add_scenes"
    scenes:
      scene1:
        image_prefix: "levels_%04d"
```

(continues on next page)
Here is another `ascent_actions.yaml` example that renders isosurfaces and particles:

```yaml
  - action: "add_pipelines"
    pipelines:
      p1:
        f1:
          type: "contour"
          params:
            field: "Bx"
            levels: 3
  - action: "add_scenes"
    scenes:
      scene1:
        plots:
          plot1:
            type: "pseudocolor"
            pipeline: "p1"
            field: "Ex"
          plot2:
            type: "pseudocolor"
            field: "particle_electrons_Bx"
            points:
              radius: 0.0000005
        renders:
          r1:
            camera:
              azimuth: 100
              elevation: 10
            image_prefix: "out_render_3d_%06d"
```

Finally, here is a more complex `ascent_actions.yaml` example that creates the same images as the prior example, but adds a trigger that creates a Cinema Database at cycle 300:

```yaml
  - action: "add_triggers"
    triggers:
      t1:
        params:
          condition: "cycle() == 300"
          actions_file: "trigger.yaml"
  - action: "add_pipelines"
    pipelines:
      p1:
```

(continues on next page)
When the trigger condition is meet, `cycle() == 300`, the actions in `trigger.yaml` are also executed:

```yaml
- action: "add_pipelines"
  pipelines:
    p1:
      f1:
        type: "contour"
        params:
          field: "jy"
          iso_values: [1000000000000.0, -1000000000000.0]
  action: "add_scenes"
  scenes:
    scene1:
      plots:
        plot1:
          type: "pseudocolor"
          pipeline: "p1"
          field: "jy"
        plot2:
          type: "pseudocolor"
          field: "particle_electrons_w"
          points:
            radius: 0.0000002
      renders:
        r1:
          camera:
            azimuth: 100
            elevation: 10
          image_prefix: "out_render_jy_part_w_3d_%06d"
```

(continues on next page)
You can view the Cinema Database result by opening `cinema_databases/cinema_out/index.html`.

### Replay

With Ascent/Conduit, one can store the intermediate data files before the rendering step is applied to custom files. These so-called Conduit Blueprint HDF5 files can be “replayed”, i.e. rendered without running the simulation again. VisIt 3.0+ also supports those files.

**Replay** is a utility that allows the user to replay a simulation from aforementioned files and rendering them with Ascent. Replay enables the user or developer to pick specific time steps and load them for Ascent visualization, without running the simulation again.

We will guide you through the replay procedure.

#### Get Blueprint Files

To use replay, you first need Conduit Blueprint HDF5 files. The following block can be used in an ascent action to extract Conduit Blueprint HDF5 files from a simulation run.

```plaintext
 action: "add_extracts"
 extracts:
   el:
     type: "relay"
     params:
       path: "conduit_blueprint"
       protocol: "blueprint/mesh/hdf5"
```

The output in the WarpX run directory will look as in the following listing. The `.root` file is a metadata file and the corresponding directory contains the conduit blueprint data in an internal format that is based on HDF5.

```plaintext
conduit_blueprint.cycle_000000/
conduit_blueprint.cycle_000000.root
conduit_blueprint.cycle_000050/
conduit_blueprint.cycle_000050.root
conduit_blueprint.cycle_000100/
conduit_blueprint.cycle_000100.root
```

In order to select a few time steps after the fact, a so-called cycles file can be created. A cycles file is a simple text file that lists one root file per line, e.g.:

```plaintext
conduit_blueprint.cycle_000100.root
conduit_blueprint.cycle_000050.root
```
Run

For Ascent Replay, two command line tools are provided in the utilities/replay directory of the Ascent installation. There are two versions of replay: the MPI-parallel version `replay_mpi` and a serial version, `replay_ser`. Use an MPI-parallel replay with data sets created with MPI-parallel builds of WarpX. Here we use `replay_mpi` as an example.

The options for replay are:

- `--root`: specifies Blueprint root file to load
- `--cycles`: specifies a text file containing a list of Blueprint root files to load
- `--actions`: specifies the name of the actions file to use (default: `ascent_actions.yaml`)

Instead of starting a simulation that generates data for Ascent, we now execute `replay_ser`/`replay_mpi`. Replay will loop over the files listed in `cycles` in the order in which they appear in the `cycles` file.

For example, for a small data example that fits on a single computer:

```
./replay_ser --root=conduit_blueprint.cycle_000400.root --actions=ascent_actions.yaml
```

Will replay the data of WarpX step 400 ("cycle" 400). A whole set of steps can be replayed with the above mentioned `cycles` file:

```
./replay_ser --cycles=warpx_list.txt --actions=ascent_actions.yaml
```

For larger examples, e.g. on a cluster with Slurm batch system, a parallel launch could look like this:

```
# one step
srun -n 8 ./replay_mpi --root=conduit_blueprint.cycle_000400.root --actions=ascent_actions.yaml

# multiple steps
srun -n 8 ./replay_mpi --cycles=warpx_list.txt --actions=ascent_actions.yaml
```

Example

Actions

A visualization of the electric field component $E_x$ (variable: Ex) with a contour plot and with added particles can be obtained with the following Ascent Action. This action can be used both in replay as well as in situ runs.

```
action: "add_pipelines"

pipelines:
  clipped_volume:
    f0:
      type: "contour"
      params:
        field: "Ex"
        levels: 16
    f1:
      type: "clip"
      params:
        topology: topo # name of the amr mesh
        multi_plane: point1:
```

(continues on next page)
x: 0.0
y: 0.0
z: 0.0

normal1:
x: 0.0
y: -1.0
z: 0.0

point2:
x: 0.0
y: 0.0
z: 0.0

normal2:
x: -0.7
y: -0.7
z: 0.0

sampled_particles:

f1:
  type: histsampling
  params:
    field: particle_electrons_uz
    bins: 64
    sample_rate: 0.90

f2:
  type: "clip"
  params:
    topology: particle_electrons # particle data

multi_plane:

point1:
x: 0.0
y: 0.0
z: 0.0

normal1:
x: 0.0
y: -1.0
z: 0.0

point2:
x: 0.0
y: 0.0
z: 0.0

normal2:
x: -0.7
y: -0.7
z: 0.0

# Uncomment this block if you want to create "Conduit Blueprint files" that can
# be used with Ascent "replay" after the simulation run.
# Replay is a workflow to visualize individual steps without running the simulation
# again.
#
# action: "add_extracts"
# extracts:
#   e1:
There are more Ascent Actions examples available for you to play.

**Workflow**

In the preparation of simulations, it is generally useful to run small, under-resolved versions of the planned simulation layout first. Ascent replay is helpful in the setup of an in situ visualization pipeline during this process. In the following, a Jupyter-based workflow is shown that can be used to quickly iterate on the design of a `ascent_actions.yaml` file, repeatedly rendering the same (small) data.

First, run a small simulation, e.g. on a local computer, and create conduit blueprint files (see above). Second, copy the Jupyter Notebook file `ascent_replay_warpx.ipynb` into the simulation output directory. Third, download and start a Docker container with a prepared Jupyter installation and Ascent Python bindings from the simulation output directory:

```bash
docker pull alpinedav/ascent-jupyter:latest
docker run -v$PWD:/home/user/ascent/install-debug/examples/ascent/tutorial/ascent_intro/˓
˓→ascent-jupyter:latest
```

Inside the Jupyter Lab is a `replay/` directory, which mounts the outer working directory. You can now open `ascent_replay_warpx.ipynb` and execute all cells. The last two cells are the replay action that can be quickly iterated: change `replay_actions.yaml` cell and execute both.

**Note:**
- Keep an eye on the terminal, if a replay action is erroneous it will show up on the terminal that started the docker container. (TODO: We might want to catch that inside python and print it in Jupyter instead.)
- If you remove a "key" from the replay action, you might see an error in the AscentViewer. Restart and execute all cells in that case.

If you like the 3D rendering of laser wakefield acceleration on the WarpX documentation front page (which is also the avatar of the ECP-WarpX organization), you can find the serial analysis script `video_yt.py` as well as a parallel analysis script `video_yt.py` used to make a similar rendering for a beam-driven wakefield simulation, running parallel.

## 4.1.3 Staggering in Data Output

Warning: currently, quantities in the output file for iteration $n$ are not all defined at the same physical time due to the staggering in time in WarpX. The table below provides the physical time at which each quantity in the output file is written, in units of time step, for time step $n$.

<table>
<thead>
<tr>
<th>quantity</th>
<th>staggering</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$n$</td>
</tr>
<tr>
<td>$B$</td>
<td>$n$</td>
</tr>
<tr>
<td>$j$</td>
<td>$n-1/2$</td>
</tr>
<tr>
<td>$\rho$</td>
<td>$n$</td>
</tr>
<tr>
<td>position</td>
<td>$n$</td>
</tr>
<tr>
<td>momentum</td>
<td>$n-1/2$</td>
</tr>
</tbody>
</table>

## 4.2 yt-project

`yt` is a Python package that can help in analyzing and visualizing WarpX data (among other data formats). It is convenient to use `yt` within a Jupyter notebook.

### 4.2.1 Data Support

`yt` primarily supports WarpX through plotfiles. There is also support for openPMD HDF5 files in `yt` (w/o mesh refinement).
4.2.2 Installation

From the terminal, install the latest version of yt:

```
python3 -m pip install cython
python3 -m pip install --upgrade yt
```

Alternatively, yt can be installed via their installation script, see yt installation web page.

4.2.3 Visualizing the data

Once data ("plotfiles") has been created by the simulation, open a Jupyter notebook from the terminal:

```
jupyter notebook
```

Then use the following commands in the first cell of the notebook to import yt and load the first plot file:

```
import yt
ds = yt.load('./diags/plotfiles/plt00000/')
```

The list of field data and particle data stored can be seen with:

```
ds.field_list
```

For a quick start-up, the most useful commands for post-processing can be found in our Jupyter notebook Visualization.ipynb

Field data

Field data can be visualized using `yt.SlicePlot` (see the docstring of this function here)

For instance, in order to plot the field $E_x$ in a slice orthogonal to $y$ (1):

```
yt.SlicePlot(ds, 1, 'Ex', origin='native')
```

**Note:** `yt.SlicePlot` creates a 2D plot with the same aspect ratio as the physical size of the simulation box. Sometimes this can lead to very elongated plots that are difficult to read. You can modify the aspect ratio with the `aspect` argument; for instance:

```
yt.SlicePlot(ds, 1, 'Ex', aspect=1./10)
```

Alternatively, the data can be obtained as a numpy array.

For instance, in order to obtain the field $j_z$ (on level 0) as a numpy array:

```
ad0 = ds.covering_grid(level=0, left_edge=ds.domain_left_edge, dims=ds.domain_dimensions)
jz_array = ad0['jz'].to_ndarray()
```
Particle data can be visualized using `yt.ParticlePhasePlot` (see the docstring here).

For instance, in order to plot the particles’ x and y positions:

```python
yt.ParticlePhasePlot( ds.all_data(), 'particle_position_x', 'particle_position_y', 'particle_weight')
```

Alternatively, the data can be obtained as a numpy array.

For instance, in order to obtain the array of position x as a numpy array:

```python
ad = ds.all_data()
x = ad['particle_position_x'].to_ndarray()
```

### 4.2.4 Further information

A lot more information can be obtained from the yt documentation, and the corresponding notebook tutorials here.

### Out-of-the-box plotting script

A ready-to-use python script for plotting simulation results is available at `plot_parallel.py`. Feel free to use it out-of-the-box or to modify it to suit your needs.

### Dependencies

Most of its dependencies are standard Python packages, that come with a default Anaconda installation or can be installed with pip or conda: os, matplotlib, sys, argparse, matplotlib, scipy.

Additional dependencies are `yt` >= 3.5 (or `yt` >= 3.6 if you are using rigid injection, see section `yt-project` on how to install yt), and mpi4py.

### Run serial

Executing the script with

```python
python plot_parallel.py
```

will loop through plotfiles named plt????? (e.g., plt00000, plt00100 etc.) and save one image per plotfile. For a 2D simulation, a 2D colormap of the Ez field is plotted by default, with 1/20 of particles of each species (with different colors). For a 3D simulation, a 2D colormap of the central slices in y is plotted, and particles are handled the same way.

The script reads command-line options (which field and particle species, rendering with yt or matplotlib, etc.). For the full list of options, run

```python
python plot_parallel.py --help
```

In particular, option `--plot_Ey_max_evolution` shows you how to plot the evolution of a scalar quantity over time (by default, the max of the Ey field). Feel free to modify it to plot the evolution of other quantities.
Run parallel

To execute the script in parallel, you can run for instance

```
mpirun -np 4 python plot_parallel.py --parallel
```

In this case, MPI ranks will share the plotfiles to process as evenly as possible. Note that each plotfile is still processed in serial. When option `--plot_Ey_max_evolution` is on, the scalar quantity is gathered to rank 0, and rank 0 plots the image.

If all dependencies are satisfied, the script can be used on Summit or Cori. For instance, the following batch script illustrates how to submit a post-processing batch job on Cori haswell with some options:

```
#!/bin/bash
# Copyright 2019 Maxence Thevenet
# # This file is part of WarpX.
# # License: BSD-3-Clause-LBNL
#SBATCH --job-name=postproc
#SBATCH --time=00:20:00
#SBATCH -C haswell
#SBATCH -N 8
#SBATCH -q regular
#SBATCH -e postproce.txt
#SBATCH -o postproco.txt
#SBATCH --mail-type=end
#SBATCH --account=m2852
export OMP_NUM_THREADS=1
# Requires python3 and yt > 3.5
srun -n 32 -c 16 python plot_parallel.py --path <path/to/plotfiles> --plotlib=yt --parallel
```

Advanced Visualization of Plotfiles With yt (for developers)

This sections contains yt commands for advanced users. The Particle-In-Cell methods uses a staggered grid (see particle-in-cell theory), so that the x, y, and z components of the electric and magnetic fields are all defined at different locations in space. Regular output (see the yt-project page, or the notebook at WarpX/Tools/PostProcessing/Visualization.ipynb for an example) returns cell-centered data for convenience, which involves an additional operation. It is sometimes useful to access the raw data directly. Furthermore, the WarpX implementation for mesh refinement contains a number of grids for each level (coarse, fine and auxiliary, see the theory for more details), and it is sometimes useful to access each of them (regular output return the auxiliary grid only). This page provides information to read raw data of all grids.
Write Raw Data

For a given diagnostic the user has the option to write the raw data by setting `<diag_name>.plot_raw_fields = 1`. Moreover, the user has the option to write also the values of the fields in the guard cells by setting `<diag_name>.plot_raw_fieldsguards = 1`. Please refer to Input Parameters for more information.

Read Raw Data

Meta-data relevant to this topic (for example, number and locations of grids in the simulation) are accessed with

```python
import yt
# get yt dataset
ds = yt.load( './plotfiles/plt00004' )
# Index of data in the plotfile
ds_index = ds.index
# Print the number of grids in the simulation
ds_index.grids.shape
# Left and right physical boundary of each grid
ds_index.grid_left_edge
nds_index.grid_right_edge
# List available fields
ds.field_list
```

When `<diag_name>.plot_raw_fields = 1`, here are some useful commands to access properties of a grid and the Ex field on the fine patch:

```python
# store grid number 2 into my_grid
my_grid = ds_index.grids[2]
# Get left and right edges of my_grid
my_grid.LeftEdge
my_grid.RightEdge
# Get Level of my_grid
my_grid.Level
# left edge of the grid, in number of points
my_grid.start_index
```

Return the Ex field on the fine patch of grid `my_grid`:

```python
my_field = my_grid['raw', 'Ex_fp'].squeeze().v
```

For a 2D plotfile, `my_field` has shape `(nx,nz,2)`. The last component stands for the two values on the edges of each cell for the electric field, due to field staggering. Numpy function `squeeze` removes empty components. While `yt` arrays are unit-aware, it is sometimes useful to extract the data into unitless numpy arrays. This is achieved with `.v`. In the case of `Ex_fp`, the staggering is on direction `x`, so that `my_field[:,::-1,1] == my_field[:,1:,0]`.

All combinations of the fields (`E` or `B`), the component (`x`, `y` or `z`) and the grid (`_fp` for fine, `_cp` for coarse and `_aux` for auxiliary) can be accessed in this way, i.e., `my_grid['raw', 'Ey_aux']` or `my_grid['raw', 'Bz_cp']` are valid queries.
When the output includes the data in the guard cells, the user can read such data using the post-processing tool `read_raw_data.py`, available in `Tools/PostProcessing/`, as illustrated in the following example:

```python
from read_raw_data import read_data

# Load all data saved in a given path
path = './diags/diag00200/

# Load Ex_fp on mesh refinement level 0
level = 0
field = 'Ex_fp'

# data[level] is a dictionary, data[level][field] is a numpy array
my_field = data[level][field]
```

Note that a list of all available raw fields written to output, that is, a list of all valid strings that the variable `field` in the example above can be assigned to, can be obtained by calling `data[level].keys()`.

In order to plot a 2D slice of the data with methods like `matplotlib.axes.Axes.imshow`, one might want to pass the correct `extent` (the bounding box in data coordinates that the image will fill), including the guard cells. One way to set the correct `extent` is illustrated in the following example (case of a 2D slice in the `(x,z)` plane):

```python
import yt
import numpy as np

from read_raw_data import read_data

# Set the number of cells in the valid domain
# by loading the standard output data with yt
ncells = yt.load(path).domain_dimensions

# Set the number of dimensions automatically (2D or 3D)
dim = 2 if (ncells[2] == 1) else 3

xdir = 0
zdir = 1 if (dim == 2) else 2

# Set the extent (bounding box in data coordinates, including guard cells)
left_edge_x = 0 - (my_field.shape[xdir] - ncells[xdir]) // 2
right_edge_x = ncells[xdir] + (my_field.shape[xdir] - ncells[xdir]) // 2
left_edge_z = 0 - (my_field.shape[zdir] - ncells[zdir]) // 2
right_edge_z = ncells[zdir] + (my_field.shape[zdir] - ncells[zdir]) // 2
```
right_edge_z = ncells[zdir] + (my_field.shape[zdir] - ncells[zdir]) // 2
extent = np.array([left_edge_z, right_edge_z, left_edge_x, right_edge_x])

4.3 openPMD-viewer

openPMD-viewer is an open-source 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

4.3.1 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
```

4.3.2 Usage

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

```
%matplotlib notebook
```

and for Jupyter Lab use this instead:

```
%matplotlib widget
```

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('./diags/diag1/
```

**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:
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 here for more info.

### 4.4 openPMD-api

openPMD-api is an open-source C++ and Python API for openPMD data.

**Please see the openPMD-api manual for a quick introduction:** [https://openpmd-api.readthedocs.io](https://openpmd-api.readthedocs.io)

### 4.5 3D Visualization: ParaView

**TODO**

WarpX results can be visualized by ParaView, an open source visualization and analysis software. ParaView can be downloaded and installed from [https://www.paraview.org](https://www.paraview.org).

#### 4.5.1 Tutorials

- [https://www.paraview.org/Wiki/The_ParaView_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+introduction)
- [https://www.youtube.com/results?search_query=paraview+tutorial](https://www.youtube.com/results?search_query=paraview+tutorial)

#### 4.5.2 openPMD

WarpX’ openPMD files can be visualized with ParaView 5.9+. ParaView supports ADIOS1, ADIOS2 and HDF5 files, as it implements (like WarpX) against openPMD-api.

For openPMD output, WarpX automatically creates an `.pmd` file per diagnostics, which can be opened with ParaView.

#### 4.5.3 Plotfiles

**TODO** (cross-ref AMReX capabilities)

### 4.6 3D Visualization: VisIt

WarpX results can be visualized by VisIt, an open source visualization and analysis software. VisIt can be downloaded and installed from [https://wci.llnl.gov/simulation/computer-codes/visit](https://wci.llnl.gov/simulation/computer-codes/visit).
4.6.1 openPMD

WarpX' openPMD files 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.

4.6.2 Plotfiles

Assuming that you ran a 2D simulation, here are instructions for making a simple plot from a given plotfile:

- Open the header file: Run VisIt, then select “File” -> “Open file …”, then select the Header file associated with the plotfile of interest (e.g., plt10000/Header).
- View the data: Select “Add” -> “Pseudocolor” -> “Ez” and select “Draw”. You can select other variable to draw, such as jx, jy, jz, Ex, ...
- View the grid structure: Select “Subset” -> “levels”. Then double click the text “Subset-levels”, enable the “Wireframe” option, select “Apply”, select “Dismiss”, and then select “Draw”.
- Save the image: Select “File” -> “Set save options”, then customize the image format to your liking, then click “Save”.

Your image should look similar to the one below

In 3D, you must apply the “Operators” -> “Slicing” -> “ThreeSlice”. You can left-click and drag over the image to rotate the image to generate image you like.
To make a movie, you must first create a text file named `movie.visit` with a list of the Header files for the individual frames.

The next step is to run VisIt, select “File” -> “Open file ...”, then select `movie.visit`. Create an image to your liking and press the “play” button on the VCR-like control panel to preview all the frames. To save the movie, choose “File” -> “Save movie ...”, and follow the instructions on the screen.

### 4.7 VisualPIC

VisualPIC is an open-source Python GUI for visual data analysis, especially for advanced accelerator simulations. It supports WarpX' data through openPMD files.

**Please see the its examples:** [https://github.com/AngelFP/VisualPIC](https://github.com/AngelFP/VisualPIC)

### 4.8 PICViewer

PICViewer is a visualization GUI implemented on PyQt. The toolkit provides various easy-to-use functions for data analysis of Warp/WarpX simulations.

It works for both plotfiles and openPMD files.
4.8.1 Main features

- 2D/3D openPMD or WarpX data visualization,
- Multi-plot panels (up to 6 rows x 5 columns) which can be controlled independently or synchronously
- Interactive mouse functions (panel selection, image zoom-in, local data selection, etc)
- Animation from a single or multiple panel(s)
- Saving your job configuration and loading it later
- Interface to use VisIt, yt, or mayavi for 3D volume rendering (currently updating)

4.8.2 Required software

- python 2.7 or higher: http://docs.continuum.io/anaconda/install.
- PyQt5
  
  conda install pyqt

- h5py
- matplotlib
- numpy
- yt

  python3 -m pip install git+https://github.com/yt-project/yt.git --user

- numba

4.8.3 Installation

  python3 -m pip install picviewer

You need to install yt and PySide separately.
You can install from the source for the latest update,

  python3 -m pip install git+https://bitbucket.org/ecp_warpx/picviewer/

4.8.4 To install manually

- Clone this repository

  git clone https://bitbucket.org/ecp_warpx/picviewer/

- Switch to the cloned directory with cd picviewer and type python setup.py install
4.8.5 To run

- You can start PICViewer from any directory. Type `picviewer` in the command line. Select a folder where your data files are located.
- You can directly open your data. Move on to a folder where your data files are located (`cd [your data folder]`) and type `picviewer` in the command line.

**Note:** We currently seek a new maintainer for PICViewer. Please contact us if you are interested.

4.9 Back-Transformed Diagnostics

When running a simulation in a boosted frame, WarpX has the capability to back-transform the simulation results to the laboratory frame of reference, which is often useful to study the physics. A set of functions can be found in the python file `read_raw_data.py`. The main commands can be found in our example jupyter notebook for postprocessing `Visualization.ipynb`.

The full back-transformed diagnostics of the entire domain is written in `lab_frame_data/snapshots/` and the back-transformed diagnostics of the reduced domain is written to `lab_frame_data/slices/`. For instance: To plot the $E_z$ field along the $z$-direction at the center of the 3D-domain of the full back-transformed diagnostics for the entire 3D domain:

```python
import read_raw_data
import matplotlib.pyplot as plt

iteration = 0
field = 'Ez'
snapshot = './lab_frame_data/snapshots/' + 'snapshot' + str(iteration).zfill(5)
header = './lab_frame_data/snapshots/Header'
allrd, info = read_raw_data.read_lab_snapshot(snapshot, header) # Read field data
F = allrd[field]
plt.plot(F[F.shape[0]//2,F.shape[1]//2-1,:])
```

Similarly, the back-transformed diagnostics on a reduced domain (1D line, 2D slice, 3D reduced diagnostic) can also be visualized using `read_raw_data.py`. For instance – let us say that the user-input is an “x-z” slice (at the center of the domain in the “y-direction”), then, to plot $E_z$ on this x-z slice:

```python
iteration = 0
field = 'Ez'
snapshot = './lab_frame_data/slices/' + 'slice' + str(iteration).zfill(5)
header = './lab_frame_data/slices/Header'
allrd, info = read_raw_data.read_lab_snapshot(snapshot, header) # Read field data
F_RD = allrd[field]
plt.plot(F_RD[F_RD.shape[0]//2,0,:])
```

Note that, in the above snippet, we compare the 0th cell of the reduced diagnostic with $F.shape[1]//2-1$. For an x-z slice at $y=y$-mid of the domain, two cells are extracted at the center to ensure that the data format is HDF5 compliant. Let us consider that the domain consists of four cells in the y-dimension: [0,1,2,3]. Then the 2D slice would contain the data that corresponds to [1,2]. That is the 0th cell of the reduced diagnostic corresponds to $ny/2-1$, (where, $ny$ is the number of cells in the y-dimension).

If the back-transformed diagnostics are written in the HDF5 format (This can be done by compiling WarpX with `USE_HDF5=True`), then the full domain snapshot and reduced domain diagnostics can be visualized using h5py:
import matplotlib.pyplot as plt
import h5py

f1 = h5py.File('lab_frame_data/snapshots/snapshot00000', 'r')
nx1 = f1['Ez'].shape[0]
ny1 = f1['Ez'].shape[1]
nz1 = f1['Ez'].shape[2]
plt.plot(f1['Ez'][nx1//2, ny1//2-1, :])

f2 = h5py.File('lab_frame_data/slices/slice00000', 'r')
nx2 = f2['Ez'].shape[0]
ny2 = f2['Ez'].shape[1]
nz2 = f2['Ez'].shape[2]
plt.figure()
plt.plot(f2['Ez'][nx2//2, 0, :])

The back-transformed particle data on the full and reduced diagnostic can be visualized as follows

```python
species = 'ions'
iteration = 1

snapshot = './lab_frame_data/snapshots/' + 'snapshot' + str(iteration).zfill(5)
xbo = get_particle_field(snapshot, species, 'x') # Read particle data
ybo = get_particle_field(snapshot, species, 'y')
zbo = get_particle_field(snapshot, species, 'z')

snapshot = './lab_frame_data/slices/' + 'slice' + str(iteration).zfill(5)
xbo_slice = get_particle_field(snapshot, species, 'x') # Read particle data
ybo_slice = get_particle_field(snapshot, species, 'y')
zbo_slice = get_particle_field(snapshot, species, 'z')
plt.figure()
plt.plot(xbo, ybo, 'r.', markersize=1.)
plt.plot(xbo_slice, ybo_slice, 'bx', markersize=1.)
```

## 4.10 Reduced diagnostics

WarpX has optional reduced diagnostics, that typically return one value (e.g., particle energy) per timestep.

A simple and quick way to read the data using python is

```python
data = numpy.genfromtxt("filename.txt")
```

where data is a two dimensional array, data[i][j] gives the data in the ith row and the jth column.

A Python function to read the data is available from module read_raw_data in WarpX/Tools/PostProcessing/:

```python
from read_raw_data import read_reduced_diags
filename = 'EF.txt'
metadata, data = read_reduced_diags( filename )
# list available diagnostics
data.keys()
# Print total field energy on level 0
```

(continues on next page)
In addition, for reduced diagnostic type `ParticleHistogram`, another Python function is available:

```python
from read_raw_data import read_reduced_diags_histogram
filename = 'velocity_distribution.txt'
metadata_dict, data_dict, bin_value, bin_data = read_reduced_diags_histogram(filename)
# 1-D array of the ith bin value
bin_value[i]
# 2-D array of the jth bin data at the ith time
bin_data[i][j]
```
5.1 Introduction

Fig. 5.1: Plasma laser-driven (top) and charged-particles-driven (bottom) acceleration (rendering from 3-D Particle-In-Cell simulations). A laser beam (red and blue disks in top picture) or a charged particle beam (red dots in bottom picture) propagating (from left to right) through an under-dense plasma (not represented) displaces electrons, creating a plasma wakefield that supports very high electric fields (pale blue and yellow). These electric fields, which can be orders of magnitude larger than with conventional techniques, can be used to accelerate a short charged particle beam (white) to high-energy over a very short distance.

Computer simulations have had a profound impact on the design and understanding of past and present plasma acceleration experiments (Tsung et al. 2006; Geddes et al. 2008; C. Geddes et al. 2009; Huang et al. 2009), with accurate modeling of wake formation, electron self-trapping and acceleration requiring fully kinetic methods (usually Particle-In-Cell) using large computational resources due to the wide range of space and time scales involved.

Numerical modeling complements and guides the design and analysis of advanced accelerators, and can reduce development costs significantly. Despite the major recent experimental successes (Leemans et al. 2014; Blumenfeld et al. 2007; Bulanov S V and Wilkens J J and Esirkepov T Zh and Korn G and Kraft G and Kraft S D and Molls M and Khoroshkov V S 2014; Steinke et al. 2016), the various advanced acceleration concepts need significant progress to
fulfill their potential. To this end, large-scale simulations will continue to be a key component toward reaching a detailed understanding of the complex interrelated physics phenomena at play.

For such simulations, the most popular algorithm is the Particle-In-Cell (or PIC) technique, which represents electromagnetic fields on a grid and particles by a sample of macroparticles. However, these simulations are extremely computationally intensive, due to the need to resolve the evolution of a driver (laser or particle beam) and an accelerated beam into a structure that is orders of magnitude longer and wider than the accelerated beam. Various techniques or reduced models have been developed to allow multidimensional simulations at manageable computational costs: quasistatic approximation (Sprangle, Esarey, and Ting 1990; Antonsen and Mora 1992; Krall et al. 1993; Mora and Antonsen 1997; Huang et al. 2006), ponderomotive guiding center (PGC) models (Antonsen and Mora 1992; Krall et al. 1993; Huang et al. 2006; Benedetti et al. 2010; Cowan et al. 2011), simulation in an optimal Lorentz boosted frame (Vay 2007; Bruhwiler et al. 2009; Vay et al. 2009, 2010; Vay et al. 2009; Martins et al. 2009; Martins, Fonseca, Lu, et al. 2010; Martins, Fonseca, Vieira, et al. 2010; S. F. Martins et al. 2010; J L Vay et al. 2011; J, Vay et al. 2011; J -L. Vay et al. 2011; Yu et al. 2016), expanding the fields into a truncated series of azimuthal modes (Godfrey 1985; Lifschitz et al. 2009; Davidson et al. 2015; Lehe et al. 2016; Andriyash, Lehe, and Lifschitz 2016), fluid approximation (Krall et al. 1993; Shadwick, Schroeder, and Esarey 2009; Benedetti et al. 2010) and scaled parameters (Cormier-Michel et al. 2009; C. G. R. Geddes et al. 2009).


5.2 Particle-in-Cell

Fig. 5.2: [fig:PIC] The Particle-In-Cell (PIC) method follows the evolution of a collection of charged macro-particles (positively charged in blue on the left plot, negatively charged in red) that evolve self-consistently with their electromagnetic (or electrostatic) fields. The core PIC algorithm involves four operations at each time step: 1) evolve the velocity and position of the particles using the Newton-Lorentz equations, 2) deposit the charge and/or current densities through interpolation from the particles distributions onto the grid, 3) evolve Maxwell’s wave equations (for electromagnetic) or solve Poisson’s equation (for electrostatic) on the grid, 4) interpolate the fields from the grid onto the particles for the next particle push. Additional “add-ons” operations are inserted between these core operations to account for additional physics (e.g. absorption/emission of particles, addition of external forces to account for accelerator focusing or accelerating component) or numerical effects (e.g. smoothing/filtering of the charge/current densities and/or fields on the grid).
In the *electromagnetic particle-in-cell method* (Birdsall and Langdon 1991), the electromagnetic fields are solved on a grid, usually using Maxwell’s equations

\[
\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} \\
\frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{B} - \mathbf{J} \\
\nabla \cdot \mathbf{E} = \rho \\
\nabla \cdot \mathbf{B} = 0
\]

given here in natural units \((\epsilon_0 = \mu_0 = c = 1)\), where \(t\) is time, \(\mathbf{E}\) and \(\mathbf{B}\) are the electric and magnetic field components, and \(\rho\) and \(\mathbf{J}\) are the charge and current densities. The charged particles are advanced in time using the Newton-Lorentz equations of motion

\[
\frac{dx}{dt} = \mathbf{v}, \\
\frac{d(\gamma \mathbf{v})}{dt} = \frac{q}{m} (\mathbf{E} + \mathbf{v} \times \mathbf{B}),
\]

where \(m, q, \mathbf{x}, \mathbf{v}\) and \(\gamma = 1/\sqrt{1 - v^2}\) are respectively the mass, charge, position, velocity and relativistic factor of the particle given in natural units \((c = 1)\). The charge and current densities are interpolated on the grid from the particles’ positions and velocities, while the electric and magnetic field components are interpolated from the grid to the particles’ positions for the velocity update.

### 5.2.1 Particle push

A centered finite-difference discretization of the Newton-Lorentz equations of motion is given by

\[
\frac{x_{i}^{n+1} - x_{i}^{n}}{\Delta t} = \mathbf{v}_{i}^{n+1/2}, \\
\frac{\gamma_{i}^{n+1/2} \mathbf{v}_{i}^{n+1/2} - \gamma_{i}^{n-1/2} \mathbf{v}_{i}^{n-1/2}}{\Delta t} = \frac{q}{m} (\mathbf{E}_{i} + \mathbf{v}_{i}^{n+1/2} \times \mathbf{B}_{i}).
\]

In order to close the system, \(\mathbf{v}_{i}^{n+1/2}\) must be expressed as a function of the other quantities. The two implementations that have become the most popular are presented below.

#### Boris relativistic velocity rotation

The solution proposed by Boris (Boris 1970) is given by

\[
\mathbf{v}_{i}^{n} = \frac{\gamma_{i}^{n+1/2} \mathbf{v}_{i}^{n+1/2} + \gamma_{i}^{n-1/2} \mathbf{v}_{i}^{n-1/2}}{2 \bar{\gamma}_{i}^{n}},
\]

where \(\bar{\gamma}_{i}^{n}\) is defined by \(\bar{\gamma}_{i}^{n} \equiv (\gamma_{i}^{n+1/2} + \gamma_{i}^{n-1/2})/2\).

The system \((Eq:leapfrog_v, Eq:boris_v)\) is solved very efficiently following Boris’ method, where the electric field push is decoupled from the magnetic push. Setting \(\mathbf{u} = \gamma \mathbf{v}\), the velocity is updated using the following sequence:

\[
\mathbf{u}^{-} = \mathbf{u}_{i}^{n-1/2} + (q \Delta t/2m) \mathbf{E}_{i}^{n} \\
\mathbf{u}^{'} = \mathbf{u}^{-} + \mathbf{u}^{-} \times \mathbf{t} \\
\mathbf{u}^{+} = \mathbf{u}^{-} + \mathbf{u}^{'} \times (1 + t^2) \\
\mathbf{u}_{i}^{n+1/2} = \mathbf{u}^{+} + (q \Delta t/2m) \mathbf{E}_{i}^{n}
\]

where \(t = (q \Delta t/2m) \mathbf{B}_{i}^{n}/\bar{\gamma}_{i}^{n}\) and where \(\bar{\gamma}_{i}^{n}\) can be calculated as \(\bar{\gamma}_{i}^{n} = \sqrt{1 + (\mathbf{u}^{-}/c)^2}\).

The Boris implementation is second-order accurate, time-reversible and fast. Its implementation is very widespread and used in the vast majority of PIC codes.
It was shown in (Vay 2008) that the Boris formulation is not Lorentz invariant and can lead to significant errors in the treatment of relativistic dynamics. A Lorentz invariant formulation is obtained by considering the following velocity average

$$\bar{v}^i = \frac{v^{i+1/2} + v^{i-1/2}}{2},$$

This gives a system that is solvable analytically (see (Vay 2008) for a detailed derivation), giving the following velocity update:

$$u^* = u^{i-1/2} + \frac{q\Delta t}{m} \left( E^i + \frac{v^{i-1/2}}{2} \times B^i \right),$$

$$u^{i+1/2} = [u^* + (u^* \cdot t) t + u^* \times t] / (1 + \tau^2),$$

where $$t = \tau / \gamma^{i+1/2}, \tau = (q\Delta t/2m) B^i, \gamma^{i+1/2} = \sqrt{\sigma + \sqrt{\sigma^2 + (\tau^2 + w^2)}}, w = u^* \cdot \tau, \sigma = (\gamma^2 - \tau^2)/2$$ and $$\gamma' = \sqrt{1 + (u^*/c)^2}$$. This Lorentz invariant formulation is particularly well suited for the modeling of ultra-relativistic charged particle beams, where the accurate account of the cancellation of the self-generated electric and magnetic fields is essential, as shown in (Vay 2008).

### 5.2.2 Field solve

Various methods are available for solving Maxwell’s equations on a grid, based on finite-differences, finite-volume, finite-element, spectral, or other discretization techniques that apply most commonly on single structured or unstructured meshes and less commonly on multiblock multiresolution grid structures. In this chapter, we summarize the widespread second order finite-difference time-domain (FDTD) algorithm, its extension to non-standard finite-differences as well as the pseudo-spectral analytical time-domain (PSATD) and pseudo-spectral time-domain (PSTD) algorithms. Extension to multiresolution (or mesh refinement) PIC is described in, e.g. (Vay et al. 2012; Vay, Adam, and Heron 2004).

**Finite-Difference**  

The most popular algorithm for electromagnetic PIC codes is the Finite-Difference Time-Domain (or FDTD) solver

\[
\begin{align*}
D_t B &= -\nabla \times E \\
D_t E &= \nabla \times B - J \\
[\nabla \cdot E] &= \rho \\
[\nabla \cdot B] &= 0.
\end{align*}
\]

The differential operator is defined as $$\nabla = D_x \hat{x} + D_y \hat{y} + D_z \hat{z}$$ and the finite-difference operators in time and space are defined respectively as

$$D_t G \big|_{i,j,k}^{n} = \left( G \big|_{i,j,k}^{n+1/2} - G \big|_{i,j,k}^{n-1/2} \right) / \Delta t$$

and $$D_x G \big|_{i,j,k}^{n} = \left( G \big|_{i+1/2,j,k}^{n} - G \big|_{i-1/2,j,k}^{n} \right) / \Delta x$$, where $$\Delta t$$ and $$\Delta x$$ are respectively the time step and the grid cell size along $$x$$, $$n$$ is the time index and $$i$$, $$j$$, and $$k$$ are the spatial indices along $$x$$, $$y$$ and $$z$$ respectively. The difference operators along $$y$$ and $$z$$ are obtained by circular permutation. The equations in brackets are given for completeness, as they are often not actually solved, thanks to the usage of a so-called charge conserving algorithm, as explained below. As shown in Figure [fig:yee_grid], the quantities are given on a staggered (or “Yee”) grid (Yee 1966), where the electric field components are located between nodes and the magnetic field components are located in the center of the cell faces. Knowing the current densities at half-integer steps, the electric field components are updated alternately with the magnetic field components at integer and half-integer steps respectively.
Fig. 5.3: [fig:yee_grid](left) Layout of field components on the staggered “Yee” grid. Current densities and electric fields are defined on the edges of the cells and magnetic fields on the faces. (right) Time integration using a second-order finite-difference “leapfrog” integrator.

Non-Standard Finite-Difference Time-Domain (NSFDTD)

In (Cole 1997, 2002), Cole introduced an implementation of the source-free Maxwell’s wave equations for narrow-band applications based on non-standard finite-differences (NSFD). In (Karkkainen et al. 2006), Karkkainen et al. adapted it for wideband applications. At the Courant limit for the time step and for a given set of parameters, the stencil proposed in (Karkkainen et al. 2006) has no numerical dispersion along the principal axes, provided that the cell size is the same along each dimension (i.e. cubic cells in 3D). The “Cole-Karkkainen” (or CK) solver uses the non-standard finite difference formulation (based on extended stencils) of the Maxwell-Ampere equation and can be implemented as follows (Vay et al. 2011):

\[
\begin{align*}
D_t B &= -\nabla^* \times E \\
D_t E &= \nabla \times B - J \\
[\nabla \cdot E &= \rho] \\
[\nabla^* \cdot B &= 0]
\end{align*}
\]

Eq. [Eq:Gauss] and [Eq:divb] are not being solved explicitly but verified via appropriate initial conditions and current deposition procedure. The NSFDT differential operators is given by \(\nabla^* = D^*_x \hat{x} + D^*_y \hat{y} + D^*_z \hat{z}\) where

\[
D^*_x = (\alpha + \beta S_{1x} + \xi S_{2x}) D_x \quad S_{1x} = x_{i,j,k} G_{i,j,k}^{n} + G_{i,j+1,k}^{n} + G_{i,j,k+1}^{n} + G_{i,j,k-1}^{n},
\]

\[
S_{2x} G_{i,j,k}^{n} = G_{i,j+1,k}^{n} + G_{i,j-1,k}^{n} + G_{i,j+1,k+1}^{n} + G_{i,j+1,k-1}^{n} + G_{i,j-1,k}^{n} + G_{i,j-1,k+1}^{n} + G_{i,j-1,k-1}^{n} G,\]

\(G\) is a sample vector component, while \(\alpha, \beta\) and \(\xi\) are constant scalars satisfying \(\alpha + 4\beta + 4\xi = 1\). As with the FDTD algorithm, the quantities with half-integer are located between the nodes (electric field components) or in the center of the cell faces (magnetic field components). The operators along \(y\) and \(z\), i.e. \(D_y, D_z, D^*_y, D^*_z, S_{1y}^y, S_{2y}^y, S_{1z}^z, S_{2z}^z\), are obtained by circular permutation of the indices.

Assuming cubic cells (\(\Delta x = \Delta y = \Delta z\)), the coefficients given in (Karkkainen et al. 2006) (\(\alpha = 7/12, \beta = 1/12\) and \(\xi = 1/48\)) allow for the Courant condition to be at \(\Delta t = \Delta x\), which equates to having no numerical dispersion along the principal axes. The algorithm reduces to the FDTD algorithm with \(\alpha = 1\) and \(\beta = \xi = 0\). An extension to non-cubic cells is provided by Cowan, et al. in 3-D in (Cowan et al. 2013) and was given by Pukhov in 2-D in (Pukhov 1999). An alternative NSFDTD implementation that enables superluminous waves is also given by Lehe et al. in (Lehe et al. 2013).

As mentioned above, a key feature of the algorithms based on NSFDTD is that some implementations (Karkkainen et al. 2006; Cowan et al. 2013) enable the time step \(\Delta t = \Delta x\) along one or more axes and no numerical dispersion along those axes. However, as shown in (Vay et al. 2011), an instability develops at the Nyquist wavelength at (or very near) such a timestep. It is also shown in the same paper that removing the Nyquist component in all the source terms using a bilinear filter (see description of the filter below) suppresses this instability.

5.2. Particle-in-Cell Method

163
Maxwell’s equations in Fourier space are given by

\[
\begin{align*}
\frac{\partial \tilde{E}}{\partial t} &= i k \times \tilde{B} - \tilde{J} \\
\frac{\partial \tilde{B}}{\partial t} &= -i k \times \tilde{E} \\
[i k \cdot \tilde{E} &= \tilde{\rho}] \\
[i k \cdot \tilde{B} &= 0]
\end{align*}
\]

where \(\tilde{a}\) is the Fourier Transform of the quantity \(a\). As with the real space formulation, provided that the continuity equation \(\frac{\partial \tilde{\rho}}{\partial t} + i k \cdot \tilde{J} = 0\) is satisfied, then the last two equations will automatically be satisfied at any time if satisfied initially and do not need to be explicitly integrated.

Decomposing the electric field and current between longitudinal and transverse components

\[
\tilde{E} = \tilde{E}_L + \tilde{E}_T = \hat{k}(k \cdot \tilde{E}) - k \times (\hat{k} \times \tilde{E}) \quad \text{and} \quad \tilde{J} = \tilde{J}_L + \tilde{J}_T = \hat{k}(k \cdot \tilde{J}) - k \times (\hat{k} \times \tilde{J})
\]

gives

\[
\begin{align*}
\frac{\partial \tilde{E}_T}{\partial t} &= i k \times \tilde{B} - \tilde{J}_T \\
\frac{\partial \tilde{E}_L}{\partial t} &= -\tilde{J}_L \\
\frac{\partial \tilde{B}}{\partial t} &= -i k \times \tilde{E}
\end{align*}
\]

with \(\hat{k} = k/k\).

If the sources are assumed to be constant over a time interval \(\Delta t\), the system of equations is solvable analytically and is given by (see Haber et al. 1973) for the original formulation and (Jean-Luc Vay, Haber, and Godfrey 2013) for a more detailed derivation):

\[
\text{[Eq:PSATD]}
\]

\[
\begin{align*}
\tilde{E}^{n+1}_T &= C\tilde{E}^n_T + iS\hat{k} \times \tilde{B}^n - \frac{S}{k}\tilde{J}_T^{n+1/2} \\
\tilde{E}^{n+1}_L &= \tilde{E}^n_L - \Delta t\tilde{J}_L^{n+1/2} \\
\tilde{B}^{n+1} &= C\tilde{B}^n - iS\hat{k} \times \tilde{E}^n \\
&\quad + i\frac{1-C}{k}\hat{k} \times \tilde{J}^{n+1/2}
\end{align*}
\]

with \(C = \cos(k\Delta t)\) and \(S = \sin(k\Delta t)\).

Combining the transverse and longitudinal components, gives

\[
\begin{align*}
\tilde{E}^{n+1} &= C\tilde{E}^n + iS\hat{k} \times \tilde{B}^n - \frac{S}{k}\tilde{J}^{n+1/2} \\
&\quad + (1-C)\hat{k}(\hat{k} \cdot \tilde{E}^n) \\
&\quad + \hat{k}(\hat{k} \cdot \tilde{J}^{n+1/2}) \left(\frac{S}{k} - \Delta t\right), \\
\tilde{B}^{n+1} &= C\tilde{B}^n - iS\hat{k} \times \tilde{E}^n \\
&\quad + i\frac{1-C}{k}\hat{k} \times \tilde{J}^{n+1/2}.
\end{align*}
\]

For fields generated by the source terms without the self-consistent dynamics of the charged particles, this algorithm is free of numerical dispersion and is not subject to a Courant condition. Furthermore, this solution is exact for any time step size subject to the assumption that the current source is constant over that time step.
As shown in (Jean-Luc Vay, Haber, and Godfrey 2013), by expanding the coefficients $S_h$ and $C_h$ in Taylor series and keeping the leading terms, the PSATD formulation reduces to the perhaps better known pseudo-spectral time-domain (PSTD) formulation (Dawson 1983; Liu 1997):

$$
\tilde{E}^{n+1}_{\nu} = \tilde{E}^n_{\nu} + i\Delta t k \times \tilde{B}^{n+1/2}_{\nu} - \Delta t \tilde{J}^{n+1/2}_{\nu},
$$
$$
\tilde{B}^{n+3/2}_{\nu} = \tilde{B}^{n+1/2}_{\nu} - i\Delta t k \times \tilde{E}^{n+1}_{\nu}.
$$

The dispersion relation of the PSTD solver is given by $\sin(\omega \Delta t) = \frac{k \Delta t}{2}$. In contrast to the PSATD solver, the PSTD solver is subject to numerical dispersion for a finite time step and to a Courant condition that is given by

$$
\Delta t \leq \frac{2}{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}^{-1/2}.
$$

The PSATD and PSTD formulations that were just given apply to the field components located at the nodes of the grid. As noted in (Ohmura and Okamura 2010), they can also be easily recast on a staggered Yee grid by multiplication of the field components by the appropriate phase factors to shift them from the collocated to the staggered locations. The choice between a collocated and a staggered formulation is application-dependent.

Spectral solvers used to be very popular in the years 1970s to early 1990s, before being replaced by finite-difference methods with the advent of parallel supercomputers that favored local methods. However, it was shown recently that standard domain decomposition with Fast Fourier Transforms that are local to each subdomain could be used effectively with PIC spectral methods (Jean-Luc Vay, Haber, and Godfrey 2013), at the cost of truncation errors in the guard cells that could be neglected. A detailed analysis of the effectiveness of the method with exact evaluation of the magnitude of the effect of the truncation error is given in (Vincenti and Vay 2016) for stencils of arbitrary order (up-to the infinite “spectral” order).

### 5.2.3 Current deposition

The current densities are deposited on the computational grid from the particle position and velocities, employing splines of various orders (Abe et al. 1986).

$$
\rho = \frac{1}{\Delta x \Delta y \Delta z} \sum_n q_n S_n
$$
$$
\mathbf{J} = \frac{1}{\Delta x \Delta y \Delta z} \sum_n q_n \mathbf{v}_n S_n
$$

In most applications, it is essential to prevent the accumulation of errors resulting from the violation of the discretized Gauss’ Law. This is accomplished by providing a method for depositing the current from the particles to the grid that preserves the discretized Gauss’ Law, or by providing a mechanism for “divergence cleaning” (Birdsall and Langdon 1991; Langdon 1992; Marder 1987; Vay and Deutsch 1998; Munz et al. 2000). For the former, schemes that allow a deposition of the current that is exact when combined with the Yee solver is given in (Villasenor and Buneman 1992) for linear splines and in (Esirkepov 2001) for splines of arbitrary order.

The NSFDTD formulations given above and in (Pukhov 1999; Vay et al. 2011; Cowan et al. 2013; Lehe et al. 2013) apply to the Maxwell-Faraday equation, while the discretized Maxwell-Ampere equation uses the FDTD formulation. Consequently, the charge conserving algorithms developed for current deposition (Villasenor and Buneman 1992; Esirkepov 2001) apply readily to those NSFDTD-based formulations. More details concerning those implementations, including the expressions for the numerical dispersion and Courant condition are given in (Pukhov 1999; Vay et al. 2011; Cowan et al. 2013; Lehe et al. 2013).
In the case of the pseudospectral solvers, the current deposition algorithm generally does not satisfy the discretized continuity equation in Fourier space \[ \tilde{\rho}^{n+1} = \tilde{\rho}^n - i \Delta t \mathbf{k} \cdot \tilde{\mathbf{j}}^{n+1/2}. \] In this case, a Boris correction (Birdsall and Langdon 1991) can be applied in \( k \) space in the form \[ \tilde{\mathbf{E}}_c^{n+1} = \tilde{\mathbf{E}}^{n+1} - \left( \mathbf{k} \cdot \tilde{\mathbf{E}}^{n+1} + i \tilde{\rho}^{n+1} \right) \hat{\mathbf{k}}/k, \] where \( \tilde{\mathbf{E}}_c \) is the corrected field. Alternatively, a correction to the current can be applied (with some similarity to the current deposition presented by Morse and Nielson in their potential-based model in (Morse and Nielson 1971)) using \[ \tilde{\mathbf{j}}_c^{n+1/2} = \tilde{\mathbf{j}}^{n+1/2} - \frac{\mathbf{k} \cdot \tilde{\mathbf{j}}^{n+1/2} - i \left( \tilde{\rho}^{n+1} - \tilde{\rho}^n \right) / \Delta t}{\hat{\mathbf{k}}/k}, \] where \( \tilde{\mathbf{j}}_c \) is the corrected current. In this case, the transverse component of the current is left untouched while the longitudinal component is effectively replaced by the one obtained from integration of the continuity equation, ensuring that the corrected current satisfies the continuity equation. The advantage of correcting the current rather than the electric field is that it is more local and thus more compatible with domain decomposition of the fields for parallel computation (Jean Luc Vay, Haber, and Godfrey 2013).

Vay deposition

Alternatively, an exact current deposition can be written for the pseudo-spectral solvers, following the geometrical interpretation of existing methods in real space (Morse and Nielson, 1971; Villasenor and Buneman, 1992; Esirkepov, 2001).

The Vay deposition scheme is the generalization of the Esirkepov deposition scheme for the spectral case with arbitrary-order stencils (Vay et al, 2013). The current density \( \tilde{\mathbf{j}}^{n+1/2} \) in Fourier space is computed as \( \tilde{\mathbf{j}}^{n+1/2} = i \tilde{\mathbf{D}}/k \) when \( k \neq 0 \) and set to zero otherwise. The quantity \( \tilde{\mathbf{D}} \) is deposited in real space by averaging the currents over all possible grid paths between the initial position \( x^n \) and the final position \( x^{n+1} \) and is defined as

- **2D Cartesian geometry:**
  \[
  D_x = \sum_i \frac{1}{\Delta x \Delta z} \frac{q_i u_i}{2\Delta t} \left[ \Gamma(x_i^{n+1}, z_i^{n+1}) - \Gamma(x_i^n, z_i^n) + \Gamma(x_i^{n+1}, z_i^n) - \Gamma(x_i^n, z_i^n) \right] \tag{5.1}
  \]
  \[
  D_y = \sum_i \frac{v_i}{\Delta x \Delta z} \frac{q_i u_i}{4} \left[ \Gamma(x_i^{n+1}, z_i^{n+1}) + \Gamma(x_i^{n+1}, z_i^n) + \Gamma(x_i^n, z_i^{n+1}) + \Gamma(x_i^n, z_i^n) \right] \tag{5.2}
  \]
  \[
  D_z = \sum_i \frac{1}{\Delta x \Delta z} \frac{q_i u_i}{2\Delta t} \left[ \Gamma(x_i^{n+1}, z_i^{n+1}) - \Gamma(x_i^{n+1}, z_i^n) + \Gamma(x_i^n, z_i^{n+1}) - \Gamma(x_i^n, z_i^n) \right] \tag{5.3}
  \]

- **3D Cartesian geometry:**
5.2. Particle-in-Cell Method

5.2.4 Field gather

\[ D_x = \sum_i \frac{1}{\Delta x \Delta y \Delta z} \frac{q_i w_i}{6\Delta t} \left[ 2\Gamma(x_i^{n+1}, y_i^{n+1}, z_i^{n+1}) - 2\Gamma(x_i^n, y_i^n, z_i^n) + \Gamma(x_i^{n+1}, y_i^n, z_i^{n+1}) - \Gamma(x_i^n, y_i^{n+1}, z_i^n) \right] \]

\[ D_y = \sum_i \frac{1}{\Delta x \Delta y \Delta z} \frac{q_i w_i}{6\Delta t} \left[ 2\Gamma(x_i^{n+1}, y_i^{n+1}, z_i^{n+1}) - 2\Gamma(x_i^n, y_i^n, z_i^n) + \Gamma(x_i^{n+1}, y_i^n, z_i^{n+1}) - \Gamma(x_i^n, y_i^{n+1}, z_i^n) \right] \]

\[ D_z = \sum_i \frac{1}{\Delta x \Delta y \Delta z} \frac{q_i w_i}{6\Delta t} \left[ 2\Gamma(x_i^{n+1}, y_i^{n+1}, z_i^{n+1}) - 2\Gamma(x_i^n, y_i^n, z_i^n) + \Gamma(x_i^{n+1}, y_i^n, z_i^{n+1}) - \Gamma(x_i^n, y_i^{n+1}, z_i^n) \right] \]

Here, \( w_i \) represents the weight of the \( i \)-th macro-particle and \( \Gamma \) represents its shape factor. Note that in 2D Cartesian geometry, \( D_y \) is effectively \( J_y \) and does not require additional operations in Fourier space.

5.2.4 Field gather

In general, the field is gathered from the mesh onto the macroparticles using splines of the same order as for the current deposition \( S = (S_x, S_y, S_z) \). Three variations are considered:

- “momentum conserving”: fields are interpolated from the grid nodes to the macroparticles using \( S = (S_{nx}, S_{ny}, S_{nz}) \) for all field components (if the fields are known at staggered positions, they are first interpolated to the nodes on an auxiliary grid),
- “energy conserving (or Galerkin)”: fields are interpolated from the staggered Yee grid to the macroparticles using \( (S_{nx-1}, S_{ny}, S_{nz}) \) for \( E_x \), \( (S_{nx}, S_{ny-1}, S_{nz}) \) for \( E_y \), \( (S_{nx}, S_{ny}, S_{nz-1}) \) for \( E_z \), \( (S_{nx}, S_{ny}, S_{nz-1}) \) for \( B_z \), \( (S_{nx-1}, S_{ny}, S_{nz-1}) \) for \( B_y \), and \( (S_{nx-1}, S_{ny-1}, S_{nz}) \) for \( B_z \) (if the fields are known at the nodes, they are first interpolated to the staggered positions on an auxiliary grid),
- “uniform”: fields are interpolated directly from the Yee grid to the macroparticles using \( S = (S_{nx}, S_{ny}, S_{nz}) \) for all field components (if the fields are known at the nodes, they are first interpolated to the staggered positions on an auxiliary grid).

As shown in \cite{BirdsallLangdon,HockneyEastwoodBook,LewisJCP1972}, the momentum and energy conserving schemes conserve momentum and energy respectively at the limit of infinitesimal time steps and generally offer better conservation of the respective quantities for a finite time step. The uniform scheme does not conserve momentum nor energy in the sense defined for the others but is given for completeness, as it has been shown to offer some interesting properties in the modeling of relativistically drifting plasmas \cite{GodfreyJCP2013}.
5.3 Filtering

It is common practice to apply digital filtering to the charge or current density in Particle-In-Cell simulations as a complement or an alternative to using higher order splines (Birdsall and Langdon 1991). A commonly used filter in PIC simulations is the three points filter \( \phi_j^f = \alpha \phi_j + (1 - \alpha) (\phi_{j-1} + \phi_{j+1}) / 2 \) where \( \phi^f \) is the filtered quantity. This filter is called a bilinear filter when \( \alpha = 0.5 \). Assuming \( \phi = e^{j k x} \) and \( \phi^f = g(\alpha, k) e^{j k x} \), the filter gain \( g \) is given as a function of the filtering coefficient \( \alpha \) and the wavenumber \( k \) by

\[
g(\alpha, k) = \alpha + (1 - \alpha) \cos(k \Delta x) \approx 1 - (1 - \alpha) \frac{(k \Delta x)^2}{2} + O(k^4).
\]

The total attenuation \( G \) for \( n \) successive applications of filters of coefficients \( \alpha_1 \ldots \alpha_n \) is given by

\[
G = \prod_{i=1}^{n} g(\alpha_i, k) \approx 1 - (n - \sum_{i=1}^{n} \alpha_i) \frac{(k \Delta x)^2}{2} + O(k^4).
\]

A sharper cutoff in \( k \) space is provided by using \( \alpha_n = n - \sum_{i=1}^{n} \alpha_i \), so that \( G \approx 1 + O(k^4) \). Such step is called a “compensation” step (Birdsall and Langdon 1991). For the bilinear filter \( (\alpha = 1/2) \), the compensation factor is \( \alpha_c = 2 - 1/2 = 3/2 \). For a succession of \( n \) applications of the bilinear factor, it is \( \alpha_c = n/2 + 1 \).

It is sometimes necessary to filter on a relatively wide band of wavelength, necessitating the application of a large number of passes of the bilinear filter or on the use of filters acting on many points. The former can become very intensive computationally while the latter is problematic for parallel computations using domain decomposition, as the footprint of the filter may eventually surpass the size of subdomains. A workaround is to use a combination of filters of limited footprint. A solution based on the combination of three point filters with various strides was proposed in (Vay et al. 2011) and operates as follows.

The bilinear filter provides complete suppression of the signal at the grid Nyquist wavelength (twice the grid cell size). Suppression of the signal at integer multiples of the Nyquist wavelength can be obtained by using a stride \( s \) in the filter \( \phi_j^f = \alpha \phi_j + (1 - \alpha) (\phi_{j-s} + \phi_{j+s}) / 2 \) for which the gain is given by

\[
g(\alpha, k) = \alpha + (1 - \alpha) \cos(sk \Delta x) \approx 1 - (1 - \alpha) \frac{(sk \Delta x)^2}{2} + O(k^4).
\]

For a given stride, the gain is given by the gain of the bilinear filter shifted in \( k \) space, with the pole \( g = 0 \) shifted from the wavelength \( \lambda = 2/\Delta x \) to \( \lambda = 2 s/\Delta x \), with additional poles, as given by \( sk \Delta x = \arccos \left( \frac{\lambda}{\omega} \right) \mod 2\pi \). The resulting filter is pass band between the poles, but since the poles are spread at different integer values in \( k \) space, a wide band low pass filter can be constructed by combining filters using different strides. As shown in (Vay et al. 2011), the successive application of 4-passes + compensation of filters with strides 1, 2 and 4 has a nearly equivalent fall-off in gain as 80 passes + compensation of a bilinear filter. Yet, the strided filter solution needs only 15 passes of a three-point filter, compared to 81 passes for an equivalent n-pass bilinear filter, yielding a gain of 5.4 in number of operations in favor of the combination of filters with stride. The width of the filter with stride 4 extends only on 9 points, compared to 81 points for a single pass equivalent filter, hence giving a gain of 9 in compactness for the stride filters combination in comparison to the single-pass filter with large stencil, resulting in more favorable scaling with the number of computational cores for parallel calculations.


---

5.3. Filtering 169


### 5.4 Mesh refinement

Fig. 5.4: Sketches of the implementation of mesh refinement in WarpX with the electrostatic (left) and electromagnetic (right) solvers. In both cases, the charge/current from particles are deposited at the finest levels first, then interpolated recursively to coarser levels. In the electrostatic case, the potential is calculated first at the coarsest level $L_0$, the solution interpolated to the boundaries of the refined patch $r$ at the next level $L_1$ and the potential calculated at $L_1$. The procedure is repeated iteratively up to the highest level. In the electromagnetic case, the fields are computed independently on each grid and patch without interpolation at boundaries. Patches are terminated by absorbing layers (PML) to prevent the reflection of electromagnetic waves. Additional coarse patch $c$ and fine grid $a$ are needed so that the full solution is obtained by substitution on $a$ as $F_{n+1}(a) = F_{n+1}(r) + I [F_n(s) - F_{n+1}(c)]$ where $F$ is the field, and $I$ is a coarse-to-fine interpolation operator. In both cases, the field solution at a given level $L_n$ is unaffected by the solution at higher levels $L_{n+1}$ and up, allowing for mitigation of some spurious effects (see text) by providing a transition zone via extension of the patches by a few cells beyond the desired refined area (red & orange rectangles) in which the field is interpolated onto particles from the coarser parent level only.

The mesh refinement methods that have been implemented in WarpX were developed following the following principles: i) avoidance of spurious effects from mesh refinement, or minimization of such effects; ii) user controllability of the spurious effects’ relative magnitude; iii) simplicity of implementation. The two main generic issues that were identified are: a) spurious self-force on macroparticles close to the mesh refinement interface (J. Vay et al. 2002; Colella and Norgaard 2010); b) reflection (and possible amplification) of short wavelength electromagnetic waves at the mesh refinement interface (Vay 2001). The two effects are due to the loss of translation invariance introduced by the asymmetry of the grid on each side of the mesh refinement interface.

In addition, for some implementations where the field that is computed at a given level is affected by the solution at finer levels, there are cases where the procedure violates the integral of Gauss’ Law around the refined patch, leading...
to long range errors (J. Vay et al. 2002; Colella and Norgaard 2010). As will be shown below, in the procedure that 
has been developed in WarpX, the field at a given refinement level is not affected by the solution at finer levels, and is 
thus not affected by this type of error.

5.4.1 Electrostatic

A cornerstone of the Particle-In-Cell method is that assuming a particle lying in a hypothetical infinite grid, then if 
the grid is regular and symmetrical, and if the order of field gathering matches the order of charge (or current) 
deposition, then there is no self-force of the particle acting on itself: a) anywhere if using the so-called “momentum 
conserving” gathering scheme; b) on average within one cell if using the “energy conserving” gathering scheme 
(Birdsall and Langdon 1991). A breaking of the regularity and/or symmetry in the grid, whether it is from the use of 
irregular meshes or mesh refinement, and whether one uses finite difference, finite volume or finite elements, results 
in a net spurious self-force (which does not average to zero over one cell) for a macroparticle close to the point of 
irregularity (mesh refinement interface for the current purpose) (J. Vay et al. 2002; Colella and Norgaard 2010).

A sketch of the implementation of mesh refinement in WarpX is given in Figure [fig:ESAMR] (left). Given the 
solution of the electric potential at a refinement level $L_n$, it is interpolated onto the boundaries of the grid patch(es) at 
the next refined level $L_{n+1}$. The electric potential is then computed at level $L_{n+1}$ by solving the Poisson equation.

This procedure necessitates the knowledge of the charge density at every level of refinement. For efficiency, the 
macroparticle charge is deposited on the highest level patch that contains them, and the charge density of each patch is 
added recursively to lower levels, down to the lowest.

Fig. 5.5: Position history of one charged particle attracted by its image induced by a nearby metallic (dirichlet) boundary. The particle is initialized at rest. Without refinement patch (reference case), the particle is accelerated by its image, is reflected specularly at the wall, then decelerates until it reaches its initial position at rest. If the particle is initialized inside a refinement patch, the particle is initially accelerated toward the wall but is spuriously reflected before it reaches the boundary of the patch whether using the method implemented in WarpX or the MC method. Providing a 
surrounding transition region 2 or 4 cells wide in which the potential is interpolated from the parent coarse solution 
reduces significantly the effect of the spurious self-force.

The presence of the self-force is illustrated on a simple test case that was introduced in (J. Vay et al. 2002) and also 
used in (Colella and Norgaard 2010): a single macroparticle is initialized at rest within a single refinement patch four 
cells away from the patch refinement boundary. The patch at level $L_1$ has $32 \times 32$ cells and is centered relative to the 
lowest $64 \times 64$ grid at level $L_0$ (“main grid”), while the macroparticle is centered in one direction but not in the other.
The boundaries of the main grid are perfectly conducting, so that the macroparticle is attracted to the closest wall by 
its image. Specular reflection is applied when the particle reaches the boundary so that the motion is cyclic. The test 
was performed with WarpX using either linear or quadratic interpolation when gathering the main grid solution onto 
the refined patch boundary. It was also performed using another method from P. McCorquodale et al (labeled “MC” 
in this paper) based on the algorithm given in (McCorquodale et al. 2004), which employs a more elaborate procedure 
involving two-ways interpolations between the main grid and the refined patch. A reference case was also run using a 
single $128 \times 128$ grid with no refined patch, in which it is observed that the particle propagates toward the closest 
boundary at an accelerated pace, is reflected specularly at the boundary, then slows down until it reaches its initial 
position at zero velocity. The particle position histories are shown for the various cases in Fig. [fig:ESselfforce]. In 
all the cases using the refinement patch, the particle was spuriously reflected near the patch boundary and was

5.4. Mesh refinement

171
effectively trapped in the patch. We notice that linear interpolation performs better than quadratic, and that the simple method implemented in WarpX performs better than the other proposed method for this test (see discussion below).

Fig. 5.6: (left) Maps of the magnitude of the spurious self-force $\varepsilon$ in arbitrary units within one quarter of the refined patch, defined as $\varepsilon = \sqrt{(E_x - E_{x ref})^2 + (E_y - E_{y ref})^2}$, where $E_x$ and $E_y$ are the electric field components within the patch experienced by one particle at a given location and $E_{x ref}$ and $E_{y ref}$ are the electric field from a reference solution. The map is given for the WarpX and the MC mesh refinement algorithms and for linear and quadratic interpolation at the patch refinement boundary. (right) Lineouts of the maximum (taken over neighboring cells) of the spurious self-force. Close to the interface boundary ($x=0$), the spurious self-force decreases at a rate close to one order of magnitude per cell (red line), then at about one order of magnitude per six cells (green line).

The magnitude of the spurious self-force as a function of the macroparticle position was mapped and is shown in Fig. [fig:ESselfforcemap] for the WarpX and MC algorithms using linear or quadratic interpolations between grid levels. It is observed that the magnitude of the spurious self-force decreases rapidly with the distance between the particle and the refined patch boundary, at a rate approaching one order of magnitude per cell for the four cells closest to the boundary and about one order of magnitude per six cells beyond. The method implemented in WarpX offers a weaker spurious force on average and especially at the cells that are the closest to the coarse-fine interface where it is the largest and thus matters most. We notice that the magnitude of the spurious self-force depends strongly on the distance to the edge of the patch and to the nodes of the underlying coarse grid, but weakly on the order of deposition and size of the patch.

A method was devised and implemented in WarpX for reducing the magnitude of spurious self-forces near the coarse-fine boundaries as follows. Noting that the coarse grid solution is unaffected by the presence of the patch and is thus free of self-force, extra “transition” cells are added around the “effective” refined area. Within the effective area, the particles gather the potential in the fine grid. In the extra transition cells surrounding the refinement patch, the force is gathered directly from the coarse grid (an option, which has not yet been implemented, would be to interpolate between the coarse and fine grid field solutions within the transition zone so as to provide continuity of the force experienced by the particles at the interface). The number of cells allocated in the transition zones is controllable by the user in WarpX, giving the opportunity to check whether the spurious self-force is affecting the calculation by repeating it using different thicknesses of the transition zones. The control of the spurious force using the transition zone is illustrated in Fig. [fig:ESselfforce], where the calculation with WarpX using linear interpolation at the patch interface was repeated using either two or four cells transition regions (measured in refined patch cell units). Using two extra cells allowed for the particle to be free of spurious trapping within the refined area and follow a trajectory that is close to the reference one, and using four extra cells improved further to the point where the resulting trajectory becomes indistinguishable from the reference one. We note that an alternative method was devised for reducing the magnitude of self-force near the coarse-fine boundaries for the MC method, by using a special deposition procedure near the interface (Colella and Norgaard 2010).
5.4.2 Electromagnetic

The method that is used for electrostatic mesh refinement is not directly applicable to electromagnetic calculations. As was shown in section 3.4 of (Vay 2001), refinement schemes relying solely on interpolation between coarse and fine patches lead to the reflection with amplification of the short wavelength modes that fall below the cutoff of the Nyquist frequency of the coarse grid. Unless these modes are damped heavily or prevented from occurring at their source, they may affect particle motion and their effect can escalate if trapped within a patch, via multiple successive reflections with amplification.

To circumvent this issue, an additional coarse patch (with the same resolution as the parent grid) is added, as shown in Fig. 5.4.2-right and described in (Vay, Adam, and Heron 2004). Both the fine and the coarse grid patches are terminated by Perfectly Matched Layers, reducing wave reflection by orders of magnitude, controllable by the user (Berenger 1996; J.-L. Vay 2002). The source current resulting from the motion of charged macroparticles within the refined region is accumulated on the fine patch and is then interpolated onto the coarse patch and added onto the parent grid. The process is repeated recursively from the finest level down to the coarsest. The Maxwell equations are then solved for one time interval on the entire set of grids, by default for one time step using the time step of the finest grid. The field on the coarse and fine patches only contain the contributions from the particles that have evolved within the refined area but not from the current sources outside the area. The total contribution of the field from sources within and outside the refined area is obtained by adding the field from the refined grid $F(r)$, and adding an interpolation $f$ of the difference between the relevant subset $s$ of the field in the parent grid $F(s)$ and the field of the coarse grid $F(c)$, on an auxiliary grid $a$, i.e. $F(a) = F(r) + f[F(s) - F(c)]$. The field on the parent grid subset $F(s)$ contains contributions from sources from both within and outside of the refined area. Thus, in effect, there is substitution of the coarse field resulting from sources within the patch area by its fine resolution counterpart. The operation is carried out recursively starting at the coarsest level up to the finest. An option has been implemented in which various grid levels are pushed with different time steps, given as a fixed fraction of the individual grid Courant conditions (assuming same cell aspect ratio for all grids and refinement by integer factors). In this case, the fields from the coarse levels, which are advanced less often, are interpolated in time.

The substitution method has two potential drawbacks due to the inexact cancellation between the coarse and fine patches of: (i) the remnants of ghost fixed charges created by the particles entering and leaving the patches (this effect is due to the use of the electromagnetic solver and is different from the spurious self-force that was described for the electrostatic case); (ii) if using a Maxwell solver with a low-order stencil, the electromagnetic waves traveling on each patch at slightly different velocity due to numerical dispersion. The first issue results in an effective spurious multipole field whose magnitude decreases very rapidly with the distance to the patch boundary, similarly to the spurious self-force in the electrostatic case. Hence, adding a few extra transition cells surrounding the patches mitigates this effect very effectively. The tunability of WarpX’s electromagnetic finite-difference and pseudo-spectral solvers provides the means to optimize the numerical dispersion so as to minimize the second effect for a given application, which has been demonstrated on the laser-plasma interaction test case presented in (Vay, Adam, and Heron 2004). Both effects and their mitigation are described in more detail in (Vay, Adam, and Heron 2004).

Caustics are supported anywhere on the grid with an accuracy that is set by the local resolution, and will be adequately resolved if the grid resolution supports the necessary modes from their sources to the points of wavefront crossing. The mesh refinement method that is implemented in WarpX has the potential to provide higher efficiency than the standard use of fixed gridding, by offering a path toward adaptive gridding following wavefronts.
5.5 Boundary conditions

5.5.1 Open boundary condition for electromagnetic waves

For the TE case, the original Berenger’s Perfectly Matched Layer (PML) writes

\[
\begin{align*}
\varepsilon \frac{\partial E_x}{\partial t} + \sigma_y E_x &= \frac{\partial H_z}{\partial y} \\
\varepsilon \frac{\partial E_y}{\partial t} + \sigma_x E_y &= - \frac{\partial H_z}{\partial x} \\
\mu \frac{\partial H_{zx}}{\partial t} + \sigma_x^* H_{zx} &= - \frac{\partial E_y}{\partial x} \\
\mu \frac{\partial H_{zy}}{\partial t} + \sigma_y^* H_{zy} &= \frac{\partial E_x}{\partial y} \\
H_z &= H_{zx} + H_{zy}
\end{align*}
\]

This can be generalized to

\[
\begin{align*}
\varepsilon \frac{\partial E_x}{\partial t} + \sigma_y E_x &= \frac{c_y}{c} \frac{\partial H_z}{\partial y} + \sigma_y H_z \\
\varepsilon \frac{\partial E_y}{\partial t} + \sigma_x E_y &= - \frac{c_x}{c} \frac{\partial H_z}{\partial x} + \sigma_x H_z \\
\mu \frac{\partial H_{zx}}{\partial t} + \sigma_x^* H_{zx} &= - \frac{c_y}{c} \frac{\partial E_y}{\partial x} + \sigma_x^* E_y \\
\mu \frac{\partial H_{zy}}{\partial t} + \sigma_y^* H_{zy} &= \frac{c_y}{c} \frac{\partial E_x}{\partial y} + \sigma_y^* E_x \\
H_z &= H_{zx} + H_{zy}
\end{align*}
\]

For \(c_x = c_y = c_x^* = c_y^* = c\) and \(\sigma_x = \sigma_y = \sigma_x^* = \sigma_y^* = 0\), this system reduces to the Berenger PML medium, while adding the additional constraint \(\sigma_x = \sigma_y = \sigma_x^* = \sigma_y^* = 0\) leads to the system of Maxwell equations in vacuum.
We consider a plane wave of magnitude \((E_0, H_{xz0}, H_{zy0})\) and pulsation \(\omega\) propagating in the APML medium with an angle \(\varphi\) relative to the x axis

\[
\begin{align*}
E_x &= -E_0 \sin \varphi e^{i\omega(t-\alpha x - \beta y)} \\
E_y &= E_0 \cos \varphi e^{i\omega(t-\alpha x - \beta y)} \\
H_{zx} &= H_{zx0} e^{i\omega(t-\alpha x - \beta y)} \\
H_{zy} &= H_{zy0} e^{i\omega(t-\alpha x - \beta y)}
\end{align*}
\]

where \(\alpha\) and \(\beta\) are two complex constants to be determined.

Introducing \((\text{Plane\_wave\_APML\_def\_1}), (\text{Plane\_wave\_APML\_def\_2}), (\text{Plane\_wave\_AMPL\_def\_3})\) and \((\text{Plane\_wave\_APML\_def\_4})\) into \((\text{APML\_def\_1}), (\text{APML\_def\_2}), (\text{APML\_def\_3})\) and \((\text{APML\_def\_4})\) gives

\[
\begin{align*}
\varepsilon_0 E_0 \sin \varphi - i \frac{\sigma_y}{\omega} E_0 \sin \varphi &= \beta \frac{c_y}{c} (H_{zx0} + H_{zy0}) + i \frac{\sigma_y}{\omega} (H_{zx0} + H_{zy0}) \\
\varepsilon_0 E_0 \cos \varphi - i \frac{\sigma_x}{\omega} E_0 \cos \varphi &= \alpha \frac{c_x}{c} (H_{zx0} + H_{zy0}) - i \frac{\sigma_x}{\omega} (H_{zx0} + H_{zy0}) \\
\mu_0 H_{zx0} - i \frac{\sigma_x}{\omega} H_{zx0} &= \beta \frac{c_y}{c} E_0 \cos \varphi - i \frac{\sigma_x}{\omega} E_0 \cos \varphi \\
\mu_0 H_{zy0} - i \frac{\sigma_y}{\omega} H_{zy0} &= \beta \frac{c_y}{c} E_0 \sin \varphi + i \frac{\sigma_y}{\omega} E_0 \sin \varphi
\end{align*}
\]

Defining \(Z = E_0 / (H_{zx0} + H_{zy0})\) and using \((\text{Plane\_wave\_APML\_I\_1}), (\text{Plane\_wave\_APML\_I\_2})\), we get

\[
\beta = \left[ Z \left( \varepsilon_0 - i \frac{\sigma_y}{\omega} \right) \sin \varphi - i \frac{\sigma_y}{\omega} \right] \frac{c}{c_y} \\
\alpha = \left[ Z \left( \varepsilon_0 - i \frac{\sigma_x}{\omega} \right) \cos \varphi + i \frac{\sigma_x}{\omega} \right] \frac{c}{c_x}
\]

Adding \(H_{zx0}\) and \(H_{zy0}\) from \((\text{Plane\_wave\_APML\_I\_3}), (\text{Plane\_wave\_APML\_I\_4})\) and substituting the expressions for \(\alpha\) and \(\beta\) from \((\text{Plane\_wave\_APML\_beta\_of\_g}), (\text{Plane\_wave\_APML\_alpha\_of\_g})\) yields

\[
\begin{align*}
\frac{1}{Z} &= Z \left( \varepsilon_0 - i \frac{\sigma_x}{\omega} \right) \cos \varphi \frac{c_x}{\varepsilon_0} + i \frac{\sigma_x}{\omega} \frac{c_x}{\varepsilon_0} - i \frac{\sigma_x}{\omega} \cos \varphi \\
&+ Z \left( \varepsilon_0 - i \frac{\sigma_y}{\omega} \right) \sin \varphi \frac{c_y}{\varepsilon_0} - i \frac{\sigma_y}{\omega} \frac{c_y}{\varepsilon_0} + i \frac{\sigma_y}{\omega} \sin \varphi \\
&\mu_0 - i \frac{\sigma_x}{\omega} \frac{\sigma_y}{\varepsilon_0} \mu_0 - i \frac{\sigma_y}{\omega} \frac{\sigma_x}{\varepsilon_0} - i \frac{\sigma_x}{\omega} \mu_0 - i \frac{\sigma_y}{\omega}
\end{align*}
\]

If \(c_x = c_x^*, c_y = c_y^*, \sigma_x = \sigma_x^*, \sigma_y = \sigma_y^*, \frac{\sigma_x}{\varepsilon_0} = \frac{\sigma_y}{\mu_0}\) and \(\frac{\sigma_y}{\varepsilon_0} = \frac{\sigma_x}{\mu_0}\) then

\[
Z = \pm \sqrt{\frac{\mu_0}{\varepsilon_0}}
\]

which is the impedance of vacuum. Hence, like the PML, given some restrictions on the parameters, the APML does not generate any reflection at any angle and any frequency. As for the PML, this property is not retained after discretization, as shown subsequently in this paper.

Calling \(\psi\) any component of the field and \(\psi_0\) its magnitude, we get from \((\text{Plane\_wave\_APML\_def\_1}), (\text{Plane\_wave\_APML\_beta\_of\_g}), (\text{Plane\_wave\_APML\_alpha\_of\_g})\) and \((\text{APML\_impedance})\) that

\[
\psi = \psi_0 e^{i\omega(t + x \cos \varphi / c_x + y \sin \varphi / c_y)} e^{-\left( \pm \frac{\sigma_x \cos \varphi}{\varepsilon_0} + \sigma_x \frac{\omega}{c_x} \right) x} e^{-\left( \pm \frac{\sigma_y \sin \varphi}{\varepsilon_0} + \sigma_y \frac{\omega}{c_y} \right) y}
\]

5.5. Boundary conditions

175
We assume that we have an APML layer of thickness $\delta$ (measured along $x$) and that $\sigma_y = \sigma_x = 0$ and $c_y = c$. Using $(\text{Plane\_wave\_absorption})$, we determine that the coefficient of reflection given by this layer is

$$R_{APML}(\theta) = e^{-(\sigma_x \cos \varphi / c_0 c_x + \sigma_x c / c_x) \delta} e^{-(\sigma_x \cos \varphi / c_0 c_x - \sigma_x c / c_x) \delta} = e^{-2(\sigma_x \cos \varphi / c_0 c_x) \delta}$$

which happens to be the same as the PML theoretical coefficient of reflection if we assume $c_x = c$. Hence, it follows that for the purpose of wave absorption, the term $\sigma_x$ seems to be of no interest. However, although this conclusion is true at the infinitesimal limit, it does not hold for the discretized counterpart.

**Discretization**

\[
\begin{align*}
\frac{E_x|_{j+1/2,k,l}^{n+1} - E_x|_{j+1/2,k,l}^n}{\Delta t} + \sigma_y \frac{E_x|_{j+1/2,k,l}^{n+1} + E_x|_{j+1/2,k,l}^n}{2} &= \frac{H_z|_{j+1/2,k+1/2,l}^{n+1} - H_z|_{j+1/2,k-1/2,l}^{n+1}}{\Delta y} \\
\frac{E_y|_{j,k+1/2,l}^{n+1} - E_y|_{j,k+1/2,l}^n}{\Delta t} + \sigma_x \frac{E_y|_{j,k+1/2,l}^{n+1} + E_y|_{j,k+1/2,l}^n}{2} &= -\frac{H_e|_{j+1/2,k+1/2,l}^{n+1} - H_e|_{j-1/2,k+1/2,l}^{n+1}}{\Delta x} \\
\frac{H_{zz}|_{j+1/2,k+1/2,l}^{n+3/2} - H_{zz}|_{j+1/2,k+1/2,l}^n}{\Delta t} + \sigma_z H_{zz}|_{j+1/2,k+1/2,l}^{n+3/2} + H_{zz}|_{j+1/2,k+1/2,l}^n}{2} &= \frac{E_x|_{j+1/2,k+1/2,l}^{n+1} - E_x|_{j+1/2,k+1/2,l}^n}{\Delta y} \\
H_z &= H_{zx} + H_{zy} \\
E_x|_{j+1/2,k,l}^{n+1} &= \left(1 - \frac{\sigma_y \Delta t/2}{1 + \sigma_y \Delta t/2}\right) E_x|_{j+1/2,k,l}^n + \frac{\Delta t/\Delta y}{1 + \sigma_y \Delta t/2} \left(H_z|_{j+1/2,k+1/2,l}^{n+1} - H_z|_{j+1/2,k-1/2,l}^{n+1}\right) \\
E_y|_{j,k+1/2,l}^{n+1} &= \left(1 - \frac{\sigma_x \Delta t/2}{1 + \sigma_x \Delta t/2}\right) E_y|_{j,k+1/2,l}^n - \frac{\Delta t/\Delta x}{1 + \sigma_x \Delta t/2} \left(H_e|_{j+1/2,k+1/2,l}^{n+1} - H_e|_{j-1/2,k+1/2,l}^{n+1}\right) \\
H_{xx}|_{j+1/2,k+1/2,l}^{n+3/2} &= \left(1 - \frac{\sigma_z \Delta t/2}{1 + \sigma_z \Delta t/2}\right) H_{xx}|_{j+1/2,k+1/2,l}^n + \frac{\Delta t/\Delta y}{1 + \sigma_z \Delta t/2} \left(E_x|_{j+1/2,k+1/2,l}^{n+1} - E_x|_{j+1/2,k+1/2,l}^n\right) \\
H_{zy}|_{j+1/2,k+1/2,l}^{n+3/2} &= \left(1 - \frac{\sigma_y \Delta t/2}{1 + \sigma_y \Delta t/2}\right) H_{zy}|_{j+1/2,k+1/2,l}^n + \frac{\Delta t/\Delta x}{1 + \sigma_y \Delta t/2} \left(E_y|_{j+1/2,k+1/2,l}^{n+1} - E_y|_{j+1/2,k+1/2,l}^n\right) \\
H_z &= H_{zx} + H_{zy}
\end{align*}
\]
5.6 Moving window and optimal Lorentz boosted frame

The simulations of plasma accelerators from first principles are extremely computationally intensive, due to the need to resolve the evolution of a driver (laser or particle beam) and an accelerated particle beam into a plasma structure that is orders of magnitude longer and wider than the accelerated beam. As is customary in the modeling of particle beam dynamics in standard particle accelerators, a moving window is commonly used to follow the driver, the wake and the accelerated beam. This results in huge savings, by avoiding the meshing of the entire plasma that is orders of magnitude longer than the other length scales of interest.

Even using a moving window, however, a full PIC simulation of a plasma accelerator can be extraordinarily demanding computationally, as many time steps are needed to resolve the crossing of the short driver beam with the plasma column. As it turns out, choosing an optimal frame of reference that travels close to the speed of light in the
direction of the laser or particle beam (as opposed to the usual choice of the laboratory frame) enables speedups by orders of magnitude (Vay 2007; J.-L. Vay et al. 2011). This is a result of the properties of Lorentz contraction and dilation of space and time. In the frame of the laboratory, a very short driver (laser or particle) beam propagates through a much longer plasma column, necessitating millions to tens of millions of time steps for parameters in the range of the BELLA or FACET-II experiments. As sketched in Fig. [fig:PIC], in a frame moving with the driver beam in the plasma at velocity $v = \beta c$ (where $c$ is the speed of light in vacuum), the beam length is now elongated by $\approx (1 + \beta)\gamma$ while the plasma contracts by $\gamma$ (where $\gamma = 1/\sqrt{1 - \beta^2}$ is the relativistic factor associated with the frame velocity). The number of time steps that is needed to simulate a “longer” beam through a “shorter” plasma is now reduced by up to $\approx (1 + \beta)\gamma^2$ (a detailed derivation of the speedup is given below).

The modeling of a plasma acceleration stage in a boosted frame involves the fully electromagnetic modeling of a plasma propagating at near the speed of light, for which Numerical Cerenkov (Boris and Lee 1973; Haber et al. 1973) is a potential issue, as explained in more details below. In addition, for a frame of reference moving in the direction of the accelerated beam (or equivalently the wake of the laser), waves emitted by the plasma in the forward direction expand while the ones emitted in the backward direction contract, following the properties of the Lorentz transformation. If one had to resolve both forward and backward propagating waves emitted from the plasma, there would be no gain in selecting a frame different from the laboratory frame. However, the physics of interest for a laser wakefield is the laser driving the wake, the wake, and the accelerated beam. Backscatter is weak in the short-pulse regime, and does not interact as strongly with the beam as do the forward propagating waves which stay in phase for a long period. It is thus often assumed that the backward propagating waves can be neglected in the modeling of plasma accelerator stages. The accuracy of this assumption has been demonstrated by comparison between explicit codes which include both forward and backward waves and envelope or quasistatic codes which neglect backward waves (Geddes et al. 2008; Geddes et al. 2009; Cowan et al. 2009).

### 5.6.1 Theoretical speedup dependency with the frame boost

The derivation that is given here reproduces the one given in (J.-L. Vay et al. 2011), where the obtainable speedup is derived as an extension of the formula that was derived earlier (Vay 2007), taking in addition into account the group velocity of the laser as it traverses the plasma.

Assuming that the simulation box is a fixed number of plasma periods long, which implies the use (which is standard) of a moving window following the wake and accelerated beam, the speedup is given by the ratio of the time taken by the laser pulse and the plasma to cross each other, divided by the shortest time scale of interest, that is the laser period. To first order, the wake velocity $v_w$ is set by the 1D group velocity of the laser driver, which in the linear (low
The intensity limit, is given by (Esarey, Schroeder, and Leemans 2009):

\[ \frac{v_w}{c} = \beta_w = \left(1 - \frac{\omega_p^2}{\omega^2}\right)^{1/2} \]

where \( \omega_p = \sqrt{(n_ee^2)/(\varepsilon_0me)} \) is the plasma frequency, \( \omega = 2\pi c/\lambda \) is the laser frequency, \( n_e \) is the plasma density, \( \lambda \) is the laser wavelength in vacuum, \( \varepsilon_0 \) is the permittivity of vacuum, \( c \) is the speed of light in vacuum, and \( e \) and \( m_e \) are respectively the charge and mass of the electron.

In practice, the runs are typically stopped when the last electron beam macro-particle exits the plasma, and a measure of the total time of the simulation is then given by

\[ T = \frac{L + \eta \lambda_p}{v_w - v_p} \]

where \( \lambda_p \approx 2\pi c/\omega_p \) is the wake wavelength, \( L \) is the plasma length, \( v_w \) and \( v_p = \beta_p c \) are respectively the velocity of the wake and of the plasma relative to the frame of reference, and \( \eta \) is an adjustable parameter for taking into account the fraction of the wake which exited the plasma at the end of the simulation. For a beam injected into the \( n^{th} \) bucket, \( \eta \) would be set to \( n - 1/2 \). If positrons were considered, they would be injected half a wake period ahead of the location of the electrons injection position for a given period, and one would have \( \eta = n - 1 \). The numerical cost \( R_t \) scales as the ratio of the total time to the shortest timescale of interest, which is the inverse of the laser frequency, and is thus given by

\[ R_t = \frac{Tc}{\lambda} = \frac{(L + \eta \lambda_p)}{(\beta_w - \beta_p) \lambda} \]

In the laboratory, \( v_p = 0 \) and the expression simplifies to

\[ R_{lab} = \frac{Tc}{\lambda} = \frac{(L + \eta \lambda_p)}{\beta_w \lambda} \]

In a frame moving at \( \beta_c \), the quantities become

\[ \lambda_p^* = \frac{\lambda_p}{\gamma (1 - \beta_w \beta)} \]
\[ L^* = \frac{L}{\gamma} \]
\[ \lambda^* = \gamma (1 + \beta) \lambda \]
\[ \beta_w^* = \frac{(\beta_w - \beta)}{(1 - \beta_w \beta)} \]
\[ v_p^* = -\beta c \]
\[ T^* = \frac{L^* + \eta \lambda_p^*}{v_w^* - v_p^*} \]
\[ R_t^* = \frac{T^* c}{\lambda^*} = \frac{(L^* + \eta \lambda_p^*)}{(\beta_c^* + \beta) \lambda^*} \]

where \( \gamma = 1/\sqrt{1 - \beta^2} \).

The expected speedup from performing the simulation in a boosted frame is given by the ratio of \( R_{lab} \) and \( R_t^* \)

\[ S = \frac{R_{lab}}{R_t^*} = \frac{(1 + \beta) (L + \eta \lambda_p)}{(1 - \beta_w) L + \eta \lambda_p} \]

We note that assuming that \( \beta_w \approx 1 \) (which is a valid approximation for most practical cases of interest) and that \( \gamma < \gamma_w \), this expression is consistent with the expression derived earlier (Vay 2007) for the laser-plasma acceleration case, which states that \( R_t^* = \alpha R_t / (1 + \beta) \) with \( \alpha = (1 - \beta + l/L) / (1 + l/L) \), where \( l \) is the laser length which is generally proportional to \( \eta \lambda_p \), and \( S = R_t / R_t^* \). However, higher values of \( \gamma \) are of interest for maximum speedup, as shown below.

5.6. Moving window and optimal Lorentz boosted frame
For intense lasers ($a \sim 1$) typically used for acceleration, the energy gain is limited by dephasing (Schroeder et al. 2011), which occurs over a scale length $L_d \sim \lambda_p^3/2\lambda^2$. Acceleration is compromised beyond $L_d$ and in practice, the plasma length is proportional to the dephasing length, i.e. $L = \xi L_d$. In most cases, $\gamma_w^2 >> 1$, which allows the approximations $\beta_w \approx 1 - \lambda^2/2\lambda_p^2$ and $L = \xi \lambda_p^3/2\lambda^2 \approx \xi \gamma_w^2 \lambda_p/2 >> \eta \lambda_p$, so that Eq.((Eq_scaling1d0)) becomes

$$S = (1 + \beta)^2 \frac{\xi \gamma_w^2}{\xi \gamma_w^2 + (1 + \beta)^2 (\xi \beta/2 + 2\eta)}$$

For low values of $\gamma$, i.e. when $\gamma << \gamma_w$, Eq.((Eq_scaling1d)) reduces to

$$S_{\gamma<<\gamma_w} = (1 + \beta)^2 \gamma^2$$

Conversely, if $\gamma \rightarrow \infty$, Eq.((Eq_scaling1d)) becomes

$$S_{\gamma\rightarrow\infty} = \frac{4}{1 + 4\eta/\xi \gamma_w^2}$$

Finally, in the frame of the wake, i.e. when $\gamma = \gamma_w$, assuming that $\beta_w \approx 1$, Eq.((Eq_scaling1d)) gives

$$S_{\gamma=\gamma_w} \approx \frac{2}{1 + 2\eta/\xi \gamma_w^2}$$

Since $\eta$ and $\xi$ are of order unity, and the practical regimes of most interest satisfy $\gamma_w^2 >> 1$, the speedup that is obtained by using the frame of the wake will be near the maximum obtainable value given by Eq.((Eq_scaling_gamma_inf)).

Note that without the use of a moving window, the relativistic effects that are at play in the time domain would also be at play in the spatial domain (Vay 2007), and the $\gamma^2$ scaling would transform to $\gamma^4$. Hence, it is important to use a moving window even in simulations in a Lorentz boosted frame. For very high values of the boosted frame, the optimal velocity of the moving window may vanish (i.e. no moving window) or even reverse.

**5.6.2 Numerical Stability and alternate formulation in a Galilean frame**

The numerical Cherenkov instability (NCI) (Godfrey 1974) is the most serious numerical instability affecting multidimensional PIC simulations of relativistic particle beams and streaming plasmas (Martins et al. 2010; Vay et al. 2010; J L Vay et al. 2011; Sironi and Spitkovsky 2011; Godfrey and Vay 2013; Xu et al. 2013). It arises from coupling between possibly numerically distorted electromagnetic modes and spurious beam modes, the latter due to the mismatch between the Lagrangian treatment of particles and the Eulerian treatment of fields (Godfrey 1975).

In recent papers the electromagnetic dispersion relations for the numerical Cherenkov instability were derived and solved for both FDTD (Godfrey and Vay 2013; Brendan B. Godfrey and Vay 2014) and PSATD (Brendan B. Godfrey, Vay, and Haber 2014a, 2014b) algorithms.

Several solutions have been proposed to mitigate the NCI (Brendan B Godfrey, Vay, and Haber 2014; Brendan B. Godfrey, Vay, and Haber 2014b, 2014a; Godfrey and Vay 2015; Yu, Xu, Decyk, et al. 2015; Yu, Xu, Tableman, et al. 2015). Although these solutions efficiently reduce the numerical instability, they typically introduce either strong smoothing of the currents and fields, or arbitrary numerical corrections, which are tuned specifically against the NCI and go beyond the natural discretization of the underlying physical equation. Therefore, it is sometimes unclear to what extent these added corrections could impact the physics at stake for a given resolution.

For instance, NCI-specific corrections include periodically smoothing the electromagnetic field components (Martins et al. 2010), using a special time step (Vay et al. 2010; J L Vay et al. 2011) or applying a wide-band smoothing of the current components (Vay et al. 2010; J L Vay et al. 2011; J. Vay et al. 2011). Another set of mitigation methods involve scaling the deposited currents by a carefully-designed wavenumber-dependent factor (Brendan B. Godfrey and Vay 2014; Brendan B. Godfrey, Vay, and Haber 2014b) or slightly modifying the ratio of electric and magnetic fields ($E/B$) before gathering their value onto the macroparticles (Brendan B. Godfrey, Vay, and Haber 2014a; Godfrey and Vay 2015). Yet another set of NCI-specific corrections (Yu, Xu, Decyk, et al. 2015; Yu, Xu, Tableman,
et al. 2015) consists in combining a small timestep $\Delta t$, a sharp low-pass spatial filter, and a spectral or high-order scheme that is tuned so as to create a small, artificial “bump” in the dispersion relation (Yu, Xu, Decyk, et al. 2015). While most mitigation methods have only been applied to Cartesian geometry, this last set of methods ((Yu, Xu, Decyk, et al. 2015; Yu, Xu, Tableman, et al. 2015)) has the remarkable property that it can be applied (Yu, Xu, Tableman, et al. 2015) to both Cartesian geometry and quasi-cylindrical geometry (i.e. cylindrical geometry with azimuthal Fourier decomposition (Lifschitz et al. 2009; Davidson et al. 2015; R. Lehe et al. 2016)). However, the use of a small timestep proportionally slows down the progress of the simulation, and the artificial “bump” is again an arbitrary correction that departs from the underlying physics.

A new scheme was recently proposed, in (Kirchen et al. 2016; Lehe et al. 2016), which completely eliminates the NCI for a plasma drifting at a uniform relativistic velocity – with no arbitrary correction – by simply integrating the PIC equations in Galilean coordinates (also known as comoving coordinates). More precisely, in the new method, the Maxwell equations in Galilean coordinates are integrated analytically, using only natural hypotheses, within the PSATD framework (Pseudo-Spectral-Analytical-Time-Domain (Haber et al. 1973; Vay, Haber, and Godfrey 2013)). The idea of the proposed scheme is to perform a Galilean change of coordinates, and to carry out the simulation in the new coordinates:

\[
 x' = x - v_{gal} t
\]

where \( x = x u_x + y u_y + z u_z \) and \( x' = x' u_x + y' u_y + z' u_z \) are the position vectors in the standard and Galilean coordinates respectively.

When choosing \( v_{gal} = v_0 \), where \( v_0 \) is the speed of the bulk of the relativistic plasma, the plasma does not move with respect to the grid in the Galilean coordinates \( x' \) – or, equivalently, in the standard coordinates \( x \), the grid moves along with the plasma. The heuristic intuition behind this scheme is that these coordinates should prevent the discrepancy between the Lagrangian and Eulerian point of view, which gives rise to the NCI (Godfrey 1975).

An important remark is that the Galilean change of coordinates (\( \text{eq:change-var} \)) is a simple translation. Thus, when used in the context of Lorentz-boosted simulations, it does of course preserve the relativistic dilatation of space and time which gives rise to the characteristic computational speedup of the boosted-frame technique. Another important remark is that the Galilean scheme is not equivalent to a moving window (and in fact the Galilean scheme can be independently combined with a moving window). Whereas in a moving window, gridpoints are added and removed so as to effectively translate the boundaries, in the Galilean scheme the gridpoints themselves are not only translated but in this case, the physical equations are modified accordingly. Most importantly, the assumed time evolution of the current \( J \) within one timestep is different in a standard PSATD scheme with moving window and in a Galilean PSATD scheme (Lehe et al. 2016).

In the Galilean coordinates \( x' \), the equations of particle motion and the Maxwell equations take the form

\[
 \frac{dx'}{dt} = \frac{p}{\gamma m} - v_{gal} \\
 \frac{dp}{dt} = q \left( E + \frac{p}{\gamma m} \times B \right) \\
 \frac{\partial}{\partial t} - v_{gal} \cdot \nabla' B = -\nabla' \times E \\
 \frac{1}{c^2} \frac{\partial}{\partial t} - v_{gal} \cdot \nabla' E = \nabla' \times B - \mu_0 J
\]

where \( \nabla' \) denotes a spatial derivative with respect to the Galilean coordinates \( x' \).

Integrating these equations from \( t = n \Delta t \) to \( t = (n + 1) \Delta t \) results in the following update equations (see (Lehe et al.

5.6. Moving window and optimal Lorentz boosted frame

181
\[
\begin{align*}
\tilde{B}^{n+1} &= \theta^2 C \tilde{B}^n - \frac{\theta^2 S}{ck} i k \times \tilde{E}^n \\
&\quad + \frac{\theta \chi_1}{\epsilon_0 ek^2} i k \times \tilde{j}^{n+1/2} \\
\tilde{E}^{n+1} &= \theta^2 C \tilde{E}^n + \frac{\theta^2 S}{k} c i k \times \tilde{B}^n \\
&\quad + \frac{i \nu \theta \chi_1 - \theta^2 S}{\epsilon_0 kc} \tilde{j}^{n+1/2} \\
&\quad - \frac{1}{\epsilon_0 k^2} \left( \chi_2 \bar{\rho}^{n+1} - \theta^2 \chi_3 \bar{\rho}^n \right) i k
\end{align*}
\]

where we used the short-hand notations \( \tilde{E}^n \equiv \tilde{E}(k, n \Delta t) \), \( \tilde{B}^n \equiv \tilde{B}(k, n \Delta t) \) as well as:

\[
\begin{align*}
C &= \cos(ck \Delta t) \\
S &= \sin(ck \Delta t) \\
k &= |k| \\
\nu &= \frac{k \cdot \mathbf{v}_{gal}}{ck} \\
\theta &= e^{i k \cdot \mathbf{v}_{gal} \Delta t / 2} \\
\theta^* &= e^{-i k \cdot \mathbf{v}_{gal} \Delta t / 2} \\
\chi_1 &= \frac{1}{1 - \nu^2} \left( \theta^* - C \theta + i \nu \theta S \right) \\
\chi_2 &= \frac{\chi_1 - \theta(1 - C)}{\theta^* - \theta} \\
\chi_3 &= \frac{\chi_1 - \theta^*(1 - C)}{\theta^* - \theta}
\end{align*}
\]

Note that, in the limit \( \mathbf{v}_{gal} = 0 \), \((\text{eq:disc-maxwell1})\) and \((\text{eq:disc-maxwell2})\) reduce to the standard PSATD equations (Haber et al. 1973), as expected. As shown in (Kirchen et al. 2016; Lehe et al. 2016), the elimination of the NCI with the new Galilean integration is verified empirically via PIC simulations of uniform drifting plasmas and laser-driven plasma acceleration stages, and confirmed by a theoretical analysis of the instability.


Sironi, L, and A Spitkovsky. 2011. “No Title.”


5.6. Moving window and optimal Lorentz boosted frame

183
5.7 Inputs and Outputs

Initialization of the plasma columns and drivers (laser or particle beam) is performed via the specification of multidimensional functions that describe the initial state with, if needed, a time dependence, or from reconstruction of distributions based on experimental data. Care is needed when initializing quantities in parallel to avoid double counting and ensure smoothness of the distributions at the interface of computational domains. When the sum of the initial distributions of charged particles is not charge neutral, initial fields are computed using generally a static approximation with Poisson solves accompanied by proper relativistic scalings (Vay 2008; Cowan et al. 2013).

Outputs include dumps of particle and field quantities at regular intervals, histories of particle distributions moments, spectra, etc, and plots of the various quantities. In parallel simulations, the diagnostic subroutines need to handle additional complexity from the domain decomposition, as well as large amount of data that may necessitate data reduction in some form before saving to disk.

Simulations in a Lorentz boosted frame require additional considerations, as described below.

5.7.1 Inputs and outputs in a boosted frame simulation

The input and output data are often known from, or compared to, experimental data. Thus, calculating in a frame other than the laboratory entails transformations of the data between the calculation frame and the laboratory frame. This section describes the procedures that have been implemented in the Particle-In-Cell framework Warp (Grote et al. 2005) to handle the input and output of data between the frame of calculation and the laboratory frame (Vay et al. 2011). Simultaneity of events between two frames is valid only for a plane that is perpendicular to the relative motion of the frame. As a result, the input/output processes involve the input of data (particles or fields) through a plane, as well as output through a series of planes, all of which are perpendicular to the direction of the relative velocity between the frame of calculation and the other frame of choice.
Fig. 5.8: (top) Snapshot of a particle beam showing “frozen” (grey spheres) and “active” (colored spheres) macroparticles traversing the injection plane (red rectangle). (bottom) Snapshot of the beam macroparticles (colored spheres) passing through the background of electrons (dark brown streamlines) and the diagnostic stations (red rectangles). The electrons, the injection plane and the diagnostic stations are fixed in the laboratory plane, and are thus counter-propagating to the beam in a boosted frame.

Input in a boosted frame simulation

Particles

Particles are launched through a plane using a technique that is generic and applies to Lorentz boosted frame simulations in general, including plasma acceleration, and is illustrated using the case of a positively charged particle beam propagating through a background of cold electrons in an assumed continuous transverse focusing system, leading to a well-known growing transverse “electron cloud” instability (Vay 2007). In the laboratory frame, the electron background is initially at rest and a moving window is used to follow the beam progression. Traditionally, the beam macroparticles are initialized all at once in the window, while background electron macroparticles are created continuously in front of the beam on a plane that is perpendicular to the beam velocity. In a frame moving at some fraction of the beam velocity in the laboratory frame, the beam initial conditions at a given time in the calculation frame are generally unknown and one must initialize the beam differently. However, it can be taken advantage of the fact that the beam initial conditions are often known for a given plane in the laboratory, either directly, or via simple calculation or projection from the conditions at a given time in the laboratory frame. Given the position and velocity \( \{x, y, z, v_x, v_y, v_z\} \) for each beam macroparticle at time \( t = 0 \) for a beam moving at the average velocity \( v_b = \beta_b c \) (where \( c \) is the speed of light) in the laboratory, and using the standard synchronization \( (z = z' = 0 \text{ at } t = t' = 0) \) between the laboratory and the calculation frames, the procedure for transforming the beam quantities for injection in a boosted frame moving at velocity \( \beta c \) in the laboratory is as follows (the superscript ‘ relates to quantities known in the boosted frame while the superscript * relates to quantities that are known at a given longitudinal position \( z^* \) but different times of arrival):

1. project positions at \( z^* = 0 \) assuming ballistic propagation

\[
\begin{align*}
  t^* &= \frac{(z - \bar{z})}{v_z} \\
  x^* &= x - v_x t^* \\
  y^* &= y - v_y t^* \\
  z^* &= 0
\end{align*}
\]
the velocity components being left unchanged,

2. apply Lorentz transformation from laboratory frame to boosted frame

\[
\begin{align*}
t' &= -\gamma t^* \\
x' &= x^* \\
y' &= y^* \\
z' &= \gamma \beta c t^* \\
v_{x}' &= \frac{v_x^*}{\gamma (1 - \beta \beta_b)} \\
v_{y}' &= \frac{v_y^*}{\gamma (1 - \beta \beta_b)} \\
v_{z}' &= \frac{v_z^* - \beta c}{1 - \beta \beta_b}
\end{align*}
\]

where \( \gamma = \frac{1}{\sqrt{1 - \beta^2}} \). With the knowledge of the time at which each beam macroparticle crosses the plane into consideration, one can inject each beam macroparticle in the simulation at the appropriate location and time.

3. synchronize macroparticles in boosted frame, obtaining their positions at a fixed \( t' = 0 \) (before any particle is injected)

\[
z' = z'^* - \bar{v}_z t'^*
\]

This additional step is needed for setting the electrostatic or electromagnetic fields at the plane of injection. In a Particle-In-Cell code, the three-dimensional fields are calculated by solving the Maxwell equations (or static approximation like Poisson, Darwin or other (Vay 2008)) on a grid on which the source term is obtained from the macroparticles distribution. This requires generation of a three-dimensional representation of the beam distribution of macroparticles at a given time before they cross the injection plane at \( z'^* \). This is accomplished by expanding the beam distribution longitudinally such that all macroparticles (so far known at different times of arrival at the injection plane) are synchronized to the same time in the boosted frame. To keep the beam shape constant, the particles are “frozen” until they cross that plane: the three velocity components and the two position components perpendicular to the boosted frame velocity are kept constant, while the remaining position component is advanced at the average beam velocity. As particles cross the plane of injection, they become regular “active” particles with full 6-D dynamics.

Figure [Fig_inputoutput] (top) shows a snapshot of a beam that has passed partly through the injection plane. As the frozen beam macroparticles pass through the injection plane (which moves opposite to the beam in the boosted frame), they are converted to “active” macroparticles. The charge or current density is accumulated from the active and the frozen particles, thus ensuring that the fields at the plane of injection are consistent.

**Laser**

Similarly to the particle beam, the laser is injected through a plane perpendicular to the axis of propagation of the laser (by default \( z \)). The electric field \( E_\perp \) that is to be emitted is given by the formula

\[
E_\perp (x, y, t) = E_0 f (x, y, t) \sin [\omega t + \phi (x, y, \omega)]
\]

where \( E_0 \) is the amplitude of the laser electric field, \( f (x, y, t) \) is the laser envelope, \( \omega \) is the laser frequency, \( \phi (x, y, \omega) \) is a phase function to account for focusing, defocusing or injection at an angle, and \( t \) is time. By default, the laser envelope is a three-dimensional gaussian of the form

\[
f (x, y, t) = e^{-\left( \frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} + \frac{c^2 t^2}{2\sigma_z^2} \right)}
\]

where \( \sigma_x, \sigma_y \) and \( \sigma_z \) are the dimensions of the laser pulse; or it can be defined arbitrarily by the user at runtime. If \( \phi (x, y, \omega) = 0 \), the laser is injected at a waist and parallel to the axis \( z \).
If, for convenience, the injection plane is moving at constant velocity $\beta_s c$, the formula is modified to take the Doppler effect on frequency and amplitude into account and becomes

$$ E_\perp(x, y, t) = (1 - \beta_s) E_0 f(x, y, t) \times \sin[(1 - \beta_s) \omega t + \phi(x, y, \omega)]. $$

The injection of a laser of frequency $\omega$ is considered for a simulation using a boosted frame moving at $\beta c$ with respect to the laboratory. Assuming that the laser is injected at a plane that is fixed in the laboratory, and thus moving at $\beta_s = -\beta$ in the boosted frame, the injection in the boosted frame is given by

$$ E_\perp(x', y', t') = (1 - \beta_s) E'_0 f(x', y', t') \times \sin[(1 - \beta_s) \omega' t' + \phi(x', y', \omega')]. $$

$$ = (E_0/\gamma) f(x', y', t') \times \sin[\omega t'/\gamma + \phi(x', y', \omega')]. $$

since $E'_0/E_0 = \omega'/\omega = 1/(1 + \beta) \gamma$.

The electric field is then converted into currents that get injected via a 2D array of macro-particles, with one positive and one dual negative macro-particle for each array cell in the plane of injection, whose weights and motion are governed by $E_\perp(x', y', t')$. Injecting using this dual array of macroparticles offers the advantage of automatically including the longitudinal component that arises from emitting into a boosted frame, and to automatically verify the discrete Gauss' law thanks to using charge conserving (e.g. Esirkepov) current deposition scheme (Esirkepov 2001).

Some quantities, e.g. charge or dimensions perpendicular to the boost velocity, are Lorentz invariant. Those quantities are thus readily available from standard diagnostics in the boosted frame calculations. Quantities that do not fall in this category are recorded at a number of regularly spaced “stations”, immobile in the laboratory frame, at a succession of time intervals to record data history, or averaged over time. A visual example is given on Fig. [Fig_inputoutput] (bottom). Since the space-time locations of the diagnostic grids in the laboratory frame generally do not coincide with the space-time positions of the macroparticles and grid nodes used for the calculation in a boosted frame, some interpolation is performed at runtime during the data collection process. As a complement or an alternative, selected particle or field quantities can be dumped at regular intervals and quantities are reconstructed in the laboratory frame during a post-processing phase. The choice of the methods depends on the requirements of the diagnostics and particular implementations.


6.1 Contribute to WarpX

We welcome new contributors! Here is how to participate to the WarpX development.

6.1.1 Git workflow

The WarpX project uses git for version control. If you are new to git, you can follow one of these tutorials:

• Learn git with bitbucket
• git - the simple guide

Configure your GitHub Account & Development Machine

First, let’s setup your Git environment and GitHub account.

1. Go to https://github.com/settings/profile and add your real name and affiliation
2. Go to https://github.com/settings/emails and add & verify the professional e-mails you want to be associated with.
3. Configure git on the machine you develop on to use the same spelling of your name and email:
   • git config --global user.name "FIRSTNAME LASTNAME"
   • git config --global user.email EMAIL@EXAMPLE.com
4. Go to https://github.com/settings/keys and add the SSH public key of the machine you develop on. (Check out the GitHub guide to generating SSH keys or troubleshoot common SSH problems.)

Make your own fork

First, fork the WarpX “mainline” repo on GitHub by pressing the Fork button on the top right of the page. A fork is a copy of WarpX on GitHub, which is under your full control.

Then, we create local copies, for development:

```
# Clone the mainline WarpX source code to your local computer.
# You cannot write to this repository, but you can read from it.
git clone git@github.com:ECP-WarpX/WarpX.git
cd WarpX
```

(continues on next page)
# rename what we just cloned: call it "mainline"
```bash
git remote rename origin mainline
```

# Add your own fork. You can get this address on your fork’s Github page.
# Here is where you will publish new developments, so that they can be
# reviewed and integrated into “mainline” later on.
# "myGithubUsername" needs to be replaced with your user name on GitHub.
```bash
git remote add myGithubUsername git@github.com:myGithubUsername/WarpX.git
```

Now you are free to play with your fork (for additional information, you can visit the Github fork help page).

---

**Note:** We only need to do the above steps for the first time.

---

**Let’s**

You are all set! Now, the basic WarpX development workflow is:

1. Implement your changes and push them on a new branch `branch_name` on your fork.

2. Create a Pull Request from branch `branch_name` on your fork to branch `development` on the main WarpX repo.

   Create a branch `branch_name` (the branch name should reflect the piece of code you want to add, like `fix-spectral-solver`) with

   ```bash
   # start from an up-to-date development branch
   git checkout development
   git pull mainline development
   
   # create a fresh branch
   git checkout -b branch_name
   ```

   and do the coding you want.

   It is probably a good time to look at the AMReX documentation and at the Doxygen reference pages:

   - AMReX Doxygen: [https://amrex-codes.github.io/amrex/doxygen](https://amrex-codes.github.io/amrex/doxygen)
   - PICSAR Doxygen: (todo)

   Once you are done developing, add the files you created and/or modified to the git staging area with

   ```bash
   git add <file_I_created> <and_file_I_modified>
   ```
Build your changes

If you changed C++ files, then now is a good time to test those changes by compiling WarpX locally. Follow the developer instructions in our manual to set up a local development environment, then compile and run WarpX.

Commit & push your changes

Periodically commit your changes with

```bash
git commit
```

The commit message (between quotation marks) is super important in order to follow the developments during code-review and identify bugs. A typical format is:

**This is a short, 40-character title**

After a newline, you can write arbitrary paragraphs. You usually limit the lines to 70 characters, but if you don't, then nothing bad will happen.

The most important part is really that you find a descriptive title and add an empty newline after it.

For the moment, commits are on your local repo only. You can push them to your fork with

```bash
git push -u myGithubUsername branch_name
```

If you want to synchronize your branch with the development branch (this is useful when the development branch is being modified while you are working on branch name), you can use

```bash
git pull mainline development
```

and fix any conflict that may occur.

Submit a Pull Request

A Pull Request (PR) is the way to efficiently visualize the changes you made and to propose your new feature/improvement/fix to the WarpX project. Right after you push changes, a banner should appear on the Github page of your fork, with your `<branch_name>`.

• Click on the compare & pull request button to prepare your PR.

• It is time to communicate your changes: write a title and a description for your PR. People who review your PR are happy to know

  – what feature/fix you propose, and why
  – how you made it (added new/edited files, created a new class than inherits from…)
  – how you tested it and what was the output you got
  – and anything else relevant to your PR (attach images and scripts, link papers, etc.)

• Press Create pull request. Now you can navigate through your PR, which highlights the changes you made.

6.1. Contribute to WarpX 191
Please DO NOT write large pull requests, as they are very difficult and time-consuming to review. As much as possible, split them into small, targeted PRs. For example, if find typos in the documentation open a pull request that only fixes typos. If you want to fix a bug, make a small pull request that only fixes a bug.

If you want to implement a feature and are not too sure how to split it, just open an issue about your plans and ping other WarpX developers on it to chime in. Generally, write helper functionality first, test it and then write implementation code. Submit tests, documentation changes and implementation of a feature together for pull request review.

Even before your work is ready to merge, it can be convenient to create a PR (so you can use Github tools to visualize your changes). In this case, please put the [WIP] tag (for Work-In-Progress) at the beginning of the PR title. You can also use the GitHub project tab in your fork to organize the work into separate tasks/PRs and share it with the WarpX community to get feedback.

Include a test to your PR

A new feature is great, a working new feature is even better! Please test your code and add your test to the automated test suite. It’s the way to protect your work from adventurous developers. Instructions are given in the testing section of our developer’s documentation.

Include documentation about your PR

Now, let users know about your new feature by describing its usage in the WarpX documentation. Our documentation uses Sphinx, and it is located in Docs/source/. For instance, if you introduce a new runtime parameter in the input file, you can add it to Docs/source/running_cpp/parameters.rst. If Sphinx is installed on your computer, you should be able to generate the html documentation with

```
make html
```

in Docs/. Then open Docs/build/html/index.html with your favorite web browser and look for your changes. Once your code is ready with documentation and automated test, congratulations! You can create the PR (or remove the [WIP] tag if you already created it). Reviewers will interact with you if they have comments/questions.

6.1.2 Style and conventions

- For indentation, WarpX uses four spaces (no tabs)
- Some text editors automatically modify the files you open. We recommend to turn on to remove trailing spaces and replace Tabs with 4 spaces.
- The number of characters per line should be <100
- Exception: in documentation files (.rst/.md) use one sentence per line independent of its number of characters, which will allow easier edits.
- Space before and after assignment operator (=)
- To define a function, for e.g., myfunction() use a space between the name of the function and the paranthesis - myfunction(). To call the function, the space is not required, i.e., just use myfunction().
- The reason this is beneficial is that when we do a git grep to search for myfunction(), we can clearly see the locations where myfunction() is defined and where myfunction() is called.
• Also, using `git grep "myfunction ()"` searches for files only in the git repo, which is more efficient compared to the `grep "myfunction ()"` command that searches through all the files in a directory, including plotfiles for example.

• It is recommended that style changes are not included in the PR where new code is added. This is to avoid any errors that may be introduced in a PR just to do style change.

• WarpX uses CamelCase convention for file names and class names, rather than snake_case.

• The names of all member variables should be prefixed with `m_`. This is particularly useful to avoid capturing member variables by value in a lambda function, which causes the whole object to be copied to GPU when running on a GPU-accelerated architecture. This convention should be used for all new piece of code, and it should be applied progressively to old code.

• `#include` directives in C++ have a distinct order to avoid bugs, see the WarpX repo structure for details

• For all new code, we should avoid relying on using namespace amrex; and all amrex types should be prefixed with `amrex::`. Inside limited scopes, AMReX type literals can be included with using namespace `amrex::literals;`. Ideally, old code should be modified accordingly.

6.2 Workflows

6.2.1 Profiling

Tiny  

The so-called “Tiny Profiler” is a component of lightweight profiling that we perform with each WarpX run by default. A table of the summary of most often called and most costly functions is printed to stdout at the end of every WarpX run.

See the full documentation in AMReX.

There is a script located here that parses the Tiny Profiler output and generates a JSON file that can be used with Hatchet in order to analyze performance.

Nvidia  

Vendor homepage and product manual.

Example on how to create traces on a multi-GPU system that uses the Slurm scheduler:

```
rm -rf profiling*

# adjust if needed
#export GPUS_PER_SOCKET=2
#export GPUS_PER_NODE=4
export OMP_NUM_THREADS=1

# record
srun --ntasks=4 --gpus=4 --cpu-bind=cores \
    nsys profile -f true \
    -o profiling_%q{SLURM_TASK_PID} \
    -t mpi,cuda,nvtx,osrt,openmp \
    --mpi-impl=openmpi \
```

(continues on next page)
In this example, the individual lines for recording a trace profile are:

- `srun`: execute multi-GPU runs with `srun` (Slurm’s `mpiexec` wrapper), here for four GPUs
- `-f true`: overwrite previously written trace profiles
- `-o`: record one profile file per MPI rank (per GPU); if you run `mpiexec/mpirun` with OpenMPI directly, replace `SLURM_TASK_PID` with `OMPI_COMM_WORLD_RANK`
- `-t`: select a couple of APIs to trace
- `--mpi--impl`: optional, hint the MPI flavor
- `./warpx...`: select the WarpX executable and a good inputs file
- `warpx.numprocs=...`: make the run short, reasonably small, and run only a few steps

Now open the created trace files (per rank) in the Nsight-Systems GUI. This can be done on another system than the one that recorded the traces. For example, if you record on a cluster and open the analysis GUI on your laptop, it is recommended to make sure that versions of Nsight-Systems match on the remote and local system.

### 6.2.2 Testing the code

When adding a new feature, you want to make sure that (i) you did not break the existing code and (ii) your contribution gives correct results. While existing capabilities are tested regularly remotely (when commits are pushed to an open PR on CI, and every night on local clusters), it can also be useful to run tests on your custom input file. This section details how to use both automated and custom tests.

#### Continuous Integration in WarpX

Our regression tests are using the suite published and documented at [AMReX-Codes/regression_testing](https://github.com/AMReX-Codes/regression_testing). Most of the configuration of our regression tests happens in `Regression/Warpx-tests.ini`. We slightly modify this file in `Regression/prepare_file_ci.py`.

For example, if you like to change the compiler to compilation to build on Nvidia GPUs, modify this block to add `--DWarpx_COMPUTE=CUDA`:

```ini
[source]
dir = /home/regtester/AMReX_RegTesting/warpx
branch = development
cmakeSetupOpts = -DAMReX_ASSERTIONS=ON -DAMReX_TESTING=ON -DWarpX_COMPUTE=CUDA
```

We also support changing compilation options: `ref:via the usual build environment variables` `<building-cmake-envvars:>`

```bash
export CXX=$(which clang++)
export CXXFLAGS="-Werror"
```
Once your new feature is ready, there are ways to check that you did not break anything. WarpX has automated tests running every time a commit is added to an open pull request. The list of automated tests is defined in `/Regression/WarpX-tests.ini`.

For easier debugging, it can be convenient to run the tests on your local machine by executing the script `/run_test.sh` from WarpX’s root folder, as illustrated in the examples below:

```
# Example:
# run all tests defined in ./Regression/WarpX-tests.ini
./run_test.sh

# Example:
# run only the test named 'pml_x_yee'
./run_test.sh pml_x_yee

# Example:
# run only the tests named 'pml_x_yee', 'pml_x_ckc' and 'pml_x_psatd'
./run_test.sh pml_x_yee pml_x_ckc pml_x_psatd
```

Note that the script `/run_test.sh` runs the tests with the exact same compile-time options and runtime options used to run the tests remotely.

Moreover, the script `/run_test.sh` compiles all the executables that are necessary in order to run the chosen tests. The default number of threads allotted for compiling is set with `numMakeJobs = 8` in `/Regression/WarpX-tests.ini`. However, when running the tests on a local machine, it is usually possible and convenient to allot more threads for compiling, in order to speed up the builds. This can be accomplished by setting the environment variable `WARPX_CI_NUM_MAKE_JOBS`, with the preferred number of threads that fits your local machine, e.g. `export WARPX_CI_NUM_MAKE_JOBS=16` (or less if your machine is smaller). On public CI, we overwrite the value to `WARPX_CI_NUM_MAKE_JOBS=2`, in order to avoid overloading the available remote resources. Note that this will not change the number of threads used to run each test, but only the number of threads used to compile each executable necessary to run the tests.

Once the execution of `/run_test.sh` is completed, you can find all the relevant files associated with each test in one single directory. For example, if you run the single test `pml_x_yee`, as shown above, on 04/30/2021, you can find all relevant files in the directory `./test_dir/rt-WarpX/WarpX-tests/2021-04-30/pml_x_yee/`. The content of this directory will look like the following (possibly including backtraces if the test crashed at runtime):

```
$ ls ./test_dir/rt-WarpX/WarpX-tests/2021-04-30/pml_x_yee/
analysis_pml_yee.py  # Python analysis script
inputs_2d            # input file
main2d.gnu.TEST.TPROF.MT MPI.OMP.QED.GPUCLOCK.ex  # executable
pml_x_yee.analysis.out # Python analysis output
pml_x_yee.err.out     # error output
pml_x_yee.make.out    # build output
pml_x_yee.plt00000/   # data output (initialization)
pml_x_yee.plt00300/   # data output (last time step)
pml_x_yee.run.out     # test output
```

6.2. Workflows
Add a test to the suite

There are three steps to follow to add a new automated test (illustrated here for PML boundary conditions):

- An input file for your test, in folder `Example/Tests/`. For the PML test, the input file is at `Examples/Tests/PML/inputs_2d`. You can also re-use an existing input file (even better!) and pass specific parameters at runtime (see below).

- A Python script that reads simulation output and tests correctness versus theory or calibrated results. For the PML test, see `Examples/Tests/PML/analysis_pml_yee.py`. It typically ends with Python statement `assert(error<0.01)`.

- If you need a new Python package dependency for testing, add it in `Regression/requirements.txt`.

- Add an entry to `Regression/WarpX-tests.ini`, so that a WarpX simulation runs your test in the continuous integration process, and the Python script is executed to assess the correctness. For the PML test, the entry is

```
[pml_x_yee]
buildDir = .
inputFile = Examples/Tests/PML/inputs2d
runtime_params = warpx.do_dynamic_scheduling=0 algo.maxwell_solver=yee
dim = 2
addToCompileString =
cmakeSetupOpts = -DWarpX_DIMS=2
restartTest = 0
useMPI = 1
numprocs = 2
useOMP = 1
numthreads = 1
compileTest = 0
doVis = 0
analysisRoutine = Examples/Tests/PML/analysis_pml_yee.py
```

If you re-use an existing input file, you can add arguments to `runtime_params`, like `runtime_params = amr.max_level=1 amr.n_cell=32 512 max_step=100 plasma_e.zmin=-200.e-6`.

**Note:** If you added `analysisRoutine = Examples/analysis_default_regression.py`, then run the new test case locally and add the `checksum` file for the expected output.

**Note:** We run those tests on our continuous integration services, which at the moment only have 2 virtual CPU cores. Thus, make sure that the product of `numprocs` and `numthreads` for a test is <=2.

Useful tool for plotfile comparison: `fcompare`

AMReX provides `fcompare`, an executable that takes two plotfiles as input and returns the absolute and relative difference for each field between these two plotfiles. For some changes in the code, it is very convenient to run the same input file with an old and your current version, and `fcompare` the plotfiles at the same iteration. To use it:

```
# Compile the executable
cd <path to AMReX>/Tools/Plotfile/ # This may change
make -j 8
```
# Run the executable to compare old and new versions

```bash
<path to AMReX>/Tools/Plotfile/fcompare.gnu.ex old/plt00200 new/plt00200
```

which should return something like

| variable name | absolute error \( (||A - B||) \) | relative error \( (||A - B||/||A||) \) |
|---------------|---------------------------------|---------------------------------|
| `level = 0`   |                                 |                                 |
| `jx`          | \( 1.044455105e+11 \)          | \( 1.021651316 \)             |
| `jy`          | \( 4.08631977e+16 \)           | \( 7.73499273 \)              |
| `jz`          | \( 1.877301764e+14 \)          | \( 1.073458933 \)             |
| `Ex`          | \( 4.196315448e+10 \)          | \( 1.235551615 \)             |
| `Ey`          | \( 3.330698083e+12 \)          | \( 6.436470137 \)             |
| `Ez`          | \( 2.598167798e+10 \)          | \( 0.6804387128 \)            |
| `Bx`          | \( 273.8687473 \)              | \( 2.340209782 \)             |
| `By`          | \( 152.3911863 \)              | \( 1.0952567 \)               |
| `Bz`          | \( 37.43212767 \)              | \( 2.1977289 \)               |
| `part_per_cell` | \( 15 \)                    | \( 0.9375 \)                  |
| `Ex_fp`       | \( 4.196315448e+10 \)          | \( 1.235551615 \)             |
| `Ey_fp`       | \( 3.330698083e+12 \)          | \( 6.436470137 \)             |
| `Ez_fp`       | \( 2.598167798e+10 \)          | \( 0.6804387128 \)            |
| `Bx_fp`       | \( 273.8687473 \)              | \( 2.340209782 \)             |
| `By_fp`       | \( 152.3911863 \)              | \( 1.0952567 \)               |
| `Bz_fp`       | \( 37.43212767 \)              | \( 2.1977289 \)               |

---

### 6.2.3 Documentation

**Doxygen documentation**

WarpX uses a Doxygen documentation. Whenever you create a new class, please document it where it is declared (typically in the header file):

```cpp
/** A brief title
 * few-line description explaining the purpose of my_class.
 * If you are kind enough, also quickly explain how things in my_class work.
 * (typically a few more lines)
 */

class my_class
{
  ...
}
```

Doxygen reads this docstring, so please be accurate with the syntax! See Doxygen manual for more information.

Similarly, please document functions when you declare them (typically in a header file) like:

```cpp
/** A brief title
 * few-line description explaining the purpose of my_function.
 * \param[in,out] my_int a pointer to an integer variable on which
 */
```

(continues on next page)
An online version of this documentation is linked here.

Breathe documentation

Your Doxygen documentation is not only useful for people looking into the code, it is also part of the WarpX online documentation based on Sphinx! This is done using the Python module Breathe, that allows you to read Doxygen documentation directly in the source and include it in your Sphinx documentation, by calling Breathe functions. For instance, the following line will get the Doxygen documentation for WarpXPathParticleContainer in Source/Particles/WarpXPathParticleContainer.H and include it to the html page generated by Sphinx:

```
 class WarpXPathParticleContainer : public amrex::ParticleContainer<0, 0, Plidx::nattrs>
 WarpXPathParticleContainer is the base polymorphic class from which all concrete particle container classes (that store a collection of particles) derive. Derived classes can be used for plasma particles, photon particles, or non-physical particles (e.g., for the laser antenna). It derives from amrex::ParticleContainer<0,0,Plidx::nattrs>, where the template arguments stand for the number of int and amrex::Real SoA and AoS data in amrex::Particle.

• AoS amrex::Real: x, y, z (default), 0 additional (first template parameter)
• AoS int: id, cpu (default), 0 additional (second template parameter)
• SoA amrex::Real: Plidx::nattrs (third template parameter), see Plidx for the list.

WarpXPathParticleContainer contains the main functions for initialization, interaction with the grid (field gather and current deposition) and particle push.

Note: many functions are pure virtual (meaning they MUST be defined in derived classes, e.g., Evolve) or actual functions (e.g. CurrentDeposition).

Subclassed by LaserParticleContainer, PhysicalParticleContainer
```

Building the documentation

To build the documentation on your local computer, you will need to install Doxygen as well as the Python module breathe. First, change into Docs/ and install the Python requirements:

```
  cd Docs/
  python3 -m pip install -r requirements.txt
```

You will also need Doxygen (macOS: brew install doxygen; Ubuntu: sudo apt install doxygen).

Then, to compile the documentation, use

```
  make html
  # This will first compile the Doxygen documentation (execute doxygen)
  # and then build html pages from rst files using sphinx and breathe.
```

Open the created build/html/index.html file with your favorite browser. Rebuild and refresh as needed.
6.2.4 Checksum regression tests

WarpX has checksum regression tests: as part of CI testing, when running a given test, the checksum module computes one aggregated number per field (\( \text{Ex\_checksum} = \text{np\_sum}(\text{np\_abs(Ex)}) \)) and compares it to a reference (benchmark). This should be sensitive enough to make the test fail if your PR causes a significant difference, print meaningful error messages, and give you a chance to fix a bug or reset the benchmark if needed.

The checksum module is located in Regression/Checksum/, and the benchmarks are stored as human-readable JSON files in Regression/Checksum/benchmarks_json/, with one file per benchmark (for instance, test Langmuir_2d has a corresponding benchmark Regression/Checksum/benchmarks_json/Langmuir_2d.json).

For more details on the implementation, the Python files in Regression/Checksum/ should be well documented.

From a user point of view, you should only need to use checksumAPI.py. It contains Python functions that can be imported and used from an analysis Python script. It can also be executed directly as a Python script. Here are recipes for the main tasks related to checksum regression tests in WarpX CI.

**Include a checksum regression test in an analysis Python script**

This relies on function `evaluate_checksum`:

```python
Checksum.checksumAPI.evaluate_checksum(test_name, plotfile, rtol=1e-09, atol=1e-40, do_fields=True, do_particles=True)
```

- Compare plotfile checksum with benchmark.
- Read checksum from input plotfile, read benchmark corresponding to `test_name`, and assert their equality.

@param `test_name` Name of test, as found between [] in .ini file.
@param `plotfile` Plotfile from which the checksum is computed.
@param `rtol` Relative tolerance for the comparison.
@param `atol` Absolute tolerance for the comparison.
@param `do_fields` Whether to compare fields in the checksum.
@param `do_particles` Whether to compare particles in the checksum.

For an example, see

```bash
#!/usr/bin/env python3
import os
import re
import sys
import checksumAPI

# this will be the name of the plot file
fn = sys.argv[1]

# Get name of the test
test_name = os.path.split(os.getcwd())[1]

# Run checksum regression test
```
if re.search('single_precision', fn):
    checksumAPI.evaluate_checksum(test_name, fn, rtol=2.e-6)
else:
    checksumAPI.evaluate_checksum(test_name, fn)

This can also be included in an existing analysis script. Note that the plotfile must be <test name>_plt?????, as is generated by the CI framework.

Evaluate a checksum regression test from a bash terminal

You can execute checksumAPI.py as a Python script for that, and pass the plotfile that you want to evaluate, as well as the test name (so the script knows which benchmark to compare it to).

```
./checksumAPI.py --evaluate --plotfile <path/to/plotfile> --test-name <test name>
```

See additional options

- `--skip-fields` if you don’t want the fields to be compared (in that case, the benchmark must not have fields)
- `--skip-particles` same thing for particles
- `--rtol` relative tolerance for the comparison
- `--atol` absolute tolerance for the comparison (a sum of both is used by numpy.isclose())

Reset a benchmark from a plotfile you know is correct

This is using checksumAPI.py as a Python script.

```
./checksumAPI.py --reset-benchmark --plotfile <path/to/plotfile> --test-name <test name>
```

See additional options

- `--skip-fields` if you don’t want the benchmark to have fields
- `--skip-particles` same thing for particles

Since this will automatically change the JSON file stored on the repo, make a separate commit just for this file, and if possible commit it under the Tools name:

```
git add <test name>.json
git commit -m "reset benchmark for <test name> because ..." --author="Tools <warpx@lbl.gov>"
```

6.3 Implementation

6.3.1 AMReX basics (excessively basic)

WarpX is built on the Adaptive Mesh Refinement (AMR) library AMReX. This section provides a very sporadic description of the main AMReX classes and concepts relevant for WarpX, that can serve as a reminder. Please read the AMReX basics doc page, of which this section is largely inspired.

- `amrex::Box`: Dimension-dependent lower and upper indices defining a rectangular volume in 3D (or surface in 2D) in the index space. Box is a lightweight meta-data class, with useful member functions.
- **amrex::BoxArray**: Collection of `Box` on a single AMR level. The information of which MPI rank owns which `Box` in a `BoxArray` is in `DistributionMapping`.

- **amrex::FArrayBox**: Fortran-ordered array of floating-point `amrex::Real` elements defined on a `Box`. A `FArrayBox` can represent scalar data or vector data, with `ncomp` components.

- **amrex::MultiFab**: Collection of `FAB` (= `FArrayBox`) on a single AMR level, distributed over MPI ranks. The concept of *ghost cells* is defined at the `MultiFab` level.

- **amrex::ParticleContainer**: A collection of particles, typically for particles of a physical species. Particles in a `ParticleContainer` are organized per `Box`. Particles in a `Box` are organized per tile (this feature is off when running on GPU). Particles within a tile are stored in several structures, each being contiguous in memory: (i) an Array-Of-Struct (AoS) (often called *data*, they are the 3D position, the particle ID and the index of the CPU owning the particle), where the Struct is an `amrex::Particle` and (ii) Struct-Of-Arrays (SoA) for extra variables (often called *attribs*, in WarpX they are the momentum, field on particle etc.).

The simulation domain is decomposed in several `Box`, and each MPI rank owns (and performs operations on) the fields and particles defined on a few of these `Box`, but has the metadata of all of them. For convenience, AMReX provides iterators, to easily iterate over all `FArrayBox` (or even tile-by-tile, optionally) in a `MultiFab` own by the MPI rank (`MFIter`), or over all particles in a `ParticleContainer` on a per-box basis (`ParIter`, or its derived class `WarpXParIter`). These are respectively done in loops like:

```cpp
// mf is a pointer to MultiFab
for ( amrex::MFIter mfi(mf, false); mfi.isValid(); ++mfi ) { ... }
```

and

```cpp
// *this is a pointer to a ParticleContainer
for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti) { ... }
```

When looping over `FArrayBox` in a `MultiFab`, the iterator provides functions to retrieve the metadata of the `Box` on which the `FAB` is defined (`MFIter::box()`, `MFIter::tilebox()` or variations) or the particles defined on this `Box` (`ParIter::GetParticles()`).

### 6.3.2 WarpX Structure

**Repo**

All the WarpX source code is located in `Source/`. All sub-directories have a pretty straightforward name. The PIC loop is part of the WarpX class, in function `WarpX::EvolveEM` implemented in `Source/WarpXEvolveEM.cpp`. The core of the PIC loop (i.e., without diagnostics etc.) is in `WarpX::OneStep_nosub` (when subcycling is OFF) or `WarpX::OneStep_sub1` (when subcycling is ON, with method 1). Here is a visual representation of the repository structure.

**Code organization**

The main WarpX class is WarpX, implemented in `Source/WarpX.cpp`. 
Build

WarpX uses the CMake build system generator. Each sub-folder contains a file CMakeLists.txt with the names of the source files (.cpp) that are added to the build. Do not list header files (.h) here.

For experienced developers, we also support AMReX’ GNUmake build script collection. The file Make.package in each sub-folder has the same purpose as the CMakeLists.txt file, please add new .cpp files to both dirs.

C++

All WarpX header files need to be specified relative to the Source/ directory.

- e.g. #include "Utils/WarpXConst.H"
- files in the same directory as the including header-file can be included with #include "FileName.H"

By default, in a MyName.cpp source file we do not include headers already included in MyName.H. Besides this exception, if a function or a class is used in a source file, the header file containing its declaration must be included, unless the inclusion of a facade header is more appropriate. This is sometimes the case for AMReX headers. For instance AMReX_GpuLaunch.H is a facade header for AMReX_GpuLaunchFunctsC.H and AMReX_GpuLaunchFunctsG.H, which contain respectively the CPU and the GPU implementation of some methods, and which should not be included directly. Whenever possible, forward declarations headers are included instead of the actual headers, in order to save compilation time (see dedicated section below). In WarpX forward declaration headers have the suffix *_fwd.H, while in AMReX they have the suffix *Fwd.H. The include order (see PR #874 and PR #1947) and proper quotation marks are:

In a MyName.cpp file:

1. #include "MyName.H" (its header) then
2. (further) WarpX header files #include "...", then
3. WarpX forward declaration header files #include "..._fwd.H"
4. AMReX header files #include <...> then
5. AMReX forward declaration header files #include <...Fwd.H> then
6. PICSAR header files #include <...> then
7. other third party includes #include <...> then
8. standard library includes, e.g. #include <vector>

In a MyName.H file:

1. #include "MyName_fwd.H" (the corresponding forward declaration header, if it exists) then
2. WarpX header files #include "...", then
3. WarpX forward declaration header files #include "..._fwd.H"
4. AMReX header files #include <...> then
5. AMReX forward declaration header files #include <...Fwd.H> then
6. PICSAR header files #include <...> then
7. other third party includes #include <...> then
8. standard library includes, e.g. #include <vector>
Each of these groups of header files should ideally be sorted alphabetically, and a blank line should be placed between the groups.

For details why this is needed, please see PR #874, PR #1947, the LLVM guidelines, and include-what-you-use.

**Forward Declaration Headers**

Forward declarations can be used when a header file needs only to know that a given class exists, without any further detail (e.g., when only a pointer to an instance of that class is used). Forward declaration headers are a convenient way to organize forward declarations. If a forward declaration is needed for a given class `MyClass`, declared in `MyClass.H`, the forward declaration should appear in a header file named `MyClass_fwd.H`, placed in the same folder containing `MyClass.H`. As for regular header files, forward declaration headers must have include guards. Below we provide a simple example:

**MyClass_fwd.H**:

```cpp
#ifndef MY_CLASS_FWD_H
#define MY_CLASS_FWD_H

class MyClass;

#endif //MY_CLASS_FWD_H
```

**MyClass.H**:

```cpp
#ifndef MY_CLASS_H
#define MY_CLASS_H

#include "MyClass_fwd.H"
#include "someHeader.H"

class MyClass /* stuff */;

#endif //MY_CLASS_H
```

**MyClass.cpp**:

```cpp
#include "MyClass.H"

class MyClass /* stuff */;
```

Usage: in `SimpleUsage.H`

```cpp
#include "MyClass_fwd.H"
#include <memory>

/* stuff */
std::unique_ptr<MyClass> p_my_class;
/* stuff */
```
6.3.3 Dimensionality

This section describes the handling of dimensionality in WarpX.

### Build Options

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>CMake Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D3V</td>
<td><code>WarpX_DIMS=3</code> (default)</td>
</tr>
<tr>
<td>2D3V</td>
<td><code>WarpX_DIMS=2</code></td>
</tr>
<tr>
<td>1D3V</td>
<td><code>WarpX_DIMS=1</code></td>
</tr>
<tr>
<td>RZ</td>
<td><code>WarpX_DIMS=RZ</code></td>
</tr>
</tbody>
</table>

See [building from source](#) for further details.

### Defines

Depending on the build variant of WarpX, the following preprocessor macros will be set:

<table>
<thead>
<tr>
<th>Macro</th>
<th>3D3V</th>
<th>2D3V</th>
<th>1D3V</th>
<th>RZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMREX_SPACEDIM</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>WARPX_DIM_3D</td>
<td>defined</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>WARPX_DIM_1D_Z</td>
<td>undefined</td>
<td>undefined</td>
<td>defined</td>
<td>undefined</td>
</tr>
<tr>
<td>WARPX_DIM_XZ</td>
<td>undefined</td>
<td>defined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>WARPX_DIM_RZ</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
<td>defined</td>
</tr>
<tr>
<td>WARPX_ZINDEX</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

At the same time, the following conventions will apply:

<table>
<thead>
<tr>
<th>Convention</th>
<th>3D3V</th>
<th>2D3V</th>
<th>1D3V</th>
<th>RZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMReX Box dimensions</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>WarpX axis labels</td>
<td>x, y, z</td>
<td>x, z</td>
<td>z</td>
<td>x, z</td>
</tr>
<tr>
<td>Particles</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMReX AoS .pos()</td>
<td>0, 1, 2</td>
<td>0, 1</td>
<td>0</td>
<td>0, 1</td>
</tr>
<tr>
<td>WarpX position names</td>
<td>x, y, z</td>
<td>x, z</td>
<td>z</td>
<td>r, z</td>
</tr>
<tr>
<td>extra SoA attribute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please see the following sections for particle AoS and SoA details.

### Conventions

In 2D3V, we assume that the position of a particle in y is equal to 0. In 1D3V, we assume that the position of a particle in x and y is equal to 0.
### 6.3.4 Fields

**Note:** Add info on staggering and domain decomposition. Synchronize with section initialization.

The main fields are the electric field $E_{\text{field}}$, the magnetic field $B_{\text{field}}$, the current density $\text{current}$ and the charge density $\rho$. When a divergence-cleaner is used, we add another field $F$ (containing $\nabla \cdot E - \rho$).

Due the AMR strategy used in WarpX (see section *Theory: AMR* for a complete description), each field on a given refinement level $lev$ (except for the coarsest 0) is defined on:

- **the fine patch** (suffix 
_fpa, the actual resolution on $lev$).
- **the coarse patch** (suffix _cp, same physical domain with the resolution of MR level $lev-1$).
- **the auxiliary grid** (suffix _aux, same resolution as _fp), from which the fields are gathered from the grids to particle positions. For this reason, only $E$ and $B$ are defined on this _aux grid (not the current density or charge density).
- In some conditions, i.e., when buffers are used for the field gather (for numerical reasons), a copy of $E$ and $B$ on the auxiliary grid _aux of the level below $lev-1$ is stored in fields with suffix _cax (for coarse aux).

As an example, the structures for the electric field are $E_{\text{field}}_{\text{fp}}$, $E_{\text{field}}_{\text{cp}}$, $E_{\text{field}}_{\text{aux}}$ (and optionally $E_{\text{field}}_{\text{cax}}$).

#### Declaration

All the fields described above are public members of class *WarpX*, defined in *WarpX.H*. They are defined as an `amrex::Vector` (over MR levels) of `std::array` (for the 3 spatial components $E_x, E_y, E_z$) of `std::unique_ptr` of `amrex::MultiFab`, i.e.:

```cpp
    amrex::Vector< std::array< std::unique_ptr<amrex::MultiFab>, 3 > > Efield_fp;
```

Hence, $E_x$ on MR level $lev$ is a pointer to an `amrex::MultiFab`. The other fields are organized in the same way.

#### Allocation and initialization

The `MultiFab` constructor (for, e.g., $E_x$ on level $lev$) is called in *WarpX::AllocLevelMFs*.

By default, the `MultiFab` are set to 0 at initialization. They can be assigned a different value in *WarpX::InitLevelData*.

#### Field solver

The field solver is performed in *WarpX::EvolveE* for the electric field and *WarpX::EvolveB* for the magnetic field, called from *WarpX::OneStep_nosub* in *WarpX::EvolveEM*. This section describes the FDTD field push. It is implemented in `Source/FieldSolver/FiniteDifferenceSolver/`.

As all cell-wise operation, the field push is done as follows (this is split in multiple functions in the actual implementation to avoid code duplication):

```cpp
    // Loop over MR levels
    for (int lev = 0; lev <= finest_level; ++lev) {
        // Get pointer to MultiFab Ex on level lev
        MultiFab* Ex = Efield_fp[lev][0].get();
        // Loop over boxes (or tiles if not on GPU)
    }
```

(continues on next page)
for ( MFIter mfi(*Ex, TilingIfNotGPU()); mfi.isValid(); ++mfi ) {
    // Apply field solver on the FAB
}

The innermost step // Apply field solver on the FAB could be done with 3 nested for loops for the 3 dimensions (in 3D). However, for portability reasons (see section Developers: Portability), this is done in two steps: (i) extract AMReX data structures into plain-old-data simple structures, and (ii) call a general ParallelFor function (translated into nested loops on CPU or a kernel launch on GPU, for instance):

```
// Get Box corresponding to the current MFIter
const Box& tex = mfi.tilebox(Ex_nodal_flag);
// Extract the FArrayBox into a simple structure, for portability
Array4<Real> const& Exfab = Ex->array(mfi);
// Loop over cells and perform stencil operation
amrex::ParallelFor(tex,
    [=] AMREX_GPU_DEVICE (int j, int k, int l)
    {
        Ex(i, j, k) += c2 * dt * (  
            - T_Algo::DownwardDz(By, coefs_z, n_coefs_z, i, j, k)  
            + T_Algo::DownwardDy(Bz, coefs_y, n_coefs_y, i, j, k)  
            - PhysConst::mu0 * jx(i, j, k) );
    })
```

where T_Algo::DownwardDz and T_Algo::DownwardDy represent the discretized derivative for a given algorithm (represented by the template parameter T_Algo). The available discretization algorithms can be found in Source/FieldSolver/FiniteDifferenceSolver/FiniteDifferenceAlgorithms.

Guard cells exchanges

Communications are mostly handled in Source/Parallelization/.

For E and B guard cell exchanges, the main functions are variants of amrex::FillBoundary(amrex::MultiFab, ...) (or amrex::MultiFab::FillBoundary(...)) that fill guard cells of all amrex::FArrayBox in an amrex::MultiFab with valid cells of corresponding amrex::FArrayBox neighbors of the same amrex::MultiFab. There are a number of FillBoundaryE, FillBoundaryB etc. Under the hood, amrex::FillBoundary calls amrex::ParallelCopy, which is also sometimes directly called in WarpX. Most calls a

For the current density, the valid cells of neighboring MultiFabs are accumulated (added) rather than just copied. This is done using amrex::MultiFab::SumBoundary, and mostly located in Source/Parallelization/WarpXSumGuardCells.H.
### Interpolations for MR

This is mostly implemented in `Source/Parallelization`, see the following functions (you may complain to the authors if the documentation is empty)

```cpp
void WarpX::SyncCurrent()

void WarpX::RestrictCurrentFromFineToCoarsePatch(int lev)
    Fills the values of the current on the coarse patch by averaging the values of the current of the fine patch (on the same level).

void WarpX::AddCurrentFromFineLevelandSumBoundary(int lev)
```

### Filter

General functions for filtering can be found in `Source/Filter/`, where the main `Filter` class is defined (see below). All filters (so far there are two of them) in WarpX derive from this class.

```cpp
class Filter
    Subclassed by BilinearFilter, NCIGodfreyFilter
```

#### Bilinear

The multi-pass bilinear filter (applied on the current density) is implemented in `Source/Filter/`, and class `WarpX` holds an instance of this class in member variable `WarpX::bilinear_filter`. For performance reasons (to avoid creating too many guard cells), this filter is directly applied in communication routines, see `WarpX::AddCurrentFromFineLevelandSumBoundary` above and

```cpp
void WarpX::ApplyFilterandSumBoundaryJ(int lev, PatchType patch_type)
```

### Godfrey’s anti-NCI filter for FDTD simulations

This filter is applied on the electric and magnetic field (on the auxiliary grid) to suppress the Numerical Cherenkov Instability when running FDTD. It is implemented in `Source/Filter/`, and there are two different stencils, one for $E_x$, $E_y$ and $B_z$ and the other for $E_z$, $B_x$ and $B_y$.

```cpp
class NCIGodfreyFilter: public Filter
    Class for Godrey’s filter to suppress Numerical Cherenkov Instability.

    It derives from the base class Filter. The filter stencil is initialized in method ComputeStencils. Computing the stencil requires to read parameters from a table, where each lines stands for a value of $c^*dt/dz$. The filter is applied using the base class’ method ApplyStencil.

    The class `WarpX` holds two corresponding instances of this class in member variables `WarpX::nci_godfrey_filter_exeybz` and `WarpX::nci_godfrey_filter_bxbyez`. It is a 9-point stencil (is the $z$ direction only), for which the coefficients are computed using tabulated values (depending on $dz/dx$) in `Source/Utils/NCIGodfreyTables.H`, see variable `table_nci_godfrey_galerkin_Ex_Ey_Bz`. The filter is applied in `PhysicalParticleContainer::Evolve`, right after field gather and before particle push, see
```

### 6.3. Implementation Details

207
void PhysicalParticleContainer::applyNCIFilter(int lev, const amrex::Box &box, amrex::Elixir &exeli, amrex::Elixir &eyeli, amrex::Elixir &ezeli, amrex::Elixir &bxeli, amrex::Elixir &byeli, amrex::Elixir &bzeli, amrex::FArrayBox &filtered_Ex, amrex::FArrayBox &filtered_Ey, amrex::FArrayBox &filtered_Ez, amrex::FArrayBox &filtered_Bx, amrex::FArrayBox &filtered_By, amrex::FArrayBox &filtered_Bz, const amrex::FArrayBox &Ex, const amrex::FArrayBox &Ey, const amrex::FArrayBox &Ez, const amrex::FArrayBox &Bx, const amrex::FArrayBox &By, const amrex::FArrayBox &Bz, amrex::FArrayBox const *&exfab, amrex::FArrayBox const *&eyfab, amrex::FArrayBox const *&ezfab, amrex::FArrayBox const *&bxfab, amrex::FArrayBox const *&byfab, amrex::FArrayBox const *&bzf fab)

Apply NCI Godfrey filter to all components of E and B before gather.

The NCI Godfrey filter is applied on Ex, the result is stored in filtered_Ex and the pointer exfab is modified (before this function is called, it points to Ex and after this function is called, it points to Ex_filtered)

Parameters

- **lev** – MR level
- **box** – box onto which the filter is applied
- **exeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field Ex
- **eyeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field Ey
- **ezeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field Ez
- **bxeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field Bx
- **byeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field By
- **bzeli** – safeguard Elixir object (to avoid de-allocating too early between ParIter iterations on GPU) for field Bz
- **filtered_Ex** – Array containing filtered value
- **filtered_Ey** – Array containing filtered value
- **filtered_Ez** – Array containing filtered value
- **filtered_Bx** – Array containing filtered value
- **filtered_By** – Array containing filtered value
- **filtered_Bz** – Array containing filtered value
- **Ex** – Field array before filtering (not modified)
- **Ey** – Field array before filtering (not modified)
- **Ez** – Field array before filtering (not modified)
• **Bx** – Field array before filtering (not modified)
• **By** – Field array before filtering (not modified)
• **Bz** – Field array before filtering (not modified)
• **exfab** – pointer to the Ex field (modified)
• **eyfab** – pointer to the Ey field (modified)
• **ezfab** – pointer to the Ez field (modified)
• **bxfab** – pointer to the Bx field (modified)
• **byfab** – pointer to the By field (modified)
• **bzfab** – pointer to the Bz field (modified)

### 6.3.5 Particles

**Particle containers**

Particle structures and functions are defined in `Source/Particles/`. WarpX uses the `Particle` class from AMReX for single particles. An ensemble of particles (e.g., a plasma species, or laser particles) is stored as a `WarpXParticleContainer` (see description below) in a per-box (and even per-tile on CPU) basis.

```cpp
class WarpXParticleContainer : public amrex::ParticleContainer<0, 0, PIdx::nattribs>
```

`WarpXParticleContainer` is the base polymorphic class from which all concrete particle container classes (that store a collection of particles) derive. Derived classes can be used for plasma particles, photon particles, or non-physical particles (e.g., for the laser antenna). It derives from `amrex::ParticleContainer<0,0,PIdx::nattribs>`, where the template arguments stand for the number of int and `amrex::Real` SoA and AoS data in `amrex::Particle`.

- **AoS amrex::Real**: x, y, z (default), 0 additional (first template parameter)
- **AoS int**: id, cpu (default), 0 additional (second template parameter)
- **SoA amrex::Real**: PIdx::nattribs (third template parameter), see PIdx for the list.

`WarpXParticleContainer` contains the main functions for initialization, interaction with the grid (field gather and current deposition) and particle push.

Note: many functions are pure virtual (meaning they MUST be defined in derived classes, e.g., Evolve) or actual functions (e.g. CurrentDeposition).

Subclassed by LaserParticleContainer, PhysicalParticleContainer

Physical species are stored in `PhysicalParticleContainer`, that derives from `WarpXParticleContainer`. In particular, the main function to advance all particles in a physical species is `PhysicalParticleContainer::Evolve` (see below).
virtual void PhysicalParticleContainer::Evolve(int lev, const amrex::MultiFab &Ex, const amrex::MultiFab &Ey, const amrex::MultiFab &Ez, const amrex::MultiFab &Bx, const amrex::MultiFab &By, const amrex::MultiFab &Bz, amrex::MultiFab &jx, amrex::MultiFab &jy, amrex::MultiFab &jz, amrex::MultiFab &cjx, amrex::MultiFab &cjy, amrex::MultiFab &cjz, amrex::MultiFab &rho, amrex::MultiFab &crho, const amrex::MultiFab *cEx, const amrex::MultiFab *cEy, const amrex::MultiFab *cEz, const amrex::MultiFab *cBx, const amrex::MultiFab *cBy, const amrex::MultiFab *cBz, amrex::Real t, amrex::Real dt, DtType a_dt_type = DtType::Full, bool skip_deposition = false) override

Finally, all particle species (physical plasma species PhysicalParticleContainer, photon species PhotonParticleContainer or non-physical species LaserParticleContainer) are stored in MultiParticleContainer. The class WarpX holds one instance of MultiParticleContainer as a member variable, called WarpX::mypc (where mypc stands for “my particle containers”):

class MultiParticleContainer
The class MultiParticleContainer holds multiple instances of the polymorphic class WarpXParticleContainer, stored in its member variable “allcontainers”. The class WarpX typically has a single (pointer to an) instance of MultiParticleContainer.

MultiParticleContainer typically has two types of functions:

- Functions that loop over all instances of WarpXParticleContainer in allcontainers and calls the corresponding function (for instance, MultiParticleContainer::Evolve loops over all particles containers and calls the corresponding WarpXParticleContainer::Evolve function).
- Functions that specifically handle multiple species (for instance ReadParameters or mapSpeciesProduct).

Loop over particles

A typical loop over particles reads:

```cpp
// pc is a std::unique_ptr<WarpXParticleContainer>
// Loop over MR levels
for (int lev = 0; lev <= finest_level; ++lev) {
    // Loop over particles, box by box
    for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti) {
        // Do something on particles
        // [MY INNER LOOP]
    }
}
```

The innermost step [MY INNER LOOP] typically calls amrex::ParallelFor to perform operations on all particles in a portable way. For this reasons, the particle data needs to be converted in plain-old-data structures. The innermost loop in the code snippet above could look like:

```cpp
// Get Array-Of-Struct particle data, also called data
// (x, y, z, id, cpu)
const auto& particles = pti.GetArrayOfStructs();
// Get Struct-Of-Array particle data, also called attribs
// (ux, uy, uz, w, Exp, Ey, Ez, Bx, By, Bz)
```

(continues on next page)
auto& attrs = pti.GetAttribs();
auto& Exp = attrs[PIdx::Ex];
// [...] // Number of particles in this box
const long np = pti.numParticles();

Link fields and particles?

In WarpX, the loop over boxes through a MultiFab iterator MFIter and the loop over boxes through a ParticleContainer iterator WarpXParIter are consistent.

On a loop over boxes in a MultiFab (MFIter), it can be useful to access particle data on a GPU-friendly way. This can be done by:

// Index of grid (= box)
const int grid_id = mfi.index();
// Index of tile within the grid
const int tile_id = mfi.LocalTileIndex();
// Get GPU-friendly arrays of particle data
auto& ptile = GetParticles(lev)[std::make_pair(grid_id,tile_id)];
ParticleType* pp = particle_tile.GetArrayOfStructs().data();
// Only need attribs (i.e., SoA data)
auto& soa = ptile.GetStructOfArrays();
// As an example, let's get the ux momentum
const ParticleReal * const AMREX_RESTRICT ux = soa.GetData(PIdx::ux).data();

On a loop over particles it can be useful to access the fields on the box we are looping over (typically when we use both field and particle data on the same box, for field gather or current deposition for instance). This is done for instance by adding this snippet in [MY INNER LOOP]:

// E is a reference to, say, WarpX::Efield_aux
// Get the Ex field on the grid
const FArrayBox& exfab = (*E[lev][0])[pti];
// Let's be generous and also get the underlying box (i.e., index info)
const Box& box = pti.validbox();

Main functions

Warning: doxygenfunction: Cannot find function “PhysicalParticleContainer::FieldGather” in doxygen xml output for project “WarpX” from directory: ../../doxyxml/

virtual void PhysicalParticleContainer::PushPX(WarpXParIter &pti, amrex::FArrayBox const *efxfab, amrex::FArrayBox const *eyfab, amrex::FArrayBox const *ezfab, amrex::FArrayBox const *bxfab, amrex::FArrayBox const *byfab, amrex::FArrayBox const *bzfab, const IntVect ngEB, const int, const long offset, const long np_to_push, int lev, int gather_lev, amrex::Real dt, ScaleFields scaleFields, DType a_dt_type = DType::Full)
Buffers

To reduce numerical artifacts at the boundary of a mesh-refinement patch, WarpX has an option to use buffers: When particles evolve on the fine level, they gather from the coarse level (e.g., Efield_cax, a copy of the aux data from the level below) if they are located on the fine level but fewer than WarpX::n_field_gather_buffer cells away from the coarse-patch boundary. Similarly, when particles evolve on the fine level, they deposit on the coarse level (e.g., Efield_cp) if they are located on the fine level but fewer than WarpX::n_current_deposition_buffer cells away from the coarse-patch boundary.

WarpX::gather_buffer_masks and WarpX::current_buffer_masks contain masks indicating if a cell is in the interior of the fine-resolution patch or in the buffers. Then, particles depending on this mask in

```
void PhysicalParticleContainer::PartitionParticlesInBuffers(long &nfine_current, long &nfine_gather, long const np, WarpXParIter &pti, int const lev, amrex::cMultiFab const *current_masks, amrex::cMultiFab const *gather_masks)
```

Note: Buffers are complex!

Particle attributes

WarpX adds the following particle attributes by default to WarpX particles. These attributes are either stored in an Array-of-Struct (AoS) or Struct-of-Array (SoA) location of the AMReX particle containers. The data structures for those are either pre-described at compile-time (CT) or runtime (RT).
<table>
<thead>
<tr>
<th>Attribute name</th>
<th>int/real</th>
<th>Description</th>
<th>Where</th>
<th>When</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>position_x/y/z</td>
<td>real</td>
<td>Particle position.</td>
<td>AoS</td>
<td>CT</td>
<td></td>
</tr>
<tr>
<td>cpu</td>
<td>int</td>
<td>CPU index where the particle was created.</td>
<td>AoS</td>
<td>CT</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>int</td>
<td>CPU-local particle index where the particle was created.</td>
<td>AoS</td>
<td>CT</td>
<td></td>
</tr>
<tr>
<td>ionizationLevel</td>
<td>int</td>
<td>Ion ionization level</td>
<td>SoA</td>
<td>RT</td>
<td>Added when ionization physics is used.</td>
</tr>
<tr>
<td>opticalDepthQSR</td>
<td>real</td>
<td>QED: optical depth of the Quantum-Synchrotron process</td>
<td>SoA</td>
<td>RT</td>
<td>Added when PICSAR QED physics is used.</td>
</tr>
<tr>
<td>opticalDepthBW</td>
<td>real</td>
<td>QED: optical depth of the Breit-Wheeler process</td>
<td>SoA</td>
<td>RT</td>
<td>Added when PICSAR QED physics is used.</td>
</tr>
</tbody>
</table>

WarpX allows extra runtime attributes to be added to particle containers (through `AddRealComp("attrname")` or `AddIntComp("attrname")`). The attribute name can then be used to access the values of that attribute. For example, using a particle iterator, `pti`, to loop over the particles the command `pti.GetAttribs(particle_comps["attrname"]).dataPtr();` will return the values of the "attrname" attribute.

User-defined integer or real attributes are initialized when particles are generated in `AddPlasma()`. The attribute is initialized with a required user-defined parser function. Please see the input options `addIntegerAttributes` and `addRealAttributes` for a user-facing documentation.

Commonly used runtime attributes are described in the table below and are all part of SoA particle storage:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>int/real</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev_x/y/z</td>
<td>real</td>
<td>The coordinates of the particles at the previous timestep.</td>
<td>user-defined</td>
</tr>
<tr>
<td>orig_x/y/z</td>
<td>real</td>
<td>The coordinates of the particles when they were created.</td>
<td>user-defined</td>
</tr>
</tbody>
</table>

A Python example that adds runtime options can be found in `Examples/Tests/ParticleDataPython`.

**Note:** Only use _ to separate components of vectors!
6.3.6 Initialization

Note: Section almost empty!!

General simulation initialization
Regular simulation
Running in a boosted frame
Field initialization
Particle initialization

6.3.7 Diagnostics

Regular Diagnostics (plotfiles)

Note: Section empty!

Back-Transformed Diagnostics

Note: Section empty!

6.3.8 Moving Window

Note: Section empty!

6.3.9 QED

Quantum synchrotron

Note: Section empty!
If the code is compiled with QED and the user activates the Schwinger process in the input file, electron-positron pairs can be created in vacuum in the function `MultiParticleContainer::doQEDSchwinger`:

```cpp
void MultiParticleContainer::doQEDSchwinger()
```

If Schwinger process is activated, this function is called at every timestep in Evolve and is used to create Schwinger electron-positron pairs. Within this function we loop over all cells to calculate the number of created physical pairs. If this number is higher than 0, we create a single particle per species in this cell, with a weight corresponding to the number of physical particles.

`MultiParticleContainer::doQEDSchwinger` in turn calls the function `filterCreateTransformFromFAB`:

```cpp
template<int N, typename DstTile, typename FABs, typename Index, typename FilterFunc, typename CreateFunc1, typename CreateFunc2, typename TransFunc>
Index filterCreateTransformFromFAB(DstTile &dst1, DstTile &dst2, const amrex::Box &box, const FABs &src_FABs, const Index dst1_index, const Index dst2_index, FilterFunc &&filter, CreateFunc1 &&create1, CreateFunc2 &&create2, TransFunc &&transform) noexcept
```

Apply a filter on a list of FABs, then create and apply a transform operation to the particles depending on the output of the filter.

This version of the function takes as input a filter functor (and an array of FABs that can be used in the filter functor), uses it to obtain a mask and a FAB and then calls another version of `filterCreateTransformFromFAB` that takes the mask and the FAB as inputs.

**Template Parameters**

- `N` – number of particles created in the dst(s) in each cell
- `DstTile` – the dst particle tile type
- `FABs` – the src array of Array4 type
- `Index` – the index type, e.g. unsigned int
- `FilterFunc` – the filter function type
- `CreateFunc1` – the create function type for dst1
- `CreateFunc2` – the create function type for dst2
- `TransFunc` – the transform function type

**Parameters**

- `dst1` – [inout] the first destination tile
- `dst2` – [inout] the second destination tile
- `box` – [in] the box where the particles are created
- `src_FABs` – [in] A collection of source data, e.g. a class with Array4 to the EM fields, defined on box on which the filter operation is applied
- `dst1_index` – [in] the location at which to starting writing the result to dst1
• **dst2_index** – [in] the location at which to starting writing the result to dst2
• **filter** – [in] a callable returning a value > 0 if particles are to be created in the considered cell.
• **create1** – [in] callable that defines what will be done for the create step for dst1.
• **create2** – [in] callable that defines what will be done for the create step for dst2.
• **transform** – [in] callable that defines the transformation to apply on dst1 and dst2.

**Returns**  
um_added the number of particles that were written to dst1 and dst2.

`filterCreateTransformFromFAB` proceeds in three steps. In the filter phase, we loop on every cell and calculate the number of physical pairs created within the time step $dt$ as a function of the electromagnetic field at the given cell position. This probabilistic calculation is done via a wrapper that calls the PICSAR library. In the create phase, the particles are created at the desired positions, currently at the cell nodes. In the transform phase, we assign a weight to the particles depending on the number of physical pairs created. At most one macroparticle is created per cell per timestep per species, with a weight corresponding to the total number of physical pairs created.

So far the Schwinger module requires using `warpx.do_nodal=1` or `algo.field_gathering=momentum-conserving` (so that the auxiliary fields are calculated on the nodes) and is not compatible with either mesh refinement, RZ coordinates or single precision.

### 6.3.10 Portability

**Note:** Section empty!

### 6.3.11 Warning logger

The warning logger allows grouping the warning messages raised during the simulation, in order to display them together in a list (e.g., right after step 1 and at the end of the simulation).

<table>
<thead>
<tr>
<th>General</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>If no warning messages are raised, the warning list should look as follows:</td>
<td></td>
</tr>
</tbody>
</table>

```
**** WARNINGS ******************************************
* GLOBAL warning list after [ FIRST STEP ]
*
* No recorded warnings.
******************************************
```

On the contrary, if warning messages are raised, the list should look as follows:

```
**** WARNINGS ******************************************
* GLOBAL warning list after [ FIRST STEP ]
*
* --> [!! ] [Species] [raised once]
* Both 'electrons.charge' and electrons.species_type' are specified.
* electrons.charge' will take precedence.
* @ Raised by: ALL
*
```

(continues on next page)
Here, GLOBAL indicates that warning messages are gathered across all the MPI ranks (specifically after the FIRST STEP).

Each entry of warning list respects the following format:

```plaintext
* --> [PRIORITY] [TOPIC] [raised COUNTER]
*   MULTILINE MESSAGE
*   MULTILINE MESSAGE
*   @ Raised by: WHICH_RANKS
```

where:

- [PRIORITY] can be [!] (low priority), [!!] (medium priority) or [!!!] (high priority). It indicates the importance of the warning.
- [TOPIC] indicates which part of the code is concerned by the warning (e.g., particles, laser, parallelization...)
- MULTILINE MESSAGE is an arbitrary text message. It can span multiple-lines. Text is wrapped automatically.
- COUNTER indicates the number of times the warning was raised across all the MPI ranks. This means that if we run WarpX with 2048 MPI ranks and each rank raises the same warning once, the displayed message will be [raised 2048 times]. Possible values are once, twice, XX times.
- WHICH_RANKS can be either ALL or a sequence of rank IDs. It is the list of the MPI ranks which have raised the warning message.

Entries are sorted first by priority (high priority first), then by topic (alphabetically) and finally by text message (alphabetically).

**How to record a warning for later display**

In the code, instead of using amrex::Warning to immediately print a warning message, the following method should be called:

```cpp
WarpX::GetInstance().RecordWarning(
    "QED",
    "Using default value (2*me*c^2) for photon energy creation threshold",
    WarnPriority::low);
```

In this example, QED is the topic, Using [...] is the warning message and WarnPriority::low is the priority. RecordWarning is not a collective call and should also be thread-safe (it can be called in OpenMP loops). In case the user wants to also print the warning messages immediately, the runtime parameter warpx.always_warn_immediately can be set to 1.
How to print the warning list

The warning list can be printed as follows:

```c
warpx.PrintGlobalWarnings("THE END");
```

where the string is a temporal marker that appears in the warning list. At the moment this is done right after step one and at the end of the simulation. Calling this method triggers several collective calls that allow merging all the warnings recorded by all the MPI ranks.

### Implementation details

#### How warning messages are recorded

Warning messages are stored by each rank as a map associating each message with a counter. A message is defined by its priority, its topic and its text. Given two messages, if any of these components differ between the two, the messages are considered as different.

#### How the global warning list is generated

In order to generate the global warning list we follow the strategy outlined below.

1. Each MPI rank has a `map<Msg, counter>`, associating each with a counter, which counts how many times the warning has been raised on that rank.

2. When `PrintGlobalWarnings` is called, the MPI ranks send to the I/O rank the number of different warnings that they have observed. The I/O rank finds the rank having more warnings and broadcasts this information back to all the others. This rank, referred in the following as *gather rank*, will lead the generation of the global warning list.

3. The *gather rank* serializes its warning messages `[……..]` into a byte array and broadcasts this array to all the other ranks.

4. The other ranks unpack this byte array, obtaining a list of messages `[……..]`.

5. For each message seen by the *gather rank*, each rank prepares a vector containing the number of times it has seen that message (i.e., the counter in `map<Msg, counter>` if `Msg` is in the map): `[1,0,1,4,0,…]`

6. In addition, each rank prepares a vector containing the messages seen only by that rank, associated with the corresponding counter: `[(1), (4),...]`

7. Each rank appends the second list to the first one and packs them into a byte array: `[1,0,1,4,0,…] [(1), (4),…]` –> –> –> –> `[1,0,1,4,0,…] [(1), (4),…]` I/O rank

This procedure is described in more details in these slides.
How to test the warning logger

In order to test the warning logger there is the possibility to inject “artificial” warnings with the input file. For instance, the following input file

```
#################################
####### GENERAL PARAMETERS ######
#################################
max_step = 10
amr.n_cell = 128 128
amr.max_grid_size = 64
amr.blocking_factor = 32
amr.max_level = 0
geometry.dims = 2
geometry.prob_lo = -20.e-6 -20.e-6  # physical domain
geometry.prob_hi = 20.e-6 20.e-6

#################################
####### Boundary condition #######
#################################
boundary.field_lo = periodic periodic
boundary.field_hi = periodic periodic

#################################
############ NUMERICS ############
#################################
warpx.serialize_initial_conditions = 1
warpx.verbose = 1
warpx.cfl = 1.0
warpx.use_filter = 0

# Order of particle shape factors
algo.particle_shape = 1

#################################
######## DEBUG WARNINGS ##########
#################################
warpx.test_warnings = w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12 w13 w14 w15 w16 w17 w18 w19...
w20 w21 w22

w1.topic = "Priority Sort Test"
w1.msg = "Test that priority is correctly sorted"
w1.priority = "low"
w1.all_involved = 1

w2.topic = "Priority Sort Test"
w2.msg = "Test that priority is correctly sorted"
w2.priority = "medium"
w2.all_involved = 1

w3.topic = "Priority Sort Test"
w3.msg = "Test that priority is correctly sorted"
```

(continues on next page)
w3.priority = "high"
w3.all_involved = 1

w4.topic  = "ZZA Topic sort Test"
w4.msg    = "Test that topic is correctly sorted"
w4.priority = "medium"
w4.all_involved = 1

w5.topic  = "ZZB Topic sort Test"
w5.msg    = "Test that topic is correctly sorted"
w5.priority = "medium"
w5.all_involved = 1

w6.topic  = "ZZC Topic sort Test"
w6.msg    = "Test that topic is correctly sorted"
w6.priority = "medium"
w6.all_involved = 1

w7.topic  = "Msg sort Test"
w7.msg    = "AAA Test that msg is correctly sorted"
w7.priority = "medium"
w7.all_involved = 1

w8.topic  = "Msg sort Test"
w8.msg    = "BBB Test that msg is correctly sorted"
w8.priority = "medium"
w8.all_involved = 1

w9.topic  = "Long line"
w9.msg    = "Test very long line: a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a aodule

w9.priority = "medium"
w9.all_involved = 1

w10.topic   = "Repeated warnings"
w10.msg     = "Test repeated warnings"
w10.priority = "high"
w10.all_involved = 1

w11.topic   = "Repeated warnings"
w11.msg     = "Test repeated warnings"
w11.priority = "high"
w11.all_involved = 1

w12.topic   = "Repeated warnings"
w12.msg     = "Test repeated warnings"
w12.priority = "high"
w12.all_involved = 1

w13.topic   = "Not all involved (0)"
w13.msg     = "Test warnings raised by a fraction of ranks"
w13.priority = "high"
w13.all_involved = 0
w13.who_involved = 0

w14.topic = "Not all involved (0)"
w14.msg = "Test warnings raised by a fraction of ranks"
w14.priority = "high"
w14.all_involved = 0
w14.who_involved = 0

w15.topic = "Not all involved (1)"
w15.msg = "Test warnings raised by a fraction of ranks"
w15.priority = "high"
w15.all_involved = 0
w15.who_involved = 1

w16.topic = "Not all involved (1,2)"
w16.msg = "Test warnings raised by a fraction of ranks"
w16.priority = "high"
w16.all_involved = 0
w16.who_involved = 1 2

w17.topic = "Different counters"
w17.msg = "Test that different counters are correctly summed"
w17.priority = "low"
w17.all_involved = 1

w18.topic = "Different counters"
w18.msg = "Test that different counters are correctly summed"
w18.priority = "low"
w18.all_involved = 1

w19.topic = "Different counters"
w19.msg = "Test that different counters are correctly summed"
w19.priority = "low"
w19.all_involved = 0
w19.who_involved = 0

w20.topic = "Different counters B"
w20.msg = "Test that different counters are correctly summed"
w20.priority = "low"
w20.all_involved = 1

w21.topic = "Different counters B"
w21.msg = "Test that different counters are correctly summed"
w21.priority = "low"
w21.all_involved = 1

w22.topic = "Different counters B"
w22.msg = "Test that different counters are correctly summed"
w22.priority = "low"
w22.all_involved = 0
w22.who_involved = 1
should generate the following warning list (if run on 4 MPI ranks):

```plaintext
**** WARNINGS ******************************************************************
* GLOBAL warning list after [ THE END ]
* 
* --> [!!!] [Not all involved (0)] [raised twice]
  Test warnings raised by a fraction of ranks
  @ Raised by: 0
*
* --> [!!!] [Not all involved (1)] [raised once]
  Test warnings raised by a fraction of ranks
  @ Raised by: 1
*
* --> [!!!] [Not all involved (1,2)] [raised twice]
  Test warnings raised by a fraction of ranks
  @ Raised by: 1 2
*
* --> [!!!] [Priority Sort Test] [raised 4 times]
  Test that priority is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [Repeated warnings] [raised 12 times]
  Test repeated warnings
  @ Raised by: ALL
*
* --> [!! ] [Long line] [raised 4 times]
  Test very long line: a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
  @ Raised by: ALL
*
* --> [!!!] [Msg sort Test] [raised 4 times]
  AAA Test that msg is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [Msg sort Test] [raised 4 times]
  BBB Test that msg is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [Priority Sort Test] [raised 4 times]
  Test that priority is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [ZZA Topic sort Test] [raised 4 times]
  Test that topic is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [ZZB Topic sort Test] [raised 4 times]
  Test that topic is correctly sorted
  @ Raised by: ALL
*
* --> [!!!] [ZZC Topic sort Test] [raised 4 times]
  Test that topic is correctly sorted
  @ Raised by: ALL

(continues on next page)
```
6.3.12 Python interface

The Python interface provides low and high level access to much of the data in WarpX. With the low level access, a user has direct access to the underlying memory contained in the MultiFabs and in the particle arrays. The high level provides a more user friendly interface.

High level interface

There are two python modules that provide convenient access to the fields and the particles.

**Fields**

The `fields` module provides wrapper around most of the MultiFabs that are defined in the WarpX class. For a list of all of the available wrappers, see the file `Python/pywarpx/fields.py`. For each MultiFab, there is a function that will return a wrapper around the data. For instance, the function `ExWrapper` returns a wrapper around the $x$ component of the MultiFab vector $E_{\text{field\_aux}}$.

```python
from pywarpx import fields
Ex = fields.ExWrapper()
```

By default, this wraps the MultiFab for level 0. The `level` argument can be specified for other levels. By default, the wrapper only includes the valid cells. To include the ghost cells, set the argument `include_ghosts=True`.

The wrapper provides access to the data via global indexing. Using standard array indexing (with exceptions) with square brackets, the data can be accessed using indices that are relative to the full domain (across the MultiFab and across processors). With multiple processors, the result is broadcast to all processors. This example will return the $B_z$ field at all points along $x$ at the specified $y$ and $z$ indices.

```python
from pywarpx import fields
Bz = fields.BzWrapper()
Bz_along_x = Bz[:,5,6]
```

The same global indexing can be done to set values. This example will set the values over a range in $y$ and $z$ at the specified $x$. The data will be scattered appropriately to the underlying FABs.
from pywarpx import fields
Jy = fields.JyFPWrapper()

The code does error checking to ensure that the specified indices are within the bounds of the global domain. Note that negative indices are handled differently than with numpy arrays because of the possibility of having ghost cells. With ghost cells, the lower ghost cells are accessed using negative indices (since 0 is the index of the lower bound of the valid cells). Without ghost cells, a negative index will always raise an out of bounds error since there are no ghost cells.

Under the covers, the wrapper object has a list of numpy arrays that have pointers to the underlying data, one array for each FAB. When data is being fetched, it loops over that list to gather the data. The result is then gathered among all processors. Note that the result is not writable, in the sense that changing it won’t change the underlying data since it is a copy. When the data is set, using the global indexing, a similar process is done where the processors loop over their FABs and set the data at the appropriate indices.

The wrappers are always up to date since whenever an access is done (either a get or a set), the list of numpy arrays for the FABs is regenerated. In this case, efficiency is sacrificed for consistency.

If it is needed, the list of numpy arrays associated with the FABs can be obtained using the wrapper method _getfields. Additionally, there are the methods _getloveccts and _gethivects that get the list of the bounds of each of the arrays.

Particles

This is still in development.

Tip: A tutorial-style overview of the code structure can also be found in a developer presentation from 03/2020. It contains information about the code structure, a step-by-step description of what happens in a simulation (initialization and iterations) as well as slides on topics relevant to WarpX development.

Information in the following pages are generally more up-to-date, but the slides above might still be useful.

6.4 C++ Objects & Functions

We generate the documentation of C++ objects and functions from our C++ source code by adding Doxygen strings.

Our Doxygen C++ API documentation in classic formatting is located here.

6.5 GNUmake Build System (Legacy)

CMake is our primary build system. In this section, we describe our legacy build scripts - do not use them unless you used them before.

WarpX is built on AMReX, which also provides support for a Linux-centric set of build scripts implemented in GNUmake. Since we sometimes need to move fast and test highly experimental compilers and Unix derivates on core components of WarpX, this set of build scripts is used by some of our experienced developers.
This page describes the most basic build with GNUmake files and points to instructions for more advanced builds.

### 6.5.1 Downloading the source code

Clone the source codes of WarpX, and its dependencies AMReX and PICSAR into one single directory (e.g. `warpx_directory`):

```bash
mkdir warpx_directory
cd warpx_directory
git clone https://github.com/ECP-WarpX/WarpX.git
git clone https://github.com/ECP-WarpX/picsar.git
git clone https://github.com/ECP-WarpX/warpx-data.git
git clone https://github.com/AMReX-Codes/amrex.git
```

**Note:** The `warpx-data` repository is currently only needed for MCC cross-sections.

### 6.5.2 Basic compilation

WarpX requires a C/C++ compiler (e.g., GNU, LLVM or Intel) and an MPI implementation (e.g., OpenMPI or MPICH). Start a GNUmake build by `cd`-ing into the directory `WarpX` and type

```bash
make -j 4
```

This will generate an executable file in the `Bin` directory.

### 6.5.3 Compile-time vs. run-time options

WarpX has multiple compile-time and run-time options. The compilation options are set in the file `GNUmakefile`. The default options correspond to an optimized code for 3D geometry. The main compile-time options are:

- **`DIM=3` or `2`:** Geometry of the simulation (note that running an executable compiled for 3D with a 2D input file will crash).
- **`DEBUG=FALSE` or `TRUE`:** Compiling in DEBUG mode can help tremendously during code development.
- **`USE_PSAATD=FALSE` or `TRUE`:** Compile the Pseudo-Spectral Analytical Time Domain Maxwell solver. Requires an FFT library.
- **`USE_RZ=FALSE` or `TRUE`:** Compile for 2D axisymmetric geometry.
- **`COMP=gcc` or `intel`:** Compiler.
- **`USE_MPI=TRUE` or `FALSE`:** Whether to compile with MPI support.
- **`USE_OMP=TRUE` or `FALSE`:** Whether to compile with OpenMP support.
- **`USE_GPU=TRUE` or `FALSE`:** Whether to compile for Nvidia GPUs (requires CUDA).
- **`USE_OPENPMD=TRUE` or `FALSE`:** Whether to support openPMD for I/O (requires openPMD-api).
• **MPI_THREAD_MULTIPLE=TRUE or FALSE**: Whether to initialize MPI with thread multiple support. Required to use asynchronous IO with more than `amrex.async_out_nfiles` (by default, 64) MPI tasks. Please see data formats for more information.

• **PRECISION=FLOAT USE_SINGLE_PRECISION_PARTICLES=TRUE**: Switch from default double precision to single precision (experimental).

  For a description of these different options, see the corresponding page in the AMReX documentation.

Alternatively, instead of modifying the file `GNUmakefile`, you can directly pass the options in command line; for instance:

```
make -j 4 USE_OMP=FALSE
```

In order to clean a previously compiled version (typically useful for troubleshooting, if you encounter unexpected compilation errors):

```
make realclean
```

before re-attempting compilation.

### 6.5.4 Advanced GNUmake instructions

**Building WarpX with support for openPMD output**

WarpX can dump data in the openPMD format. This feature currently requires to have a parallel version of HDF5 installed; therefore we recommend to use spack in order to facilitate the installation.

More specifically, we recommend that you try installing the openPMD-api library 0.14.2 or newer using spack (first section below). If this fails, a back-up solution is to install parallel HDF5 with spack, and then install the openPMD-api library from source.

In order to install spack, you can simply do:

```
git clone https://github.com/spack/spack.git
export SPACK_ROOT=$PWD/spack
. $SPACK_ROOT/share/spack/setup-env.sh
```

You may want to auto-activate spack when you open a new terminal by adding this to your `$HOME/.bashrc` file:

```
echo -e "# activate spack package manager

${SPACK_ROOT}/share/spack/setup-env.sh" >>

$HOME/.bashrc
```

**WarpX Development Environment with Spack**

Create and activate a Spack environment with all software needed to build WarpX

```
spack env create warpx-dev # you do this once
spack env activate warpx-dev
spack add gmake
spack add mpi
spack add openpmd-api
spack add pkg-config
spack install
```
This will download and compile all dependencies.

Whenever you need this development environment in the future, just repeat the quick spack env activate warpx-dev step. For example, we can now compile WarpX by cd-ing into the WarpX folder and typing:

```
spack env activate warpx-dev
make -j 4 USE_OPENPMD=TRUE
```

You will also need to load the same spack environment when running WarpX, for instance:

```
spack env activate warpx-dev
mpirun -np 4 ./warpx.exe inputs
```

You can check which Spack environments exist and if one is still active with

```
spack env list  # already created environments
spack env st    # is an environment active?
```

Installing openPMD-api from source

You can also build openPMD-api from source, e.g. to build against the module environment of a supercomputer cluster.

First, load the according modules of the cluster to support the openPMD-api dependencies. You can find the required and optional dependencies here.

You usually just need a C++ compiler, CMake, and one or more file backend libraries, such as HDF5 and/or ADIOS2.

If optional dependencies are installed in non-system paths, one needs to hint their installation location with an environment variable during the build phase:

```
# optional: only if you manually installed HDF5 and/or ADIOS2 in custom directories
export HDF5_ROOT=$HOME/path_to_installed_software/hdf5-1.12.0/
export ADIOS2_ROOT=$HOME/path_to_installed_software/adios2-2.7.1/
```

Then, in the $HOME/warpx_directory/, download and build openPMD-api:

```
git clone https://github.com/openPMD/openPMD-api.git
mkdir openPMD-api-build
cd openPMD-api-build
make ..//openPMD-api -DopenPMD_USE_PYTHON=OFF -DCMAKE_INSTALL_PREFIX=$HOME/warpx_directory/openPMD-install/ -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=ON -DCMAKE_INSTALL_RPATH='$ORIGIN'
cmake --build . --target install
```

Finally, compile WarpX:

```
cd ../WarpX
# Note that one some systems, /lib might need to be replaced with /lib64.
export PKG_CONFIG_PATH=$HOME/warpx_directory/openPMD-install/lib/pkgconfig:$PKG_CONFIG_PATH
export CMAKE_PREFIX_PATH=$HOME/warpx_directory/openPMD-install:$CMAKE_PREFIX_PATH
make -j 4 USE_OPENPMD=TRUE
```
Note: If you compile with CMake, all you need to add is the -DWarpX_OPENPMD=ON option (on by default), and we will download and build openPMD-api on-the-fly.

When running WarpX, we will recall where you installed openPMD-api via RPATHs, so you just need to load the same module environment as used for building (same MPI, HDF5, ADIOS2, for instance).

```bash
# module load ... (compiler, MPI, HDF5, ADIOS2, ...)
mpirun -np 4 ./warpx.exe inputs
```

### Building the spectral solver

By default, the code is compiled with a finite-difference (FDTD) Maxwell solver. In order to run the code with a spectral solver, you need to:

- Install (or load) an MPI-enabled version of FFTW. For instance, for Debian, this can be done with
  ```bash
  apt-get install libfftw3-dev libfftw3-mpi-dev
  ```
- Set the environment variable FFTW_HOME to the path for FFTW. For instance, for Debian, this is done with
  ```bash
  export FFTW_HOME=/usr/
  ```
- Set USE_PSATD=TRUE when compiling:
  ```bash
  make -j 4 USE_PSATD=TRUE
  ```

See Building WarpX to use RZ geometry for using the spectral solver with USE_RZ. Additional steps are needed. PSATD is compatible with single precision, but please note that, on CPU, FFTW needs to be compiled with option --enable-float.

### Building WarpX to use RZ geometry

WarpX can be built to run with RZ geometry. Both an FDTD solver (the default) and a PSATD solver are available. Both solvers allow multiple azimuthal modes.

To select RZ geometry, set the flag USE_RZ = TRUE when compiling:

```bash
make -j 4 USE_RZ=TRUE
```

Note that this sets DIM=2, which is required with USE_RZ=TRUE. The executable produced will have “RZ” as a suffix.
RZ geometry with spectral solver

Additional steps are needed to build the spectral solver. Some of the steps are the same as is done for the Cartesian spectral solver, setting up the FFTW package and setting `USE_PSATD=TRUE`.

- Install (or load) an MPI-enabled version of FFTW. For instance, for Debian, this can be done with

  ```
  apt-get install libfftw3-dev libfftw3-mpi-dev
  ```

- Set the environment variable `FFTW_HOME` to the path for FFTW. For instance, for Debian, this is done with

  ```
  export FFTW_HOME=/usr/
  ```

- Download and build the blaspp and lapackpp packages. These can be obtained from bitbucket.

  ```
  git clone https://bitbucket.org/icl/blaspp.git
  git clone https://bitbucket.org/icl/lapackpp.git
  ```

  The two packages can be built in multiple ways. A recommended method is to follow the cmake instructions provided in the INSTALL.md that comes with the packages. They can also be installed using spack.

- Set the environment variables `BLASPP_HOME` and `LAPACKPP_HOME` to the locations where the packages libraries were installed. For example, using bash:

  ```
  export BLASPP_HOME=/location/of/installation/blaspp
  export LAPACKPP_HOME=/location/of/installation/lapackpp
  ```

- In some case, the blas and lapack libraries need to be specified. If needed, this can be done by setting the `BLAS_LIB` and `LAPACK_LIB` environment variables appropriately. For example, using bash:

  ```
  export BLAS_LIB=-lblas
  ```

- Set `USE_PSATD=TRUE` when compiling:

  ```
  make -j 4 USE_RZ=TRUE USE_PSATD=TRUE
  ```

Building WarpX with GPU support (Linux only)

**Warning:** In order to build WarpX on a specific GPU cluster (e.g. Summit), look for the corresponding specific instructions, instead of those on this page.

In order to build WarpX with GPU support, make sure that you have `cuda` and `mpich` installed on your system. (Compiling with `openmpi` currently fails.) Then compile WarpX with the option `USE_GPU=TRUE`, e.g.

```
make -j 4 USE_GPU=TRUE
```
## Installing WarpX as a Python Package

A full Python installation of WarpX can be done, which includes a build of all of the C++ code, or a pure Python version can be made which only installs the Python scripts. WarpX requires Python version 3.6 or newer.

### For a full Python installation of WarpX

WarpX’ Python bindings depend on `numpy`, `periodictable`, `picmistandard`, and `mpi4py`.

Type

```
make -j 4 USE_PYTHON_MAIN=TRUE
```

or edit the `GNUmakefile` and set `USE_PYTHON_MAIN=TRUE`, and type

```
make -j 4
```

Additional compile time options can be specified as needed. This will compile the code, and install the Python bindings and the Python scripts as a package (named `pywarpx`) in your standard Python installation (i.e. in your `site-packages` directory).

If you do not have write permission to the default Python installation (e.g. typical on computer clusters), there are two options. The recommended option is to use a virtual environment, which provides the most flexibility and robustness.

Alternatively, add the `--user` install option to have WarpX installed elsewhere.

```
make -j 4 PYINSTALLOPTIONS=--user
```

With `--user`, the default location will be in your home directory, `~/.local`, or the location defined by the environment variable `PYTHONUSERBASE`.

In HPC environments, it is often recommended to install codes in scratch or work space which typically have faster disk access.

The different dimensioned versions of WarpX, 3D, 2D, and RZ, can coexist in the Python installation. The appropriate one will be imported depending on the input file. Note, however, other options will overwrite - compiling with `DEBUG=TRUE` will replace the version compiled with `DEBUG=FALSE` for example.

### For a pure Python installation

This avoids the compilation of the C++ and is recommended when only using the Python input files as preprocessors.

This installation depend on `numpy`, `periodictable`, and `picmistandard`.

Go into the Python subdirectory and run

```
python setup.py install
```

This installs the Python scripts as a package (named `pywarpx`) in your standard Python installation (i.e. in your `site-packages` directory). If you do not have write permission to the default Python installation (e.g. typical on computer clusters), there are two options. The recommended option is to use a virtual environment, which provides the most flexibility and robustness.

Alternatively, add the `--user` install option to have WarpX installed elsewhere.

```
python setup.py install --user
```
With --user, the default location will be in your home directory, ~/.local, or the location defined by the environment variable PYTHONUSERBASE.

Building with Spack

As mentioned in the install section, WarpX can be installed using Spack. From the Spack web page: “Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments.”

**Note:** Quick-start hint for macOS users: Before getting started with Spack, please check what you manually installed in /usr/local. If you find entries in bin/, lib/ et al. that look like you manually installed MPI, HDF5 or other software at some point, then remove those files first.

If you find software such as MPI in the same directories that are shown as symbolic links then it is likely you brew installed software before. Run brew unlink ... on such packages first to avoid software incompatibilities.

Spack is available from github. Spack only needs to be cloned and can be used right away - there are no installation steps. Do not miss out on the official Spack tutorial if you are new to Spack.

The spack command, spack/bin/spack, can be used directly or spack/bin can be added to your PATH environment variable.

WarpX is built with the single command

```
spack install warpx
```

This will build the 3-D version of WarpX using the development branch. At the very end of the output from build sequence, Spack tells you where the WarpX executable has been placed. Alternatively, spack load warpx can be called, which will put the executable in your PATH environment variable.

WarpX can be built in several variants, see

```
spack info warpx
spack info py-warpx
```

for all available options.

For example

```
spack install warpx dims=2 build_type=Debug
```

will build the 2-D version and also turns debugging on.

See spack help --spec for all syntax details. Also, please consult the basic usage section of the Spack package manager for an extended introduction to Spack.

The Python version of WarpX is available through the py-warpx package.
6.6 FAQ

This section lists frequently asked developer questions.

6.6.1 What is \(0.0_{\text{rt}}\)?

It’s a C++ floating-point literal for zero of type `amrex::Real`.

We use literals to define constants with a specific type, in that case the zero-value. There is also \(0.0_{\text{prt}}\), which is a literal zero of type `amrex::ParticleReal`. In std C++, you know: \(0.0\) (literal double), \(0.0f\) (literal float) and \(0.0L\) (literal long double). We do not use use those, so that we can configure floating point precision at compile time and use different precision for fields (`amrex::Real`) and particles (`amrex::ParticleReal`).

You can also write things like \(42.0_{\text{prt}}\) if you like to have another value than zero.

We use these C++ user literals ([1], [2], [3]), because we want to avoid that double operations, i.e., \(3. / 4.\), implicit casts, or even worse integer operations, i.e., \(3 / 4\), sneak into the code base and make results wrong or slower.

6.6.2 Do you worry about using `size_t` vs. `uint` vs. `int` for indexing things?

`std::size_t` is the C++ unsigned int type for all container sizes.

Close to but not necessarily `uint`, depends on the platform. For “hot” inner loops, you want to use `int` instead of an unsigned integer type. Why? Because `int` has no handling for overflows (it is intentional, undefined behavior in C++), which allows compilers to vectorize easier, because they don’t need to check for an overflow every time one reaches the control/condition section of the loop.

C++20 will also add support for `ssize` (signed size), but we currently require C++17 for builds. Thus, sometimes you need to `static_cast<int>(...)`.

6.6.3 What does `std::make_unique` do?

`make_unique` is a C++ factory method that creates a `std::unique_ptr<T>`.

Follow-up: Why use this over just `*my_ptr = new <class>`?

Because so-called smart-pointers, such as `std::unique_ptr<T>`, do delete themselves automatically when they run out of scope. That means: no memory leaks, because you cannot forget to delete them again.

6.6.4 Why name header files `.H` instead of `.h`?

This is just a convention that we follow through the code base, which slightly simplifies what we need to parse in our various build systems. We inherited that from AMReX. Generally speaking, C++ file endings can be arbitrary, we just keep them consistent to avoid confusion in the code base.

To be explicit and avoid confusion (with C/Object), we might change them all to `.hpp` and `.cpp/.cxx` at some point, but for now `.H` and `.cpp` is what we do (as in AMReX).
6.6.5 What does const int /*i_buffer*/ mean in argument list?

This is often seen in a derived class, overwriting an interface method. It means we do not name the parameter because we do not use it when we overwrite the interface. But we add the name as a comment /* ... */ so that we know what we ignored when looking at the definition of the overwritten method.
MAINTENANCE

7.1 Dependencies & Releases

7.1.1 Update WarpX’ Core Dependencies

WarpX has direct dependencies on AMReX and PICSAR, which we periodically update.

The following scripts automate this workflow, in case one needs a newer commit of AMReX or PICSAR between releases:

`./Tools/Release/updateAMReX.py`
`./Tools/Release/updatePICSAR.py`

7.1.2 Create a new WarpX release

WarpX has one release per month. The version number is set at the beginning of the month and follows the format `YYYY.MM`.

In order to create a GitHub release, you need to:

1. Create a new branch from `development` and update the version number in all source files. We usually wait for the AMReX release to be tagged first, then we also point to its tag.

   There is a script for updating core dependencies of WarpX and the WarpX version:

   `./Tools/Release/updateAMReX.py`
   `./Tools/Release/updatePICSAR.py`
   `./Tools/Release/newVersion.sh`

   For a WarpX release, ideally a `git tag` of AMReX & PICSAR shall be used instead of an unnamed commit.

   Then open a PR, wait for tests to pass and then merge.

2. **Local Commit (Optional):** at the moment, @ax3l is managing releases and signs tags (naming: `YYYY.MM`) locally with his GPG key before uploading them to GitHub.

   **Publish:** On the [GitHub Release page](https://github.com), create a new release via Draft a new release. Either select the locally created tag or create one online (naming: `YYYY.MM`) on the merged commit of the PR from step 1.

   In the release description, please specify the compatible versions of dependencies (see previous releases), and provide info on the content of the release. In order to get a list of PRs merged since last release, you may run
### Automated performance tests

WarpX has automated performance test scripts, which run weak scalings for various tests on a weekly basis. The results are stored in the `perf_logs` repo and plots of the performance history can be found on this page.

These performance tests run automatically, so they need to do git operations etc. For this reason, they need a separate clone of the source repos, so they don’t conflict with one’s usual operations. This is typically in a sub-directory in the `$HOME`, with variable `$AUTOMATED_PERF_TESTS` pointing to it. Similarly, a directory is needed to run the simulations and store the results. By default, it is `$SCRATCH/performance_warpx`.

The test runs a weak scaling (1,2,8,64,256,512 nodes) for 6 different tests `Tools/PerformanceTests/automated_test_{1,2,3,4,5,6}_*`, gathered in 1 batch job per number of nodes to avoid submitting too many jobs.

#### Setup on Summit @ OLCF

Here is an example setup for Summit:

```bash
# I put the next three lines in $HOME/my_bashrc.sh
export proj=aph114       # project for job submission
export AUTOMATED_PERF_TESTS=$HOME/AUTOMATED_PERF_TESTS/
export SCRATCH=/gpfs/alpine/scratch/$(whoami)/$proj/

mkdir $HOME/AUTOMATED_PERF_TESTS
cd $AUTOMATED_PERF_TESTS
git clone https://github.com/ECP-WarpX/WarpX.git warpx
git clone https://github.com/ECP-WarpX/picsar.git
git clone https://github.com/AMReX-Codes/amrex.git
git clone https://github.com/ECP-WarpX/perf_logs.git
```

Then, in `$AUTOMATED_PERF_TESTS`, create a file `run_automated_performance_tests_512.sh` with the following content:

```bash
#!/bin/bash -l
#BSUB -P APH114
#BSUB -W 00:15
#BSUB -nnodes 1
#BSUB -J PERFTEST
#BSUB -e err_automated_tests.txt
#BSUB -o out_automated_tests.txt

module load nano
module load cmake/3.20.2
module load gcc/9.3.0
module load cuda/11.0.3
module load blaspp/2021.04.01
module load lapackpp/2021.04.00
```

(continues on next page)
module load boost/1.76.0
module load adios2/2.7.1
module load hdf5/1.10.7

module unload darshan-runtime

export AMREX_CUDA_ARCH=7.0
export CC=$(which gcc)
export CXX=$(which g++)
export FC=$(which gfortran)
export CUDACXX=$(which nvcc)
export CUDAHOSTCXX=$(which g++)

# Make sure all dependencies are installed and loaded

# Make sure all dependencies are installed and loaded
cd $HOME
module load python/3.8.10
module load freetype/2.10.4 # matplotlib
module load openblas/0.3.5-omp
export BLAS=$OLCF_OPENBLAS_ROOT/lib/libopenblas.so
export LAPACK=$OLCF_OPENBLAS_ROOT/lib/libopenblas.so

python3 -m pip install --user --upgrade pip
python3 -m pip install --user virtualenv
python3 -m venv $HOME/sw/venvs/warpx-perftest
source $HOME/sw/venvs/warpx-perftest/bin/activate

# While setting up the performance tests for the first time, # execute the lines above this comment and then the commented # lines below this comment once, before submission. # The commented lines take too long for the job script.
#python3 -m pip install --upgrade pip
#python3 -m pip install --upgrade cython
#python3 -m pip install --upgrade numpy
#python3 -m pip install --upgrade markupsafe
#python3 -m pip install --upgrade pandas
#python3 -m pip install --upgrade matplotlib==3.2.2 # does not try to build freetype._ # itself
#python3 -m pip install --upgrade bokeh
#python3 -m pip install --upgrade gitpython
#python3 -m pip install --upgrade tables

# Run the performance test suite

# Run the performance test suite

Then, running

bsub run_automated_performance_tests_512.sh

will submit this job once, and all the following ones. It will:

7.2. Automated performance tests 237
• Create directory `$SCRATCH/performance_warpx` if doesn’t exist.

• Create 1 sub-directory per week per number of nodes (1,2,8,64,256,512).

• Submit one job per number of nodes. It will run 6 different tests, each twice (to detect fluctuations).

• Submit an analysis job, that will read the results ONLY AFTER all runs are finished. This uses the dependency feature of the batch system.

• This job reads the Tiny Profiler output for each run, and stores the results in a pandas file at the hdf5 format.

• Execute `write_csv.py` from the `perf_logs` repo to append a csv and a hdf5 file with the new results.

• Commit the results (but DO NOT PUSH YET)

    Then, the user periodically has to

```
cd $AUTOMATED_PERF_TESTS/perf_logs
git pull # to get updates from someone else, or from another supercomputer
git push
```

This will update the database but not the online plots. For this, you need to periodically run something like

```
cd $AUTOMATED_PERF_TESTS/perf_logs
git pull
git add -u
python generate_index_html.py
git commit -m "upload new html page"
git push
```

7.2.2 Setup on Cori @ NERSC

Still to be written!
8.1 Glossary

In daily communication, we tend to abbreviate a lot of terms. It is important to us to make it easy to interact with the WarpX community and thus, this list shall help to clarify often used terms.

8.1.1 Abbreviations

• 2FA: Two-factor-authentication
• ABLASTR: Accelerated BLAST Recipes, the library inside WarpX to share functionality with other BLAST codes
• ALCF: Argonne Leadership Computing Facility, a supercomputing center located near Chicago, IL (USA)
• BLAST: Beam, Plasma & Accelerator Simulation Toolkit
• AMR: adaptive mesh-refinement
• BC: boundary condition (of a simulation)
• BTD: backtransformed diagnostics, a method to collect data for analysis from a boosted frame simulation
• CFL: the Courant-Friedrichs-Lewy condition, a numerical parameter for the numerical convergence of PDE solvers
• CI: continuous integration, automated tests that we perform before a proposed code-change is accepted; see PR
• CPU: central processing unit, we usual mean a socket or generally the host-side of a computer (compared to the accelerator, e.g. GPU)
• DOE: The United States Department of Energy, the largest sponsor of national laboratory research in the United States of America
• ECP: Exascale Computing Project, a U.S. DOE funding source that supports WarpX development
• ECT: Enlarged Cell Technique, an electromagnetic solver with accurate resolution of perfectly conducting embedded boundaries
• EB: embedded boundary, boundary conditions inside the simulation box, e.g. following material surfaces
• EM: electromagnetic, e.g. EM PIC
• ES: electrostatic, e.g. ES PIC
• FDTD: Finite-difference time-domain or Yee’s method, a class of grid-based finite-difference field solvers
• GPU: originally graphics processing unit, now used for fast general purpose computing (GPGPU); also called (hardware) accelerator
• **LDRD:** Laboratory Directed Research and Development, a *funding program in U.S. DOE laboratories* that kick-started ABLASTR development

• **LPA:** laser-plasma acceleration, historically used for laser-electron acceleration

• **LPI:** laser-plasma interaction

• **LWFA:** laser-wakefield acceleration (of electrons/leptons)

• **MR:** mesh-refinement

• **MVA:** magnetic-vortex acceleration (of protons/ions)

• **NERSC:** National Energy Research Scientific Computing Center, a supercomputing center located in Berkeley, CA (USA)

• **NSF:** the *National Science Foundation*, a large public agency in the United States of America, supporting research and education

• **OLCF:** Oak Ridge Leadership Computing Facility, a supercomputing center located in Oak Ridge, TN (USA)

• **OTP:** One-Time-Password; see 2FA

• **PDE:** *partial differential equation*, an equation which imposes relations between the various partial derivatives of a multivariable function

• **PIC:** *particle-in-cell*, the method implemented in WarpX

• **PR:** github pull request, a proposed change to the WarpX code base

• **PSATD:** pseudo-spectral analytical time-domain method, a spectral field solver with better numerical properties than FDTD solvers

• **PWFA:** plasma-wakefield acceleration

• **QED:** *quantum electrodynamics*

• **RPA:** radiation-pressure acceleration (of protons/ions), e.g. hole-boring (HB) or light-sail (LS) acceleration

• **RZ:** for the coordinate system r-z in cylindrical geometry; we use “RZ” when we refer to quasi-cylindrical geometry, decomposed in azimuthal modes (see details here)

• **TNSA:** target-normal sheet acceleration (of protons/ions)

### 8.1.2 Terms

- **accelerator:** depending on context, either a *particle accelerator* in physics or a *hardware accelerator* (e.g. GPU) in computing

- **AMReX:** C++ library for block-structured adaptive mesh-refinement, a primary dependency of WarpX

- **boosted frame:** a *Lorentz-boosted frame of reference* for a simulation

- **evolve:** this is a generic term to advance a quantity (same nomenclature in AMReX).
  For instance, `WarpX::EvolveE(dt)` advances the electric field for duration `dt`, `PhysicalParticleContainer::Evolve(...)` does field gather + particle push + current deposition for all particles in `PhysicalParticleContainer`, and `WarpX::EvolveEM` is the central `WarpX` function that performs 1 PIC iteration.

- **Frontier:** an Exascale supercomputer at OLCF

- **laser:** most of the time, we mean a laser pulse

- **openPMD:** Open Standard for Particle-Mesh Data Files, a community meta-data project for scientific data

- **Perlmutter:** a Berkeley Lab nobel laureate and a Pre-Exascale supercomputer at NERSC
8.2 Funding and Acknowledgements

WarpX is 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) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.

CEA-LIDYL actively contributes to the co-development of WarpX. As part of this initiative, WarpX also receives funding from the French National Research Agency (ANR - Plasm-On-Chip), the Horizon H2020 program and CEA.

We acknowledge all the contributors and users of the WarpX community who participate to the code quality with valuable code improvement and important feedback.
<table>
<thead>
<tr>
<th>B</th>
<th>built-in function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum.checksumAPI.evaluate_checksum()</td>
<td>199</td>
</tr>
<tr>
<td>read_sim_output.read_sim_output()</td>
<td>115</td>
</tr>
<tr>
<td>warpx_simf.run_warpx()</td>
<td>114</td>
</tr>
<tr>
<td>write_sim_input.write_sim_input()</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>built-in function, 199</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum.checksumAPI.evaluate_checksum()</td>
<td>199</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th>Filter (C++ class), 207</th>
</tr>
</thead>
<tbody>
<tr>
<td>filterCreateTransformFromFAB (C++ function)</td>
<td>215</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>MultiParticleContainer (C++ class), 210</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiParticleContainer::doQEDSchwinger (C++ function)</td>
<td>215</td>
</tr>
</tbody>
</table>

| N | NCIGodfreyFilter (C++ class), 207 |

<table>
<thead>
<tr>
<th>P</th>
<th>PhysicalParticleContainer::applyNCIFilter (C++ function), 207</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhysicalParticleContainer::Evolve (C++ function), 209</td>
<td></td>
</tr>
<tr>
<td>PhysicalParticleContainer::PartitionParticlesInBuffers (C++ function), 212</td>
<td></td>
</tr>
<tr>
<td>PhysicalParticleContainer::PushPX (C++ function), 211</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>built-in function, 115</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_sim_output.read_sim_output()</td>
<td>115</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th>WarpX::AddCurrentFromFineLevelandSumBoundary (C++ function), 207</th>
</tr>
</thead>
<tbody>
<tr>
<td>WarpX::RestrictCurrentFromFineToCoarsePatch (C++ function), 207</td>
<td></td>
</tr>
<tr>
<td>WarpX::SyncCurrent (C++ function), 207</td>
<td></td>
</tr>
<tr>
<td>warpx_simf.run_warpx()</td>
<td>built-in function, 114</td>
</tr>
<tr>
<td>WarpXParticleContainer (C++ class), 198, 209</td>
<td></td>
</tr>
<tr>
<td>write_sim_input.write_sim_input()</td>
<td>built-in function, 114</td>
</tr>
<tr>
<td>WarpX::ApplyFilterandSumBoundaryJ (C++ function), 207</td>
<td></td>
</tr>
<tr>
<td>WarpX::RestrictCurrentFromFineToCoarsePatch (C++ function), 207</td>
<td></td>
</tr>
<tr>
<td>WarpX::SyncCurrent (C++ function), 207</td>
<td></td>
</tr>
<tr>
<td>warpx_simf.run_warpx()</td>
<td>built-in function, 114</td>
</tr>
<tr>
<td>WarpXParticleContainer (C++ class), 198, 209</td>
<td></td>
</tr>
<tr>
<td>write_sim_input.write_sim_input()</td>
<td>built-in function, 114</td>
</tr>
</tbody>
</table>

WarpX::AddCurrentFromFineLevelandSumBoundary (C++ function), 207