WarpX Documentation

WarpX collaboration

Mar 31, 2020

Contents

Warning: This is an **alpha release** of WarpX. The code is still in active development. Robustness and performance may fluctuate at this stage. The input and output formats may evolve.

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 section *Theoretical background*.

In addition, WarpX is a highly-parallel and highly-optimized code and features hybrid OpenMP/MPI parallelization, advanced vectorization techniques and load balancing capabilities.

In order to learn to use the code, please see the sections below:

CHAPTER 1

Building/installing WarpX

WarpX can currently be built (and run) in two variants:

- as a compiled executable (run with the command line)
- as a Python package (run through a Python script)

Currently, for both of these options, the user needs to build the code from source.

1.1 Downloading the source code

Clone the source codes of WarpX, and its dependencies AMReX and PICSAR into one single directory (e.g. `warpx_directory`):

```
mkdir warpx_directory
cd warpx_directory
git clone https://github.com/ECP-WarpX/WarpX.git
git clone https://bitbucket.org/berkeleylab/picsar.git
git clone https://github.com/AMReX-Codes/amrex.git
```

Then switch to the branch `development` of AMReX

```
cd amrex/
git checkout development
cd ..
```

1.2 Compiling WarpX as an executable

`cd` into the directory `WarpX` and type

```
make -j 4
```

This will generate an executable file in the `Bin` directory.

Note: The compilation options are set in the file `GNUmakefile`. The default options correspond to an optimized code for 3D geometry. You can modify the options in this file in order to (for instance):

- Use 2D geometry
- Disable OpenMP
- Profile or debug the code
- Choose a given compiler

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:

```
make realclean
```

1.3 Installing WarpX as a Python package

Type

```
make -j 4 USE_PYTHON_MAIN=TRUE
```

or edit the GNUmakefile and set *USE_PYTHON_MAIN=TRUE*, and type

```
make -j 4
```

This will compile the code, and install the Python bindings as a package (named `pywarpx`) in your standard Python installation (i.e. in your `site-packages` directory). The note on compiler options from the previous section also holds when compiling the Python package.

In case you do not have write permissions to the default Python installation (e.g. typical on computer clusters), use the following command instead:

```
make -j 4 PYINSTALLOPTIONS=--user
```

In this case, you can also set the variable *PYTHONUSERBASE* to set the folder where *pywarpx* will be installed.

1.4 Advanced building instructions

1.4.1 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

```
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/
```

- Set `USE_PSATD=TRUE` when compiling:

```
make -j 4 USE_PSATD=TRUE
```

1.4.2 Building WarpX for Cori (NERSC)

For the Cori cluster at NERSC, you need to type the following command when compiling:

Note: In order to compile the code with a spectral solver, type

```
module load cray-fftw
```

before typing any of the commands below, and add `USE_PSATD=TRUE` at the end of the command containing `make`.

In order to compile for the **Haswell architecture**:

- with the Intel compiler

```
make -j 16 COMP=intel
```

- with the GNU compiler

```
module swap PrgEnv-intel PrgEnv-gnu
make -j 16 COMP=gnu
```

In order to compile for the **Knight's Landing (KNL) architecture**:

- with the Intel compiler

```
module swap craype-haswell craype-mic-knl
make -j 16 COMP=intel
```

- with the GNU compiler

```
module swap craype-haswell craype-mic-knl
module swap PrgEnv-intel PrgEnv-gnu
make -j 16 COMP=gnu
```

1.4.3 Building WarpX for Summit-dev (OLCF)

For the Summit-dev cluster at OLCF, use the following commands to download the source code, and switch to the correct branch:

```
mkdir warpx_directory
cd warpx_directory

git clone https://github.com/ECP-WarpX/WarpX.git
```

(continues on next page)

(continued from previous page)

```
cd WarpX
git checkout gpu
cd ..

git clone https://bitbucket.org/berkeleylab/picsar.git
cd picsar
git checkout gpu
cd ..

git clone https://github.com/AMReX-Codes/amrex.git
cd amrex
git checkout development
cd ..
```

Then, use the following set of commands to compile:

```
module load pgi
module load cuda
make
```

CHAPTER 2

Running WarpX as an executable

2.1 How to run a new simulation

After compiling the code, the WarpX executable is stored in the folder `warpx/Bin`. (Its name starts with *main* but depends on the compiler options.)

In order to run a new simulation:

- Create a **new directory**, where the simulation will be run.
- Copy the **executable** to this directory:

```
cp warpx/Bin/<warpx_executable> <run_directory>/warpx.exe
```

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.

- Add an **input file** in the directory.

This file contains the numerical and physical parameters that define the situation to be simulated. Example input files can be found in the section *Example input files*. The different parameters in these files are explained in the section *Input parameters*.

- **Run** the executable:

```
mpirun -np <n_ranks> ./warpx.exe <input_file>
```

where `<n_ranks>` is the number of MPI ranks used, and `<input_file>` is the name of the input file.

2.2 Example input files

This section allows you to **download input files** that correspond to different physical situations. For a definition of the different parameters that are set in these files, see the section *Input parameters*.

2.2.1 Beam-driven acceleration

- `2D case`
- `2D case in boosted frame`
- `3D case in boosted frame`

2.2.2 Laser-driven acceleration

- `2D case`
- `2D case in boosted frame`
- `3D case`

2.2.3 Plasma mirror

`2D case`

2.2.4 Uniform plasma

`2D case` `3D case`

2.3 Input parameters

Warning: This section is currently in development.

2.3.1 Overall simulation parameters

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

  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.)

2.3.2 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*)** When using mesh refinement, the number of refinement levels that will be used.

  Use 0 in order to disable mesh refinement.

- **geometry.is_periodic (*2 integers in 2D, 3 integers in 3D*)** Whether the boundary conditions are periodic, in each direction.

  For each direction, use 1 for periodic conditions, 0 otherwise.

- **geometry.prob_lo and geometry.prob_hi (*2 floats in 2D, 3 integers 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`).

- **warpx.fine_tag_lo and warpx.fine_tag_hi (*2 floats in 2D, 3 integers in 3D*; in meters) 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`).

2.3.3 Distribution across MPI ranks and parallelization

- **amr.max_grid_size (*integer*)** 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.

- **warpx.load_balance_int (*integer*)** How often WarpX should try to redistribute the work across MPI ranks, in order to have better load balancing (expressed in number of PIC cycles inbetween two consecutive attempts at redistributing the work). 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`).

- **warpx.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.

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

2.3.4 Math parser and user-defined constants

WarpX provides a math parser that reads expressions in the input file. It can be used to define the plasma density profile, the plasma momentum distribution or the laser field (see below *Particle initialization* and *Laser initialization*).

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 and `x` and `y` are variables. The factor `(x>0)` is *1* where *x>0* and *0* where *x<=0*. It allows the user to define functions by intervals. User-defined constants can be used in parsed functions only (i.e., `density_function(x,y,z)` and `field_function(x,y,t)`, see below). They are specified with:

- **constants.use_my_constants (*bool*)** Whether to use user-defined constants.

- **constants.constant_names** (*strings, separated by spaces*) A list of variables the user wants to define, e.g., `constants.constant_names = a0 n0`.
- **constants.constant_values** (*floats, sepatated by spaces*) Values for the user-defined constants., e.g., `constants.constant_values = 3. 1.e24`.

2.3.5 Particle initialization

- **particles.nspecies** (*int*) The number of species that will be used in the simulation.
- **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*) Whether to activate the FDTD Numerical Cherenkov Instability corrector.
- **particles.rigid_injected_species** (*strings*, **separated by spaces**) List of species injected using the rigid injection method. 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.
- **<species_name>.charge** (*float*) The charge of one *physical* particle of this species.
- **<species_name>.mass** (*float*) The mass of one *physical* particle of this species.
- **<species_name>.injection_style** (*string*) Determines how the particles will be injected in the simulation. 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`.
- **<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}.
  - `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. Note that using this density profile will turn `warpx.serialize_ics` to `1`, which may slow down the simulation.
- **<species_name>.momentum_distribution_type** (*string*) Distribution of the normalized momentum (*u=p/mc*) for this species. The options are:
  - `constant`: constant momentum profile. This requires additional parameters `<species_name>.ux`, `<species_name>.uy` and `<species_name>.uz`, the normalized momenta in the x, y and z direction respectively.
  - `gaussian`: gaussian momentum distribution in all 3 directions. This requires 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`.
  - `radial_expansion`: momentum depends on the radial coordinate linearly. This requires additional parameter `u_over_r` which is the slope.

  - `parse_momentum_function`: the momentum is given by a function in the input file. It requires additional arguments `<species_name>.momentum_function_ux(x,y,z)`, `<species_name>.momentum_function_uy(x,y,z)` and `<species_name>.momentum_function_uz(x,y,z)`, which gives the distribution of each component of the momentum as a function of space. Note that using this momentum distribution type will turn `warpx.serialize_ics` to `1`, which may slow down the simulation.

- **`<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_avance`** (*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>.do_backward_injection`** (*bool*) Inject a backward-propagating beam to reduce the effect of charge-separation fields when running in the boosted frame. See examples.
- **`warpx.serialize_ics`** (*0 or 1*) Whether or not to use OpenMP threading for particle initialization.

2.3.6 Laser initialization

- **`warpx.use_laser`** (*0 or 1*) Whether to activate the injection of a laser pulse in the simulation
- **`laser.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.position` and `laser.direction`.

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

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

  When running a **boosted-frame simulation**, provide the value of `laser.position` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame. Note that, in this case, the laser antenna will be moving, in the boosted frame.
- **`laser.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.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.direction` should be parallel to `warpx.boost_direction`, for now.

- **`laser.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:

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

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

- **`laser.wavelength` (*float*; in meters)** The wavelength of the laser in vacuum.

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

- **`laser.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.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.field_function(X,Y,t)`, which is a mathematical expression , e.g. `laser.field_function(X,Y,t) = "a0*X**2 * (X>0) * cos(omega0*t)"` where `a0` and `omega0` are a 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.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.parse_field_function=1`. Even though `laser.wavelength` and `laser.e_max` should be included in the laser function, they still have to be specified as they are used for numerical purposes.

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

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

- `laser.profile_duration` (*float* ; in seconds)

  The duration of the laser, defined as τ below:

  - For the `"gaussian"` profile:

    $$E(\boldsymbol{x}, t) \propto \exp\left(-\frac{(t - t_{peak})^2}{\tau^2}\right)$$

  - For the `"harris"` profile:

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

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

- **`laser.profile_waist` (*float* ; in meters)** The waist of the transverse Gaussian laser profile, defined as w_0 :

  $$E(\boldsymbol{x}, t) \propto \exp\left(-\frac{\boldsymbol{x}_\perp^2}{w_0^2}\right)$$

- **`laser.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.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.profile_focal_distance` in the laboratory frame, and use `warpx.gamma_boost` to automatically perform the conversion to the boosted frame.

- **`laser.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 (2014).

- **`laser.zeta` (*float*; in meters.seconds) optional (default *0.*)** Spatial chirp at focus in direction `laser.stc_direction`. See definition in Akturk et al., Opt Express, vol 12, no 19 (2014).

- **`laser.beta` (*float*; in seconds) optional (default *0.*)** Angular dispersion (or angular chirp) at focus in direction `laser.stc_direction`. See definition in Akturk et al., Opt Express, vol 12, no 19 (2014).

- **`laser.phi2` (*float*; in seconds**2) optional (default *0.*)** Temporal chirp at focus. See definition in Akturk et al., Opt Express, vol 12, no 19 (2014).

2.3.7 Numerics and algorithms

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

- **`warpx.use_filter` (*0 or 1*)** Whether to smooth the charge and currents on the mesh, after depositing them from the macroparticles. This uses a bilinear filter (see the sub-section **Filtering** in *Theoretical background*).

- **`algo.current_deposition` (*integer*)** The algorithm for current deposition:

  - `0`: Esirkepov deposition, vectorized
  - `1`: Esirkepov deposition, non-optimized
  - `2`: Direct deposition, vectorized
  - `3`: Direct deposition, non-optimized

  Warning: The vectorized Esirkepov deposition (`algo.current_deposition=0`) is currently not functional in WarpX. All the other methods (`1`, `2` and `3`) are functional.

- **`algo.charge_deposition` (*integer*)** The algorithm for the charge density deposition:

  - `0`: Vectorized version
  - `1`: Non-optimized version

- **`algo.field_gathering` (*integer*)** The algorithm for field gathering:

  - `0`: Vectorized version
  - `1`: Non-optimized version

- **`algo.particle_pusher` (*integer*)** The algorithm for the particle pusher:

  - `0`: Boris pusher
  - `1`: Vay pusher

- **algo.maxwell_fdtd_solver (*string*)** The algorithm for the FDTD Maxwell field solver:
  - `yee`: Yee FDTD solver
  - `ckc`: Cole-Karkkainen solver with Cowan coefficients (see Cowan - PRST-AB 16, 041303 (2013))
- **interpolation.nox, interpolation.noy, interpolation.noz (*integer*)** The order of the shape factors for the macroparticles, for the 3 dimensions of space. Lower-order shape factors result in faster simulations, but more noisy results,

  Note that the implementation in WarpX is more efficient when these 3 numbers are equal, and when they are between 1 and 3.
- **psatd.nox, psatd.noy, pstad.noz (*integer*) optional (default *16* for all)** The order of accuracy of the spatial derivatives, when using the code compiled with a PSATD solver.
- **psatd.ngroups_fft (*integer*)** The number of MPI groups that are created for the FFT, when using the code compiled with a PSATD solver. The FFTs are global within one MPI group and use guard cell exchanges in between MPI groups. (If `ngroups_fft` is larger than the number of MPI ranks used, than the actual number of MPI ranks is used instead.)
- **psatd.fftw_plan_measure (*0* or *1*)** Defines whether the parameters of FFTW plans will be initialized by measuring and optimizing performance (`FFTW_MEASURE` mode; activated by default here). If `psatd.fftw_plan_measure` is set to `0`, then the best parameters of FFTW plans will simply be estimated (`FFTW_ESTIMATE` mode). See this section of the FFTW documentation for more information.

2.3.8 Diagnostics and output

- **amr.plot_int (*integer*)** The number of PIC cycles inbetween two consecutive data dumps. Use a negative number to disable data dumping.
- **warpx.do_boosted_frame_diagnostic (*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.num_snapshots_lab (*integer*)** Only used when `warpx.do_boosted_frame_diagnostic` is `1`. The number of lab-frame snapshots that will be written.
- **warpx.dt_snapshots_lab (*float*, in seconds)** Only used when `warpx.do_boosted_frame_diagnostic` is `1`. The time interval inbetween the lab-frame snapshots (where this time interval is expressed in the laboratory frame).
- **warpx.plot_raw_fields (*0* or *1*) optional (default *0*)** By default, the fields written in the plot files are averaged on the nodes. When `` `warpx.plot_raw_fields `` is *1*, then the raw (i.e. unaveraged) fields are also saved in the plot files.
- **warpx.plot_raw_fields_guards (*0* or *1*)** Only used when `warpx.plot_raw_fields` is `1`. Whether to include the guard cells in the output of the raw fields.
- **warpx.plot_finepatch (*0* or *1*)** Only used when mesh refinement is activated and `warpx.plot_raw_fields` is `1`. Whether to output the data of the fine patch, in the plot files.
- **warpx.plot_crsepatch (*0* or *1*)** Only used when mesh refinement is activated and `warpx.plot_raw_fields` is `1`. Whether to output the data of the coarse patch, in the plot files.

2.3.9 Checkpoints and restart

WarpX supports checkpoints/restart via AMReX.

- **amr.check_int** (*integer*) The number of iterations between two consecutive checkpoints. Use a negative number to disable checkpoints.
- **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.

2.4 Profiling the code

2.4.1 Profiling with AMREX's built-in profiling tools

See this page in the AMReX documentation.

2.4.2 Profiling the code with Intel Advisor on NERSC

Follow these steps:

- Instrument the code during compilation

```
module swap craype-haswell craype-mic-knl
make -j 16 COMP=intel USE_VTUNE=TRUE
```

(where the first line is only needed for KNL)

- In your SLURM submission script, use the following lines in order to run the executable. (In addition to setting the usual `OMP` environment variables.)

```
module load advisor
export ADVIXE_EXPERIMENTAL=roofline
srun -n <n_mpi> -c <n_logical_cores_per_mpi> --cpu_bind=cores advixe-cl -collect␣
↪survey -project-dir advisor -trace-mpi -- <warpx_executable> inputs
srun -n <n_mpi> -c <n_logical_cores_per_mpi> --cpu_bind=cores advixe-cl -collect␣
↪tripcounts -flop -project-dir advisor -trace-mpi -- <warpx_executable> inputs
```

where `<n_mpi>` and `<n_logical_cores_per_mpi>` should be replaced by the proper values, and `<warpx_executable>` should be replaced by the name of the WarpX executable.

- Launch the Intel Advisor GUI

```
module load advisor
advixe-gui
```

(Note: this requires to use `ssh -XY` when connecting to Cori.)

CHAPTER 3

Running WarpX from Python

3.1 How to run a new simulation

After installing WarpX as a Python package, you can use its functionalities in a Python script to run a simulation.

In order to run a new simulation:

- Create a **new directory**, where the simulation will be run.
- Add a **Python script** in the directory.

This file contains the numerical and physical parameters that define the situation to be simulated. Example input files can be found in the section *Example input files*.

- **Run** the script with Python:

```
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.

3.2 Example input files

This section allows you to **download Python scripts** that correspond to different physical situations.

3.2.1 Beam-driven acceleration

- `Without mesh refinement`
- `With mesh refinement`

3.2.2 Laser-driven acceleration

- `Without mesh refinement`

CHAPTER 4

Visualizing the simulation results

4.1 Visualization with yt

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.

4.1.1 Installation

From the terminal:

```
pip install yt jupyter
```

or with the Anaconda distribution of python (recommended):

```
conda install -c atmyers yt
```

4.1.2 Visualizing the data

Once data ("plot files") 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('./plt00000/')
```

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 `Ex` in a slice orthogonal to `y` (`1`):

```
yt.SlicePlot( ds, 1, 'Ex' )
```

Alternatively, the data can be obtained as a numpy array.

For instance, in order to obtain the field *jz* (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

Particle data can be visualized using `yt.ParticlePhasePlot` (see the docstring here).

For instance, in order to plot the particles' `x` and `y` positions:

```
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:

```
ad = ds.all_data()
x = ad['particle_position_x'].to_ndarray()
```

4.1.3 Further information

A lot more information can be obtained from the yt documentation, and the corresponding notebook tutorials here.

4.2 Visualization with Visit

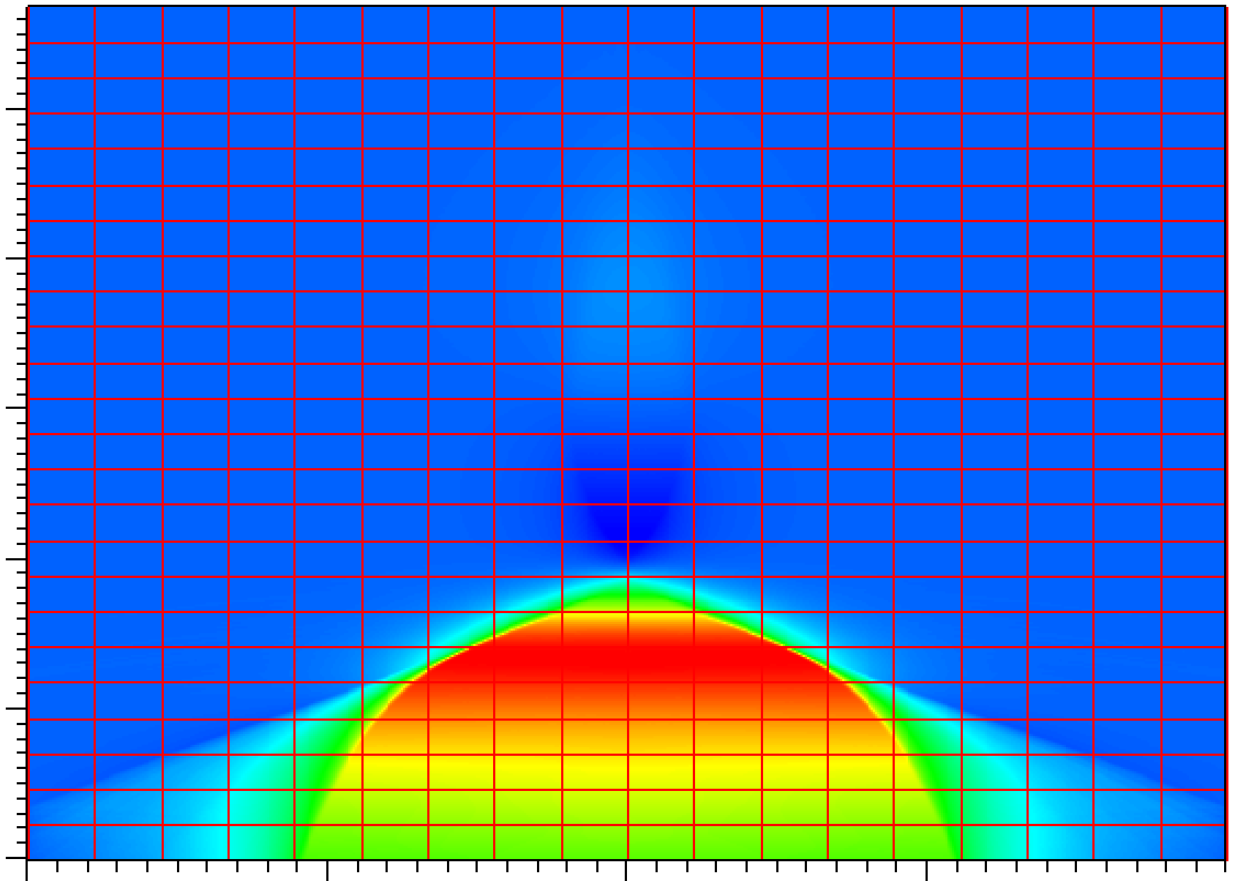
WarpX results can also be visualized by VisIt, an open source visualization and analysis software. VisIT can be downloaded and installed from https://wci.llnl.gov/simulation/computer-codes/visit.

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 clik 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

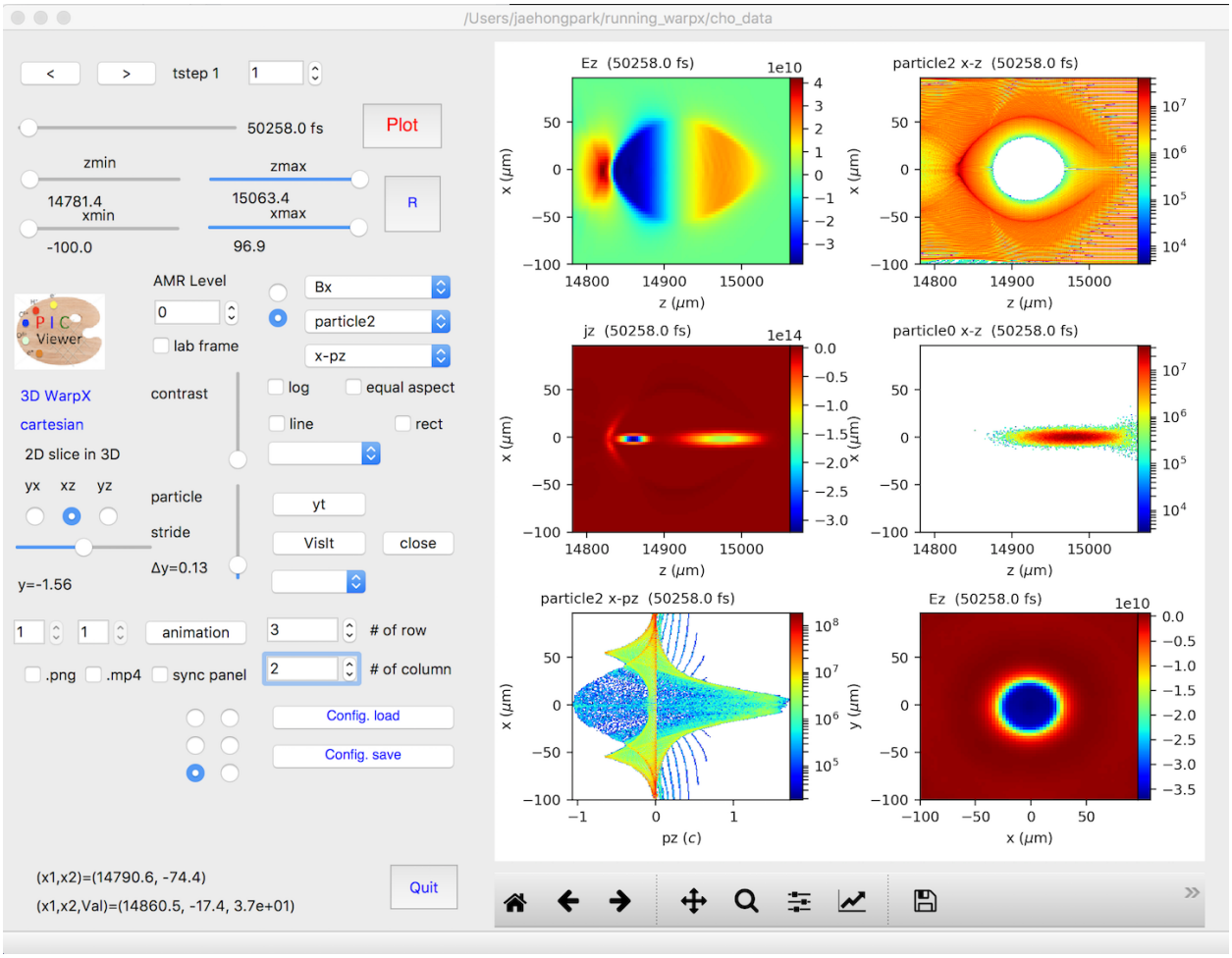
In 3D, you must apply the "Operators" -> "Slicing" -> "ThreeSlice", You can left-click and drag over the image to rotate the image to generate image you like.



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.

4.3 PyQt-based visualization GUI: PICViewer



The toolkit provides various easy-to-use functions for data analysis of Warp/WarpX simulations.

4.3.1 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)

4.3.2 Required software

- python 2.7 or higher: http://docs.continuum.io/anaconda/install.
- PyQt5

```
conda install pyqt
```

- h5py
- matplotlib
- numpy
- yt

```
pip install git+https://github.com/yt-project/yt.git --user
```

- numba

4.3.3 Installation

```
pip install picviewer
```

You need to install yt and PySide separately.

You can install from the source for the latest update,

```
pip install git+https://bitbucket.org/ecp_warpx/picviewer/
```

4.3.4 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*

4.3.5 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 ae located (*cd [your data folder]*) and type *picviewer* in the command line.

4.4 In situ Visualization with SENSEI

SENSEI is a light weight 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.

4.4.1 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
```

4.4.2 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 following 3 ParmParse parameters are used:

```
insitu.int = 2
insitu.start = 0
insitu.config = render_iso_catalyst_2d.xml
```

`insitu.int` turns in situ processing on or off and controls how often data is processed. `insitu.start` controls when in situ processing starts. `insitu.config` points to the SENSEI XML file which selects and configures the desired back end.

4.4.3 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 master branch.

To ease the burden of wrangling back end 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 back ends 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.

NERSC Cori

SENSEI is deployed at NERSC on Cori. The NERSC deployment includes the major back ends such as ParaView Catalyst, VisIt Libsim, and Python.

CHAPTER 5

Theoretical background

This page contains information on the algorithms that are used in WarpX.

Topics: