amrex Documentation

Release 18.01-dev

AMReX Team

Dec 18, 2018

CONTENTS:

AMReX is a software framework library containing all the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications. AMReX is freely available at https://github.com/AMReX-Codes/amrex.

AMReX Tutorials are a set of small stand-alone example codes that demonstrate how to use different parts of the AMReX functionality.

We are always happy to have users contribute to AMReX Tutorials as well as the AMReX source code. To contribute, issue a pull request against the development branch (details at https://help.github.com/articles/creating-a-pull-request/).

The amrex/Tutorials directory is broken into the following categories:

CHAPTER

ONE

TUTORIALS/AMR

CHAPTER

TWO

TUTORIALS/BASIC

The tutorials in amrex/Tutorials/Basic demonstrate the most fundamental operations supported by AMReX.

2.1 HelloWorld

HelloWorld_C and HellowWorld_F demonstrate the GNU Make system – with a sample Make.package and GNUmakefile – and the amrex::Initialize and amrex::Finalize functions.

In addition, in HelloWorld_C, the amrex::Print() operation, which only prints from the I/O processor, is used to print out the AMReX version (as defined by amrex::Version()) being used.

HelloWorld_F is a simple example of how to use the F_Interface routines, which are Fortran wrappers for the underlying C++ data strutures and iterators. Here, for example, rather than calling amrex::Print() in C++, we test on whether amrex_parallel_ioprocessor() is true, and if so, invoke the usual Fortran print call.

2.2 main

main_C and main_F introduce the following:

1. By default, AMReX initializes MPI and uses MPI_COMM_WORLD as its communicator. However, applications could choose to initialize MPI themselves and pass in an existing communicator.

2. By default, AMReX treats command line arguments as inputs parameters. The expected format of argv is

   executable inputs_file parm=value

   Here, *executable* is the filename of the executable, *inputs_file* is the file containing runtime parameters used to build AMReX ParmParse database, and *parm=value* is an input parameter that will override its value in *inputs_file*. Both *inputs_file* and *parm=value* are optional. At most one *inputs_file* is allowed. Howeer, there can be multiple *parm=value* s.

   The parsing of the command line arguments is performed in amrex::Initialize. Applications can choose to skip command line parsing. Applications can also provide a function that adds parameters to AMReX ParmParse database.

2.3 HeatEquation

The HeatEquation examples solve a 2D or 3D (determined by how you set DIM in the GNUmakefile) heat equation explicitly on a domain-decomposed mesh. This example is described in detail in the Basics chapter of the amrex Documentation

CHAPTER

THREE

TUTORIALS/BLUEPRINT

These tests, `AssignMultiLevelDensity` and `HeatEquation_EX1_C`, demonstrate how to convert AMReX Mesh data into an in-memory Conduit Mesh Blueprint description for consumption by the ALPINE Ascent in situ visualization and analysis tool. These are variants, respectively, of `amrex/Tests/Particles/AssignMultiLevelDensity` and `amrex/Tutorials/Basic/HeatEquation_EX1_C`.

For details about what mesh features are currently supported, see: amrex/Src/Base/AMReX_Conduit_Blueprint.H

These tests use the interfaces in Src/Base/AMReX_Conduit_Blueprint.H, which are built when USE_CONDUIT=TRUE. These tests' GNUmakefiles provide a template of how to enable and link Conduit and Ascent.

For more details about Conduit and Ascent, please see:

Conduit: Repo: https://github.com/llnl/conduit Docs http://llnl-conduit.readthedocs.io/en/latest/ Blueprint Docs: http://llnl-conduit.readthedocs.io/en/latest/blueprint.html

Ascent: Ascent Repo: http://github.com/alpine-dav/ascent Ascent Docs: http://ascent.readthedocs.io/en/latest/

(or ping Cyrus Harrison <cyrush@llnl.gov> or Matt Larsen <<larsen30@llnl.gov>)

CHAPTER

FOUR

TUTORIALS/CVODE

There are two CVODE tutorials in the `amrex/Tutorials/CVODE` directory, called `EX1` and `EX2`. `EX1` consists of a single ODE that is integrated with CVODE within each cell of a 3-D grid. It demonstrates how to initialize the CVODE solver, how to call the ODE right-hand-side (RHS), and, more importantly, how to *re*-initialize the solver between cells, which avoids allocating and freeing solver memory between each cell (see the call to `FCVReInit()` in the `integrate_ode.f90` file in the `EX1` directory.)

The `EX2` example demonstrates the slightly more complicated case of integrating a system of coupled ODEs within each cell. Similarly to `EX1`, it provides an RHS and some solver initialization. However, it also demonstrates the performance effect of providing an analytic Jacobian matrix for the system of ODEs, rather than requiring the solver to compute the Jacobian matrix numerically using a finite-difference approach. The tutorial integrates the same system of ODEs on the same 3-D grid, but in one sweep it instructs CVODE to use the analytic function that computes the Jacobian matrix, and in the other case, it does not, which requires CVODE to compute it manually. One observes a significant performance gain by providing the analytic Jacobian function.

See the CVODE section of the AMReX documentation for general instructions on how to include CVODE in an AMReX application.

CHAPTER

FIVE

TUTORIALS/EB

CHAPTER

SIX

TUTORIALS/GPU

CHAPTER

SEVEN

TUTORIALS/LINEARSOLVERS

There are three examples in the Tutorials/LinearSolvers directory.

`ABecLaplacian_C` demonstrates how to solve with cell-centered data in a C++ framework. This example shows how to use either hypre or PETSc as a bottom-solver (or to solve the equation at the finest level if you set the "max coarsening level" to 0.

`ABecLaplacian_F` demonstrates how to solve with cell-centered data using the Fortran interfaces.

`NodalPoisson` demonstrates how to solve with nodal data using the C++ framework.

CHAPTER

EIGHT

TUTORIALS/MUI

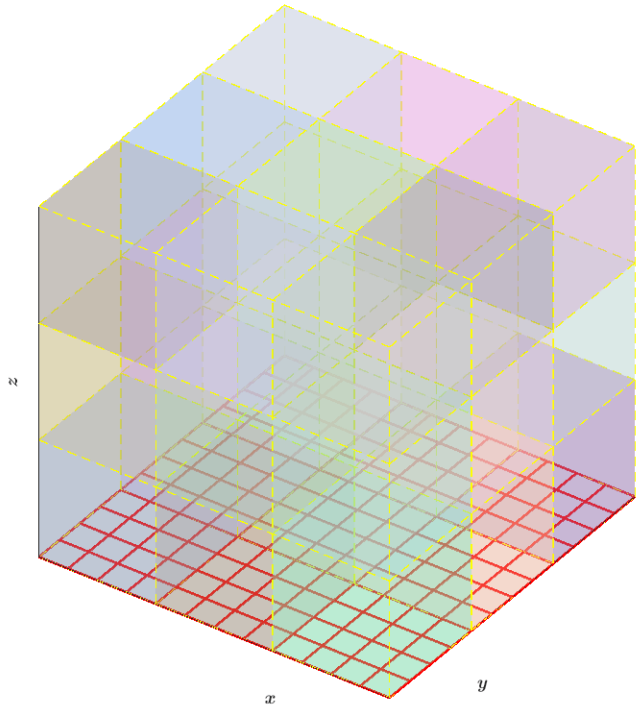
The goal of this tutorial is to incorporate the MxUI/MUI (Multiscale Universal Interface) framework into AMReX. This framework allows two separate executables to communicate with one another in parallel using MPI. In addition, this framework is adaptable for different geometries, in which the bounds of data one would like to send and/or receive can be specified using the `announce_send_span()` and `announce_recv_span()` commands.

In this tutorial, two different C++ codes are built separately. Each has different spatial dimensions: one is built in 3D (`AMREX_SPACEDIM = 3`), and the other in 2D (`AMREX_SPACEDIM = 2`). Each code is compiled separately within their respective "exec" directories `Exec_01` & `Exec_02`}, after which the two executables are run together using the following command, specifying the number of MPI processes to designate to each executable:

```
$ mpirun -np N1 ../Exec_01/main3d.gnu.MPI.ex inputs
: -np n2 ../Exec_02/main2d.gnu.MPI.ex inputs
```

on a single line within the `Exec_coupled` directory. `N1` and `n2` are the number of MPI ranks designated for each executable, respectively. Each executable is given the same inputs file within `Exec_coupled`. Input variables `max_grid_size_3d` and `max_grid_size_2d` determine the respective grid sizes for 3D and 2D. As far as I am aware, the code works for any AMReX grid structure. Details of how to build and run the code are contained in the script `cmd_mpirun`.

The figure below shows one possible grid structure of the 2D (red grid) and 3D (multicolored blocks) setup.



MUI interface: 2D and 3D grid setup

The 3D code initializes a 3D MultiFab (Note: with no ghost cells), and sends a 2D slice of this data at the $k = 0$ location to the 2D executable, which stores the data in a 2D MultiFab, multiplies the data by a constant, and sends the modified platter back to the 3D executable. Finally, the 3D executable receives the modified data and places it back into the 3D MultiFab, at $k = 0$.

The 2D, original 3D, and modified 3D data are all written to separate plot files, which can be visualized using software such as Amrvis.

Although our code does not include this, it would be possible to pair an AMReX code with code that is outside of the AMReX framework, because each code is compiled separately. For example, using the `announce_send_span()` and `announce_recv_span()` commands, MUI would be able to determine the overlap between the two regions to correctly exchange the data, even if the two grid structures differ.

CHAPTER

NINE

TUTORIALS/PARTICLES

There are two tutorials in amrex/Tutorials/Particles that demonstrate the basic usage of AMReX's particle data structures.

9.1 ElectrostaticPIC

This tutorial demonstrates how to perform an electrostatic Particle-in-Cell calculation using AMReX. The code initializes a single particle in a conducting box (i.e. Dirichlet zero boundary conditions) that is slightly off-center in one direction. Because of the boundary conditions, the particle sees an image charge and is accelerated in this direction.

The code is currently set up to use one level of static mesh refinement. The charge density, electric field, and electrostatic potential are all defined on the mesh nodes. To solve Poisson's equation, we use AMReX's Fortran-based multigrid solver. The Fortran routines for performing charge deposition, field gathering, and the particle push are all defined in `electrostatic_pic_2d.f90` and `electrostatic_pic_3d.f90` for 2D and 3D, respectively.

The particle container in this example using a Struct-of-Arrays layout, with `1 + 2*BL_SPACEDIM` real components to store the particle weight, velocity, and the electric field interpolated to the particle position. To see how to set up such a particle container, see `ElectrostaticParticleContainer.H`.

9.2 NeighborList

This tutorial demonstrates how to have AMReX's particles undergo short-range collisions with each other. To facilite this, a neighbor list data structure is created, in which all of the partners that could potentially collide with a given particle are pre-computed. This is done by first constructing a cell-linked list, and then looping over all 27 neighbor cells to test for potential collision partners. The Fortran subroutine `amrex_compute_forces_nl` defined in `neighbor_list_2d.f90` and `neighbor_list_3d.f90` demonstrates how to loop over the resulting data structure.

The particles in this example store velocity and acceleration in addition to the default components. They are initially placed at cell centers and given random velocities. When a particle reaches the domain boundary, it is specularly reflected back into the domain. To see how the particle data structures are set up, see `NeighborListParticleContainer.cpp`.

CHAPTER

TEN

TUTORIALS/SDC

10.1 IMEX_Advec_Diff_C

10.2 MISDC_ADR_2d

This tutorial presents an example of using a "multi-implicit" spectral deferred corrections (MISDC) integrator to solve a simple scalar advection-diffusion-reaction equation in two dimensions. Both diffusion and reaction terms are treated implicitly but solved for independently in an operator splitting fashion. The advection is treated explicitly. The relative strengths of the three terms can be adjusted by changing the coefficients a,d, and r in inputs_2d.

The advection operator is a 4th-order centered difference in flux form. The diffusion operator is a 2nd order discretization of the Laplacian, and the implicit diffusion solve is done using multigrid. The "reaction" term here is just a simple linear damping hence the implicit solve is trivial. See the routines in functions_2d.f90 for the code that evaluates the rhs terms.

The simple form of the equation allows for an exact solution of the PDE in a periodic geometry. There is a flag called "plot_err" in main.cpp, which if set equal 1 will cause the code to output the error in the solution for plotting. If the advection term is omitted (a=0), then an exact solution to the method of lines ODE is computed and used to compute the error. Hence the error in this case will scale in dt with the order of the time integrator.

This code can also be run as an IMEX advection-diffusion example simply by setting Npieces=2 in main.cpp. This should also be equivalent to setting r=0.

CHAPTER

ELEVEN

TUTORIALS/SENSEI

CHAPTER

TWELVE

TUTORIALS/SWFFT

This Tutorial demonstrates how to call the SWFFT wrapper to the FFTW3 solver.

Note that the SWFFT source code was developed by Adrian Pope and colleagues and is available at:

https://xgitlab.cels.anl.gov/hacc/SWFFT

Please refer to the AMReX documentation for a brief explanation of how the SWFFT redistributes data into pencil grids.

AMReX contains two SWFFT tutorials, `SWFFT_poisson` and `SWFFT_simple`:

- `SWFFT_poisson` tutorial: The tutorial found in `amrex/Tutorials/SWFFT/SWFFT_poisson` solves a Poisson equation with periodic boundary conditions. In it, both a forward FFT and reverse FFT are called to solve the equation, however, no reordering of the DFT data in k-space is performed.
- `SWFFT_simple` tutorial: This tutorial: `amrex/Tutorials/SWFFT/SWFFT_simple`, is useful if the objective is to simply take a forward FFT of data, and the DFT's ordering in k-space matters to the user. This tutorial initializes a 3D or 2D `MultiFab`, takes a forward FFT, and then redistributes the data in k-space back to the "correct," 0 to 2π, ordering. The results are written to a plot file.

12.1 SWFFT_poisson

In this test case we set up a right hand side (rhs), call the forward transform, modify the coefficients, then call the backward solver and output the solution to the discrete Poisson equation.

To build the code, type 'make' in `amrex/Tutorials/SWFFT/SWFFT_poisson`. This will include code from `amrex/Src/Extern/SWFFT` and you will need to link to the FFT solvers themselves (on NERSC's Cori machine, for example, you would need to "module load fft")

To run the code, type 'main3d.gnu.MPI.ex inputs' in this directory

To visualize the output, set the bool write_data to true, then use amrvis3d (source available at https://github.com/AMReX-Codes/Amrvis):

amrvis3d -mf RHS SOL_EXACT SOL_COMP

to visualize the rhs, the exact solution and the computed solution.

The max norm of the difference between the exact and computed solution is also printed.

For instructions on how to take a forward FFT only using SWFFT, please refer to *SWFFT_simple*.

12.2 SWFFT_simple

This tutorial initializes a 3D or 2D `MultiFab`, takes a forward FFT, and then redistributes the data in k-space back to the "correct," 0 to 2π, ordering. The results are written to a plot file.

In a similar fashion to the `SWFFT_poisson` tutorial:

To build the code, type 'make' in `amrex/Tutorials/SWFFT/SWFFT_simple`. This will include code from `amrex/Src/Extern/SWFFT` and you will need to link to the FFT solvers themselves (on NERSC's Cori machine, for example, you would need to "module load fft")

To run the code, type 'main*.ex inputs.oneGrid' in this directory to run the code in serial. To run the code in parallel, type 'mpiexec -n $N main*.ex inputs.multipleGrids' instead, where `N` holds the number of MPI processes (equal to the number of grids). `run_me_2d` and `run_me_3d` also provide examples of how to run the code.

Use amrvis2d or amrvis3d to visualize the output (source available at https://github.com/AMReX-Codes/Amrvis):

amrvis${dims}d plt_fft*

where `dims` specifies `AMREX_SPACEDIM`. The DFT of the data and the original data are labeled as `FFT_of_phi` and `phi` within the plot file.

The *SWFFT_poisson* tutorial provides an example of solving a Poisson equation using a discrete spectral method, in which a forward and reverse FFT of a `MultiFab` are computed.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

For a pdf version of this documentation, click `here`.