amrex Documentation

Release 21.03-dev

AMReX Team

Feb 02, 2021

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.

There is extensive documentation for AMReX at Welcome to AMReX's documentation

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

For each of these tutorials, plotfiles are generated that can be viewed with amrvis2d / amrvis3d (CCSE's native vis / spreadsheet tool, downloadable separately from ccse.lbl.gov) or with VisIt.

1.1 Advection_AmrCore

Advection_AmrCore: This tutorial contains an AMR advection code that advects a single scalar field with a velocity field that is specified on faces. It is written entirely in C++, can be built in 2D or 3D and run with the same inputs file,

It is an AMReX-based code designed to run in parallel using MPI+X, where X may be OMP for multicore machines and CUDA for hybrid CPU/GPU architectures.

This example uses source code from the amrex/Src/Base, Boundary, and AmrCore directories.

Notably, this example does not use source code from amrex/Src/Amr (see the tutorial Advection_AmrLevel).

The directory Exec includes a makefile and a sample inputs file.

1.2 Advection_AmrLevel

Advection_AmrLevel: This tutorial contains an AMR advection code that advects a single scalar field with a velocity field that is specified on faces.

It is an AMReX based code designed to run in parallel using MPI/OMP.

This example uses source code from the amrex/Src/Base, Boundary, AmrCore, and Amr directories.

The directories Exec/SingleVortex and Exec/UniformVelocity each include a makefile and a sample inputs file.

1.3 Advection_F

This code advects a single scalar field with a velocity field that is specified on faces.

It is a AMReX based code designed to run in parallel using MPI/OMP. It uses the Fortran interfaces of AMReX.

The directory Exec/SingleVortex includes a makefile and a sample inputs file.

1.4 Advection_octree_F

This code advects a single scalar field with a velocity field that is specified on faces.

It is a AMReX based code designed to run in parallel using MPI/OMP. It uses the Fortran interfaces of AMReX. The grids have an octree structure with a grid size of 8. No subcycling is used.

The directory Exec/SingleVortex includes a makefile and a sample inputs file.

CHAPTER

TWO

TUTORIALS/BASIC

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

2.1 HelloWorld

HelloWorld_C and HelloWorld_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 Build_with_libamrex

This tutorial builds on the main_C example and demonstrates how to build the executable when we want to link local files with the pre-built amrex library (libamrex.a) that has been installed elsewhere. We separate main.cpp from the main_C example into two separate files (main.cpp and test_parameters.cpp), replace MyAmr.H by MyParams.H and add a Fortran file my_func.f90. The GNUmakefile here assumes that you have already built the AMReX library; for instructions on how to do that see the **Building_libamrex_** chapter.

2.4 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/EB

amrex/Tutorials/EB/CNS is an AMR code for solving compressible Navier-Stokes equations with the embedded boundary approach.

amrex/Tutorials/EB/Poisson is a single-level code that is a proxy for solving the electrostatic Poisson equation for a grounded sphere with a point charge inside.

amrex/Tutorials/EB/MacProj is a single-level code that computes a divergence-free flow field around a sphere. A MAC projection is performed on an initial velocity field of (1,0,0).

CHAPTER

FIVE

TUTORIALS/FORKJOIN

There are two examples in the Tutorials/ForkJoin directory.

ForkJoin/Simple: demonstrates how to construct a `ForkJoin` object, specify configuration parameters, register MultiFabs with different access patterns, and invoke the `ForkJoin::fork_join()` operation.

ForkJoin/MLMG: demonstrates how to do more advanced fork-join operations, including nested fork-joins, heterogeneous tasks, customized MultiFab component splitting, and reusing `ForkJoin` objects for multiple invocations.

5.1 General Concepts

An AMReX program consists of a set of MPI ranks cooperating together on distributed data. Typically, all of the ranks in a job compute in a bulk-synchronous, data-parallel fashion, where every rank does the same sequence of operations, each on different parts of the distributed data.

The ForkJoin functionality described here allows the user to divide the job's MPI ranks into subgroups (i.e. *fork*) and assign each subgroup an independent task to compute in parallel with each other. After all of the forked child tasks complete, they synchronize (i.e. *join*), and the parent task continues execution as before.

The fork-join operation can also be invoked in a nested fashion, creating a hierarchy of fork-join operations, where each fork further subdivides the ranks of a task into child tasks. This approach enables heterogeneous computation and reduces the strong scaling penalty for operations with less inherent parallelism or with large communication overheads.

The fork-join operation is accomplished by:

a) redistributing MultiFab data so that **all** of the data in each registered MultiFab is visible to ranks within a subtask, and

b) dividing the root MPI communicator into sub-communicators so that each subgroup of ranks in a tasks will only synchronize with each other during subtask collectives (e.g. for `MPI_Allreduce`).

When the program starts, all of the ranks in the MPI communicator are in the root task.

5.2 ForkJoin/Simple

The main function in this tutorial is in `MyTest.cpp:runTest()`. It does the following things:

1. Create a `ForkJoin` object: the constructor takes the number of tasks to split the calling (in this case, root) task. This version of the constructor will divide the ranks in the calling (parent) task evenly across the spawned (child) tasks. To allow uneven distribution of ranks across tasks, there are other versions of the `ForkJoin` constructor that allow the user to specify the number (or percent) of ranks to include in each of the subtasks.

2. Set the verbosity flag and task output directory: `ForkJoin::set_verbose()` and `ForkJoin::set_task_output_dir()` are used to set each these parameters, overriding their default

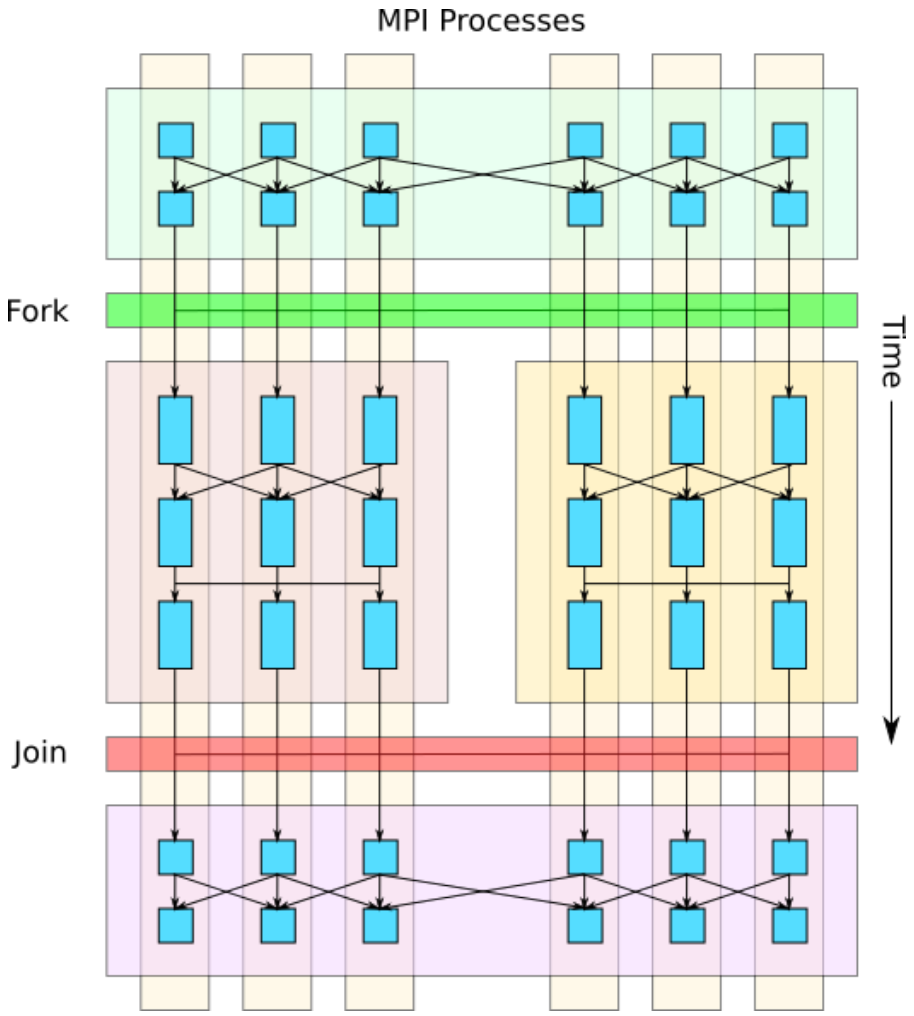


Fig. 5.1: Example of a fork-join operation where the parent task's MPI processes (ranks) are split into two independent child tasks that execute in parallel and then join to resume execution of the parent task.

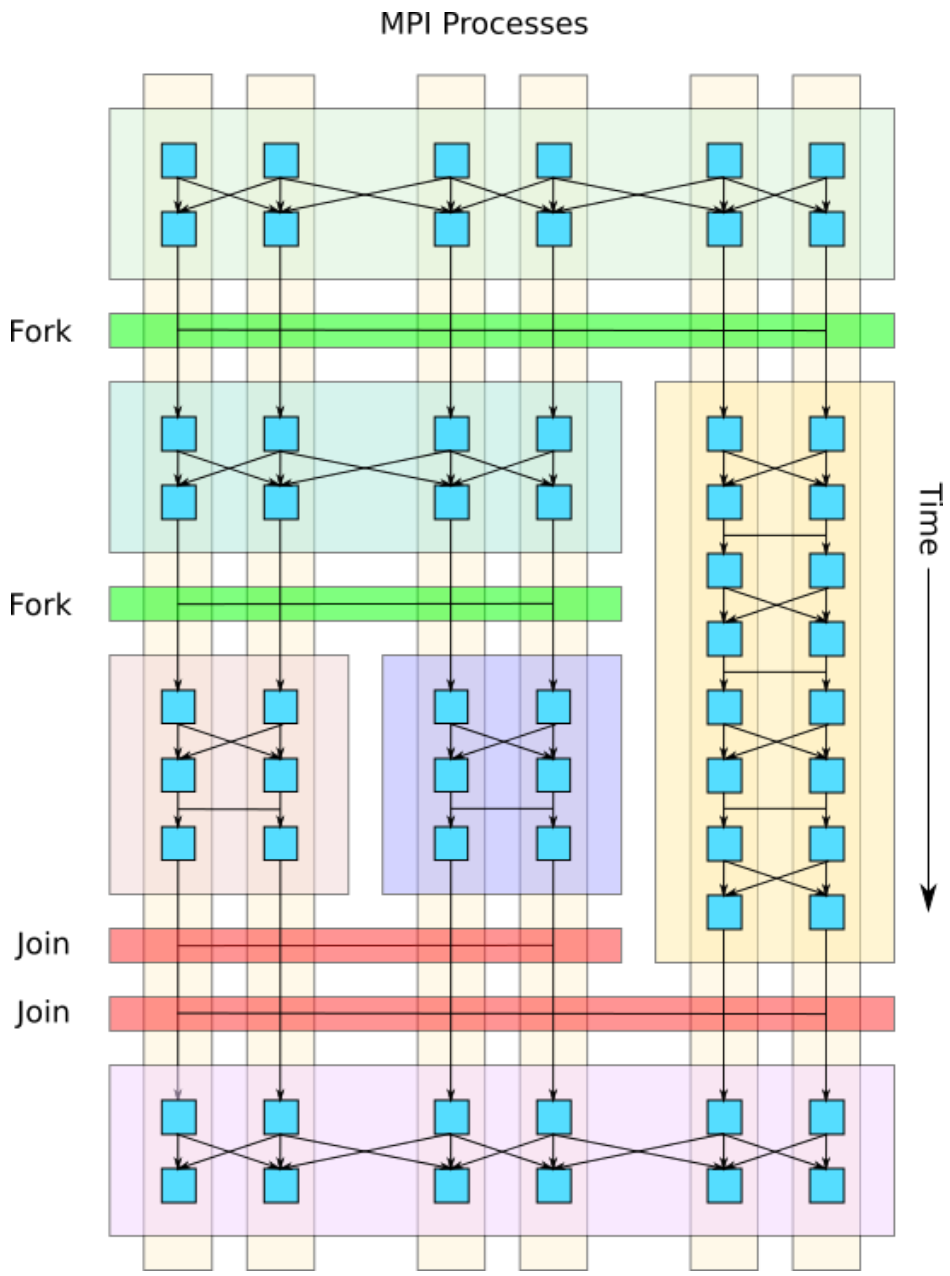


Fig. 5.2: Example of nested fork-join operations where a child task is further split into more subtasks.

values. Since the forked tasks run in parallel, their output to stdout is interleaved and may be difficult to read. By specifying a task output directory, the output from each task is written to its own file (in addition to stdout).

3. Register three MultiFab data structures: The `ForkJoin` object needs to know what data will be utilized within the spawned subtasks and how they will be accessed. For each MultiFab that will be accessed within the subtasks, there are two main parameters that need to be specified: Strategy and Intent. Strategy describes whether the MultiFab will be `duplicate` across all tasks, `split` (component-wise) across the subtasks, or accessed in only a `single` subtask. Intent describes whether the data is an input and/or output to the forked subtasks, and controls whether the data is copied in and/or out of the subtask from the calling task.

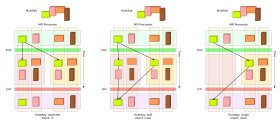


Fig. 5.3: Examples of how a MultiFab can be registered for a fork-join operation with varying Strategy and Intent.

During registration, the number of ghost cells in each dimension is also specified, along with the ID of the owner task in the case that `Strategy == single`. No data is actually copied during the call to `reg_mf()` – the MultiFab is only registered to be copied later when the `fork_join()` call is invoked.

4. Invoke the `fork_join()` operation, calling `myFunction` in every task: The `fork_join()` function launches the passed function (or lambda) on all of the spawned tasks. The passed function must take a single argument: a reference to the managing `ForkJoin` object, which can be queried for the subtask's ID, references to the registered MultiFabs, and other metadata such as the component bounds of a registered MultiFab. The tutorial's `myFunction` demonstrates these capabilities.

5.3 ForkJoin/MLMG

This tutorial demonstrates some more advanced fork-join usage:

1. Nested fork-join: `top_fork()` invokes the first level fork-join, which assigns one rank to the task 0 and the rest of the ranks to task 1 via the constructor: `ForkJoin fj(Vector<int> {1, proc_n - 1});`. Task 1 then calls `fork_solve()`, which further forks the task into sub-tasks.

2. Passing a lambda function to `fork_join()` and heterogeneous tasking: In `top_fork()`, we pass a lambda that takes the ForkJoin object reference as an argument. The ForkJoin object can be queried for the task ID, which is used to dispatch to different tasks for heterogeneous task execution.

3. Custom component splitting: if a MultiFab is registered with `Strategy == split`, then all the components of the MultiFab are split as evenly as possible across the tasks. In some cases, it may be desirable to either omit some components entirely or split the components in an uneven fashion. In `fork_solve`, we demonstrate how to specify a custom component split across the tasks by using the `modify_split` member function of the ForkJoin object after a MultiFab has been registered. The `modify_split` function takes a `Vector` of `ComponentSet` objects, each specifying the custom range of components to be passed to the task. In this example, we omit the first component from being passed to the child subtasks.

4. Reusing ForkJoin objects: if several successive fork-join operations are required with the same subranks and MultiFab access pattern, we can reuse the ForkJoin object across multiple invocations. Reusing the ForkJoin object avoids unnecessary overhead of recreating the forked data structures and metadata associated with the operation. The `fork_solve()` function demonstrates this capability by invoking `fork_join()` for two iterations.

CHAPTER

SIX

TUTORIALS/GPU

The tutorials in `amrex/Tutorials/GPU` demonstrate the implementation of AMReX's GPU toolkit as well as provide GPU ported versions of CPU tutorials to help applications convert to GPUs.

6.1 Your first AMReX GPU application

This is a step-by-step guide to preparing, compiling and running your first AMReX GPU program. This guide will use `Tutorials/GPU/Launch`, and instructions will focus on ORNL's systems:

1. Before compiling, the `pgi` and `cuda` software must be available. On ORNL systems, the modules can be loaded directly by typing:

```
module load pgi cuda
```

2. Go to `Tutorials/GPU/Launch` to compile the executable. Compile with `make USE_CUDA=TRUE COMP=pgi USE_MPI=TRUE USE_OMP=FALSE`, or edit the `GNUmakefile` to match this configuration and run `make`. This should result in an executable: `main3d.pgi.MPI.CUDA.ex`.
3. On Summit systems, this executable can be submitted by using one of the run scripts found in `Tutorials/GPU`. `run.script` can be used to run on Summitdev, and `run.summit` can be used for Summit. To change the number of ranks and GPUs used in the simulation, change the number of resource sets, `n` in the `jsrun` line. For the first `Launch` tutorial, use `n=1` and set `INPUTS=""` because no input file is used in this example.

When ready, submit the job script (on Summit: `bsub run.script`). Congratulations! You've accelerated AMReX using GPUs!

6.2 Launch

Launch shows multiple examples of how GPU work can be offloaded using the tools available in AMReX. It includes examples of multiple AMReX macro launch methods, launching a Fortran function using CUDA and launching work using OpenACC and OpenMP offloading. This tutorial will be regularly updated with AMReX's preferred GPU launch methodologies.

6.3 CNS

CNS is a direct GPU port of the `Tutorials/EB/CNS` tutorial.

6.4 AmrCore

AmrCore is a direct GPU port of the `Tutorials/Amr/Advection_AmrCore` tutorial that advects a single scalar field with a velocity field specified on faces, using strategies similar to HeatEquation and CNS.

CHAPTER

SEVEN

TUTORIALS/LINEARSOLVERS

There are five 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 set up and solve a variable coefficient Poisson equation with the rhs and solution data on nodes.

`MAC_Projection_EB` demonstrates how to set up and perform an EB-aware MAC projection of a specified velocity field on a staggered grid around nine cylindrical objects.

`Nodal_Projection_EB` demonstrates how to set up and perform an EB-aware nodal projection of a specified cell-centered velocity field around nine cylindrical objects.

`MultiComponent` demonstrates how to solve using a multi-component operator with nodal data. The operator is of the form $D(\phi)_i = \alpha_{ij} \nabla^2 \phi_j$, where ϕ is the multi-component solution variable and α_{ij} is a matrix of coefficients. The operator (`MCNodalLinOp`) is implemented using the "reflux-free" method.

CHAPTER

EIGHT

TUTORIALS/MUI

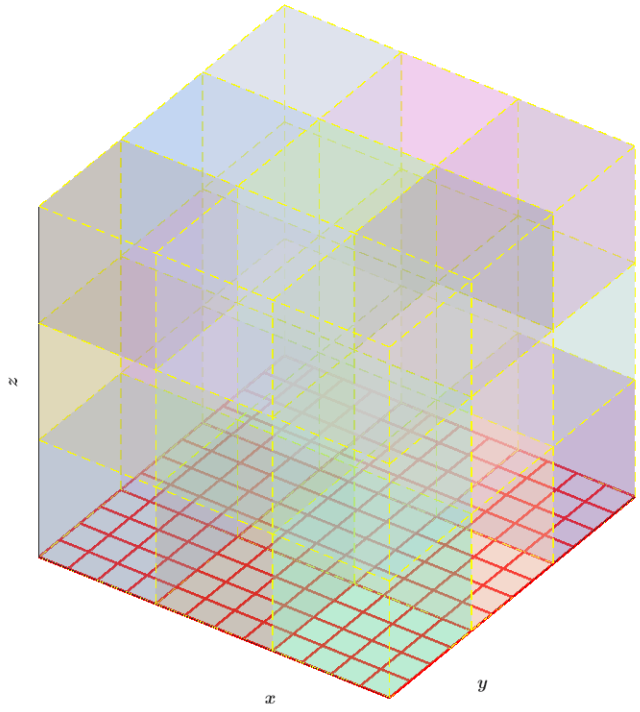
The goal of this tutorial is to incorporate the MxUI/MUI (Multiscale Universal Interface) framework into AMReX. This framework allows two separate executables to communicate with one another in parallel using MPI. In addition, this framework is adaptable for different geometries, in which the bounds 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 several 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 ElectromagneticPIC

This tutorial shows how to perform an electromagnetic particle-in-cell calculation using AMReX. Essentially, this is a mini-app version of the WarpX application code. The electric fields, magnetic fields, and current densities are stored using the staggered Yee grid, and it solves Maxwell's Equations using the finite-difference time domain method.

This tutorial also demonstrates how to offload calculations involving particle data onto the GPU using OpenACC. To compile with GPU support, use the pgi compiler, and set `USE_ACC = TRUE`, and `USE_CUDA = TRUE`, `USE_OMP = FALSE`.

You can choose between two problem types by toggling the `problem_type` parameter in the provided inputs file. Choosing the uniform plasma setup provides a nearly perfectly load balanced problem setup that is useful for performance testing. Choosing the Langmuir wave problem will automatically compare the simulated fields to the exact solution.

Currently, this tutorial does not use mesh refinement.

9.3 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`.

The file called `inputs` can be used to run this tutorial with a single level, and `inputs.mr` sets up a run with static mesh refinement.

9.4 CellSortedParticles

Sometimes, it's useful to sort particles at a finer granularity than grids or tiles. In this Tutorial, each cell contains a list of particle indices that tell you which particles belong to that cell. This is useful, for example, in Direct Simulation Monte Carlo calculations, where you want to potentially interact particles that are in the same cell as each other. Every time the particles move, we check to see whether it's still in the same cell or not. If it isn't, we mark the particle as unsorted. We then call Redistribute() as normal, and then insert the unsorted particles into the proper cells. Care is taken so that, if the Redistribute call changes the order of the particles in the Container, the indices in the cell lists are updated accordingly.

This Tutorial is currently single-level only.

CHAPTER

TEN

TUTORIALS/SDC

10.1 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/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 at SWFFT 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.

11.1 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*.

11.2 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

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

The copyright notice of AMReX is included in the AMReX home directory as README.md.

Your use of this software is under the 3-clause BSD license – the license agreement is included in the AMReX home directory as license.txt.

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