amrex Documentation

Release 23.03-dev

AMReX Team

Feb 10, 2023

CONTENTS:

AMReX is a software framework containing all the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications. AMReX is freely available on Github.

AMReX is developed at LBNL, NREL, and ANL as part of the Block-Structured AMR Co-Design Center in DOE's Exascale Computing Project.

All of AMReX's development is done in the GitHub repository under the development branch; anyone can see the latest updates. A monthly release is tagged at the beginning of each month.

We are always happy to have users contribute to the AMReX source code. To contribute, issue a pull request against the development branch (details here). Any level of changes are welcomed: documentation, bug fixes, new test problems, new solvers, etc. To obtain help, simply post a discussion or an issue on the AMReX GitHub webpage.

To learn AMReX there are walk-through guides and small stand-alone example codes that demonstrate how to use different parts of the AMReX functionality. Extensive documentation is available at AMReX Guided Tutorials and Example Codes.

Besides this documentation, there is API documentation generated by Doxygen.

Documentation on migration from BoxLib is available in the AMReX repository at Docs/Migration.

CHAPTER

ONE

AMREX INTRODUCTION

AMReX is a publicly available software framework designed for building massively parallel block-structured adaptive mesh refinement (AMR) applications.

Key features of AMReX include:

- C++ and Fortran interfaces
- 1-, 2- and 3-D support
- Support for cell-centered, face-centered, edge-centered, and nodal data
- Support for hyperbolic, parabolic, and elliptic solves on hierarchical adaptive grid structure
- Optional subcycling in time for time-dependent PDEs
- Support for particles
- Support for embedded boundary (cut cell) representations of complex geometries
- Parallelization via flat MPI, OpenMP, hybrid MPI/OpenMP, hybrid MPI/(CUDA or HIP or DPC++), or MPI/MPI
- Parallel I/O
- Plotfile format supported by AmrVis, VisIt, ParaView, and yt.

AMReX is developed at LBNL, NREL, and ANL as part of the Block-Structured AMR Co-Design Center in DOE's Exascale Computing Project.

CHAPTER

TWO

GETTING STARTED

In this chapter, we will walk you through two simple examples. It is assumed here that your machine has GNU Make, Python, GCC (including gfortran), and MPI, although AMReX can be built with CMake and other compilers.

2.1 Downloading the Code

The source code is available at https://github.com/AMReX-Codes/amrex. The GitHub repo is our central repo for development. The development branch includes the latest state of the code, and it is tagged as a release on a monthly basis with version number YY.MM (e.g., 17.04). The MM part of the version is incremented every month, and the YY part every year. Bug fix releases are tagged with YY.MM.patch (e.g., 17.04.1).

AMReX can also be obtained using Spack (https://spack.io/). Assuming you have Spack installed, simply type, `spack install amrex`. For more information see the *Spack* section in Building AMReX.

2.2 Example: Hello World

The source code of this example is at `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C/` and is also shown below.

```
#include <AMReX.H>
#include <AMReX_Print.H>

int main(int argc, char* argv[])
{
    amrex::Initialize(argc,argv);
    amrex::Print() << "Hello world from AMReX version "
                   << amrex::Version() << "\n";
    amrex::Finalize();
}
```

The main body of this short example contains three statements. Usually the first and last statements for the `int main(...)` function of every program should be calling `amrex::Initialize` and `amrex::Finalize`, respectively. The second statement calls `amrex::Print` to print out a string that includes the AMReX version returned by the `amrex::Version` function. The example code includes two AMReX header files. Note that the name of all AMReX header files starts with `AMReX_` (or just AMReX in the case of AMReX.H). All AMReX C++ functions are in the `amrex` namespace.

2.2.1 Building the Code

You build the code in the `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C/` directory. Typing `make` will start the compilation process and result in an executable named `main3d.gnu.DEBUG.ex`. The name shows that the GNU compiler with debug options set by AMReX is used. It also shows that the executable is built for 3D. Although this simple example code is dimension independent, dimensionality does matter for all non-trivial examples. The build process can be adjusted by modifying the `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C/GNUmakefile` file. More details on how to build AMReX can be found in *Building AMReX*.

2.2.2 Running the Code

The example code can be run as follows,

```
./main3d.gnu.DEBUG.ex
```

The result may look like,

```
AMReX (17.05-30-g5775aed933c4-dirty) initialized
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
AMReX (17.05-30-g5775aed933c4-dirty) finalized
```

The version string means the current commit 5775aed933c4 (note that the first letter g in g577.. is not part of the hash) is based on 17.05 with 30 additional commits and the AMReX work tree is dirty (i.e. there are uncommitted changes).

In the GNUmakefile there are compilation options for DEBUG mode (less optimized code with more error checking), dimensionality, compiler type, and flags to enable MPI and/or OpenMP parallelism. If there are multiple instances of a parameter, the last instance takes precedence.

2.2.3 Parallelization

Now let's build with MPI by typing `make USE_MPI=TRUE` (alternatively you can set `USE_MPI=TRUE` in the GNUmakefile). This should make an executable named `main3d.gnu.DEBUG.MPI.ex`. Note MPI in the file name. You can then run,

```
mpiexec -n 4 ./main3d.gnu.DEBUG.MPI.ex amrex.v=1
```

The result may look like,

```
MPI initialized with 4 MPI processes
AMReX (17.05-30-g5775aed933c4-dirty) initialized
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
AMReX (17.05-30-g5775aed933c4-dirty) finalized
```

If the compilation fails, you are referred to *Building AMReX* for more details on how to configure the build system. The *optional* command line argument `amrex.v=1` sets the AMReX verbosity level to 1 to print the number of MPI processes used. The default verbosity level is 1, and you can pass `amrex.v=0` to turn it off. More details on how runtime parameters are handled can be found in section *ParmParse*.

If you want to build with OpenMP, type make `USE_OMP=TRUE`. This should make an executable named `main3d.gnu.DEBUG.OMP.ex`. Note OMP in the file name. Make sure the `OMP_NUM_THREADS` environment variable is set on your system. You can then run,

```
OMP_NUM_THREADS=4 ./main3d.gnu.DEBUG.OMP.ex
```

The result may look like,

```
OMP initialized with 4 OMP threads
AMReX (17.05-30-g5775aed933c4-dirty) initialized
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
AMReX (17.05-30-g5775aed933c4-dirty) finalized
```

Note that you can build with both `USE_MPI=TRUE` and `USE_OMP=TRUE`. You can then run,

```
OMP_NUM_THREADS=4 mpiexec -n 2 ./main3d.gnu.DEBUG.MPI.OMP.ex
```

The result may look like,

```
MPI initialized with 2 MPI processes
OMP initialized with 4 OMP threads
AMReX (17.05-30-g5775aed933c4-dirty) initialized
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
AMReX (17.05-30-g5775aed933c4-dirty) finalized
```

2.3 Example: Heat Equation Solver

We now look at a more complicated example at `amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX1_C` and show how simulation results can be visualized. This example solves the heat equation,

$$\frac{\partial \phi}{\partial t} = \nabla^2 \phi$$

using forward Euler temporal integration on a periodic domain. We could use a 5-point (in 2D) or 7-point (in 3D) stencil, but for demonstration purposes we spatially discretize the PDE by first constructing (negative) fluxes on cell faces, e.g.,

$$F_{i+1/2,\,j} = \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x},$$

and then taking the divergence to update the cells,

$$\phi_{i,\,j}^{n+1} = \phi_{i,\,j}^{n} + \frac{\Delta t}{\Delta x}\left(F_{i+1/2,\,j} - F_{i-1/2,\,j}\right) + \frac{\Delta t}{\Delta y}\left(F_{i,\,j+1/2} - F_{i,\,j-1/2}\right)$$

The implementation details of the code are discussed in the Heat Equation example section of the Guided Tutorials. For now let's just build and run the code, and visualize the results.

2.3.1 Building and Running the Code

To build a 2D executable, go to `amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX1_C/Exec` and type `make DIM=2`. This will generate an executable named `main2d.gnu.ex`. To run it, type,

```
./main2d.gnu.ex inputs_2d
```

Note that the command takes a file `inputs_2d`. The calculation solves the heat equation in 2D on a domain with 256×256 cells. It runs $10,000$ steps and makes a plotfile every $1,000$ steps. When the run finishes, you will have a number of plotfiles, `plt00000, plt01000,` etc, in the directory where you are running. You can control runtime parameters such as how many time steps to run and how often to write plotfiles by setting them in `inputs_2d`.

2.4 Visualization

There are several visualization tools that can be used for AMReX plotfiles. One standard tool used within the AMReX-community is Amrvis, a package developed and supported by CCSE that is designed specifically for highly efficient visualization of block-structured hierarchical AMR data. (Amrvis can also be used to visualize performance data; see the *AMReX-based Profiling Tools* chapter for further details.) Plotfiles can also be viewed using the VisIt, ParaView, and yt packages. Particle data can be viewed using ParaView. Refer to Chapter on *Visualization* for how to use each of these tools.

2.5 Guided Tutorials

Users new to AMReX may be interested in following the Guided Tutorials. The Guided Tutorials are designed to provide an introduction to AMReX features by focusing on key concepts in a progressive way.

2.6 Example Codes

To assist users we have multiple example codes introducing AMReX functionality. They range from HelloWorld walk-thrus to stand-alone examples of complex features in practice. To access the available examples, please see AMReX Guided Tutorials and Example Codes.

CHAPTER

THREE

BUILDING AMREX

In this chapter, we discuss AMReX's build systems. Additionally, it is also possible to install AMReX using Spack (https://spack.io/). For more information see the *Spack* section.

There are three ways to use AMReX's build systems. Most AMReX developers use GNU Make. With this approach, there is no installation step; application codes adopt AMReX's build system and compile AMReX while compiling their own codes. This will be discussed in more detail in the section on *Building with GNU Make*.

The second approach is to build and install AMReX as a library using GNU make (*Building libamrex*); an application code then uses its own build system and links to AMReX as an external library.

Finally, AMReX can also be built with CMake, as detailed in the section on *Building with CMake*.

AMReX requires a C++ compiler that supports the C++17 standard, a Fortran compiler that supports the Fortran 2003 standard, and a C compiler that supports the C99 standard. Prerequisites for building with GNU Make include Python (>= 2.7, including 3) and standard tools available in any Unix-like environments (e.g., Perl and sed). For building with CMake, the minimal requirement is version 3.18.

Please note that we fully support AMReX for Linux systems in general and on the DOE supercomputers (e.g. Cori, Summit) in particular. Many of our users do build and use AMReX on Macs but we do not have the resources to fully support Mac users.

3.1 Building with GNU Make

In this build approach, you write your own make files defining a number of variables and rules. Then you invoke `make` to start the building process. This will result in an executable upon successful completion. The temporary files generated in the building process are stored in a temporary directory named `tmp_build_dir`.

3.1.1 Dissecting a Simple Make File

An example of building with GNU Make can be found in `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C`. Table 3.1 below shows a list of important variables.

Table 3.1: Important make variables

Variable	Value	Default
AMREX_HOME	Path to amrex	environment
COMP	gnu, cray, ibm, intel, llvm, or pgi	none
CXXSTD	C++ standard (c++17, c++20)	compiler default, at least c++17
DEBUG	TRUE or FALSE	FALSE
DIM	1 or 2 or 3	3
PRECISION	DOUBLE or FLOAT	DOUBLE
TEST	TRUE or FALSE	FALSE
USE_ASSERTION	TRUE or FALSE	FALSE
USE_MPI	TRUE or FALSE	FALSE
USE_OMP	TRUE or FALSE	FALSE
USE_CUDA	TRUE or FALSE	FALSE
USE_HIP	TRUE or FALSE	FALSE
USE_DPCPP	TRUE or FALSE	FALSE
USE_RPATH	TRUE or FALSE	FALSE
WARN_ALL	TRUE or FALSE	TRUE for DEBUG FALSE otherwise
AMREX_CUDA_ARCH or CUDA_ARCH	CUDA arch such as 70	70 if not set or detected
AMREX_AMD_ARCH or AMD_ARCH	AMD GPU arch such as gfx908	none if the machine is unknown
USE_GPU_RDC	TRUE or FALSE	TRUE

At the beginning of `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C/GNUmakefile`, `AMREX_HOME` is set to the path to the top directory of AMReX. Note that in the example `?=` is a conditional variable assignment operator that only has an effect if `AMREX_HOME` has not been defined (including in the environment). One can also set `AMREX_HOME` as an environment variable. For example in bash, one can set

```
export AMREX_HOME=/path/to/amrex
```

alternatively, in tcsh one can set

```
setenv AMREX_HOME /path/to/amrex
```

Note: when setting `AMREX_HOME` in the `GNUmakefile`, be aware that ~ does not expand, so `AMREX_HOME=~/amrex/` will yield an error.

One must set the `COMP` variable to choose a compiler. Currently the list of supported compilers includes gnu, cray, ibm, intel, llvm, and pgi.

One could set the `DIM` variable to either 1, 2, or 3, depending on the dimensionality of the problem. The default dimensionality is 3. AMReX uses double precision by default. One can change to single precision by setting `PRECISION=FLOAT`. (Particles have an equivalent flag `USE_SINGLE_PRECISION_PARTICLES=TRUE/FALSE`.)

Variables `DEBUG`, `TEST`, `USE_MPI` and `USE_OMP` are optional with default set to FALSE. The meaning of these variables should be obvious. When `DEBUG=TRUE`, aggressive compiler optimization flags are turned off and assertions in source code are turned on. For production runs, `DEBUG` should be set to FALSE. `TEST` and `USE_ASSERTION` are set by default in CI and add slight debugging, e.g., initializing default values in FABs. An advanced variable, `MPI_THREAD_MULTIPLE`, can be set to TRUE to initialize MPI with support for concurrent MPI calls from multiple threads.

Variables USE_CUDA, USE_HIP and USE_DPCPP are used for targeting Nvidia, AMD and Intel GPUs, respectively. At most one of the three can be TRUE. For HIP and DPC++/SYCL builds, we do only test against C++17 builds at the moment.

The variable USE_RPATH controls the link mechanism to dependent libraries. If enabled, the library path at link time will be saved as a rpath hint in created binaries. When disabled, dynamic library paths could be provided via `export LD_LIBRARY_PATH` hints at runtime.

For GCC and Clang, the variable WARN_ALL controls the compiler's warning options. There is also a make variable WARN_ERROR (with default of FALSE) to turn warnings into errors.

When USE_CUDA is TRUE, the make system will try to detect what CUDA arch should be used by running `$(CUDA_HOME)/extras/demo_suite/deviceQuery` if your computer is unknown. If it fails to detect the CUDA arch, the default value of 70 will be used. The user could override it by `make USE_CUDA=TRUE CUDA_ARCH=80` or `make USE_CUDA=TRUE AMREX_CUDA_ARCH=80`.

After defining these make variables, a number of files, `Make.defs, Make.package` and `Make.rules`, are included in the GNUmakefile. AMReX-based applications do not need to include all directories in AMReX; an application which does not use particles, for example, does not need to include files from the Particle directory in its build. In this simple example, we only need to include `$(AMREX_HOME)/Src/Base/Make.package`. An application code also has its own Make.package file (e.g., `./Make.package` in this example) to append source files to the build system using operator +=. Variables for various source files are shown below.

CEXE_sources
: C++ source files. Note that C++ source files are assumed to have a .cpp extension.

CEXE_headers
: C++ headers with .h, .hpp, or .H extension.

cEXE_sources
: C source files with .c extension.

cEXE_headers
: C headers with .h extension.

f90EXE_sources
: Free format Fortran source with .f90 extension.

F90EXE_sources
: Free format Fortran source with .F90 extension. Note that these Fortran files will go through preprocessing.

In this simple example, the extra source file, `main.cpp` is in the current directory that is already in the build system's search path. If this example has files in a subdirectory (e.g., `mysrcdir`), you will then need to add the following to `Make.package`.

```
VPATH_LOCATIONS += mysrcdir
INCLUDE_LOCATIONS += mysrcdir
```

Here VPATH_LOCATIONS and INCLUDE_LOCATIONS are the search path for source and header files, respectively.

3.1.2 Tweaking the Make System

The GNU Make build system is located at `amrex/Tools/GNUMake`. You can read `README.md` and the make files there for more information. Here we will give a brief overview.

Besides building executable, other common make commands include:

`make cleanconfig`
: This removes the executable, .o files, and the temporarily generated files for the given build. Note that one can add additional targets to this rule using the double colon (::)

`make clean` and `make realclean`
: These remove all files generated by make for all builds.

`make help`
: This shows the rules for compilation.

`make print-xxx`
: This shows the value of variable xxx. This is very useful for debugging and tweaking the make system.

Compiler flags are set in `amrex/Tools/GNUMake/comps/`. Note that variables like `CXX` and `CXXFLAGS` are reset in that directory and their values in environment variables are disregarded. However, one could override them with make command line arguments (e.g., `make CXX=/path/to/my/mpicxx`). Site-specific setups (e.g., the MPI installation) are in `amrex/Tools/GNUMake/sites/`, which includes a generic setup in `Make.unknown`. You can override the setup by having your own `sites/Make.$(host_name)` file, where variable `host_name` is your host name in the make system and can be found via `make print-host_name`. You can also have an `amrex/Tools/GNUMake/Make.local` file to override various variables. See `amrex/Tools/GNUMake/Make.local.template` for more examples of how to customize the build process.

If you need to pass macro definitions to the preprocessor, you can add them to your make file as follows,

```
DEFINES += -Dmyname1 -Dmyname2=mydefinition
```

To link to an additional library say `foo` with headers located at `foopath/include` and library at `foopath/lib`, you can add the following to your make file before the line that includes AMReX's `Make.defs`,

```
INCLUDE_LOCATIONS += foopath/include
LIBRARY_LOCATIONS += foopath/lib
LIBRARIES += -lfoo
```

3.1.3 Specifying your own compiler

The `amrex/Tools/GNUMake/Make.local` file can also specify your own compile commands by setting the variables `CXX`, `CC`, `FC`, and `F90`. This might be necessary if your systems contains non-standard names for compiler commands.

For example, the following `amrex/Tools/GNUMake/Make.local` builds AMReX using a specific compiler (in this case `gcc-8`) without MPI. Whenever `USE_MPI` is true, this configuration defaults to the appropriate `mpixxx` command:

```
ifeq ($(USE_MPI),TRUE)
  CXX = mpicxx
  CC  = mpicc
  FC  = mpif90
  F90 = mpif90
else
  CXX = g++-8
```

(continues on next page)

(continued from previous page)

```
  CC  = gcc-8
  FC  = gfortran-8
  F90 = gfortran-8
endif
```

For building with MPI, we assume `mpicxx`, `mpif90`, etc. provide access to the correct underlying compilers.

3.1.4 GCC on macOS

The example configuration above should also run on the latest macOS. On macOS the default cxx compiler is clang, whereas the default Fortran compiler is gfortran. Sometimes it is good to avoid mixing compilers, in that case we can use the `Make.local` to force using GCC. However, macOS' Xcode ships with its own (woefully outdated) version of GCC (4.2.1). It is therefore recommended to install GCC using the homebrew package manager. Running `brew install gcc` installs gcc with names reflecting the version number. If GCC 8.2 is installed, homebrew installs it as `gcc-8`. AMReX can be built using `gcc-8` (with and without MPI) by using the following `amrex/Tools/GNUMake/Make.local`:

```
CXX = g++-8
CC  = gcc-8
FC  = gfortran-8
F90 = gfortran-8

INCLUDE_LOCATIONS += /usr/local/include
```

The additional `INCLUDE_LOCATIONS` are installed using homebrew also. Note that if you are building AMReX using homebrew's gcc, it is recommended that you use homebrew's mpich. Normally it is fine to simply install its binaries: `brew install mpich`. But if you are experiencing problems, we suggest building mpich using homebrew's gcc: `brew install mpich --cc=gcc-8`.

3.1.5 Fortran

If your code does not use Fortran, you can add `BL_NO_FORT=TRUE` to your makefile to disable Fortran.

3.1.6 ccache

If you use ccache, you can add `USE_CCACHE=TRUE` to your makefile.

3.2 Building libamrex

If an application code already has its own elaborated build system and wants to use AMReX, an external AMReX library can be created instead. In this approach, one runs `./configure`, followed by `make` and `make install`. Other make options include `make distclean` and `make uninstall`. In the top AMReX directory, one can run `./configure -h` to show the various options for the configure script. In particular, one can specify the installation path for the AMReX library using:

```
./configure --prefix=[AMReX library path]
```

This approach is built on the AMReX GNU Make system. Thus the section on *Building with GNU Make* is recommended if any fine tuning is needed. The result of `./configure` is `GNUmakefile` in the AMReX top directory. One can modify the make file for fine tuning.

To compile an application code against the external AMReX library, it is necessary to set appropriate compiler flags and set the library paths for linking. To assist with this, when the AMReX library is built, a configuration file is created in `[AMReX library path]/lib/pkgconfig/amrex.pc`. This file contains the Fortran and C++ flags used to compile the AMReX library as well as the appropriate library and include entries.

The following sample GNU Makefile will compile a `main.cpp` source file against an external AMReX library, using the C++ flags and library paths used to build AMReX:

```
AMREX_LIBRARY_HOME ?= [AMReX library path]

LIBDIR := $(AMREX_LIBRARY_HOME)/lib
INCDIR := $(AMREX_LIBRARY_HOME)/include

COMPILE_CPP_FLAGS ?= $(shell awk '/Cflags:/ {$$1=$$2=""; print $$0}' $(LIBDIR)/pkgconfig/
↪amrex.pc)
COMPILE_LIB_FLAGS ?= $(shell awk '/Libs:/ {$$1=$$2=""; print $$0}' $(LIBDIR)/pkgconfig/
↪amrex.pc)

CFLAGS := -I$(INCDIR) $(COMPILE_CPP_FLAGS)
LFLAGS := -L$(LIBDIR) $(COMPILE_LIB_FLAGS)

all:
        g++ -o main.exe main.cpp $(CFLAGS) $(LFLAGS)
```

3.3 Building with CMake

An alternative to the approach described in the section on *Building libamrex* is to install AMReX as an external library by using the CMake build system. A CMake build is a two-step process. First `cmake` is invoked to create configuration files and makefiles in a chosen directory (`builddir`). This is roughly equivalent to running `./configure` (see the section on *Building libamrex*). Next, the actual build and installation are performed by invoking `make install` from within `builddir`. This installs the library files in a chosen installation directory (`installdir`). If no installation path is provided by the user, AMReX will be installed in `/path/to/amrex/installdir`. The CMake build process is summarized as follows:

```
mkdir /path/to/builddir
cd    /path/to/builddir
cmake [options] -DCMAKE_BUILD_TYPE=[Debug|Release|RelWithDebInfo|MinSizeRel] -DCMAKE_
↪INSTALL_PREFIX=/path/to/installdir  /path/to/amrex
make  install
make  test_install  # optional step to test if the installation is working
```

In the above snippet, `[options]` indicates one or more options for the customization of the build, as described in the subsection on *Customization options*. If the option `CMAKE_BUILD_TYPE` is omitted, `CMAKE_BUILD_TYPE=Release` is assumed. Although the AMReX source could be used as build directory, we advise against doing so. After the installation is complete, `builddir` can be removed.

3.3.1 Customization options

AMReX build can be customized by setting the value of suitable configuration variables on the command line via the `-D <var>=<value>` syntax, where `<var>` is the variable to set and `<value>` its desired value. For example, one can enable OpenMP support as follows:

```
cmake -DAMReX_OMP=YES -DCMAKE_INSTALL_PREFIX=/path/to/installdir  /path/to/amrex
```

In the example above `<var>=AMReX_OMP` and `<value>=YES`. Configuration variables requiring a boolean value are evaluated to true if they are assigned a value of `1`, `ON`, `YES`, `TRUE`, `Y`. Conversely they are evaluated to false if they are assigned a value of `0`, `OFF`, `NO`, `FALSE`, `N`. Boolean configuration variables are case-insensitive. The list of available options is reported in the *table* below.

Table 3.2: AMReX build options (refer to section Building GPU Support for GPU-related options).

Variable Name	Description	Default	Po
CMAKE_Fortran_COMPILER	User-defined Fortran compiler		use
CMAKE_CXX_COMPILER	User-defined C++ compiler		use
CMAKE_Fortran_FLAGS	User-defined Fortran flags		use
CMAKE_CXX_FLAGS	User-defined C++ flags		use
CMAKE_CXX_STANDARD	C++ standard	compiler/17	17,
AMReX_SPACEDIM	Dimension of AMReX build	3	1, 2
USE_XSDK_DEFAULTS	Use xSDK defaults settings	NO	YE
AMReX_BUILD_SHARED_LIBS	Build as shared C++ library	NO (unless xSDK)	YE
AMReX_FORTRAN	Enable Fortran language	NO	YE
AMReX_PRECISION	Set the precision of reals	DOUBLE	DO
AMReX_PIC	Build Position Independent Code	NO	YE
AMReX_IPO	Interprocedural optimization (IPO/LTO)	NO	YE
AMReX_MPI	Build with MPI support	YES	YE
AMReX_OMP	Build with OpenMP support	NO	YE
AMReX_GPU_BACKEND	Build with on-node, accelerated GPU backend	NONE	NO
AMReX_GPU_RDC	Build with Relocatable Device Code support	YES	YE
AMReX_FORTRAN_INTERFACES	Build Fortran API	NO	YE
AMReX_LINEAR_SOLVERS	Build AMReX linear solvers	YES	YE
AMReX_AMRDATA	Build data services	NO	YE
AMReX_AMRLEVEL	Build AmrLevel class	YES	YE
AMReX_EB	Build Embedded Boundary support	NO	YE
AMReX_PARTICLES	Build particle classes	NO	YE
AMReX_PARTICLES_PRECISION	Set reals precision in particle classes	Same as AMReX_PRECISION	DO
AMReX_BASE_PROFILE	Build with basic profiling support	NO	YE
AMReX_TINY_PROFILE	Build with tiny profiling support	NO	YE
AMReX_TRACE_PROFILE	Build with trace-profiling support	NO	YE
AMReX_COMM_PROFILE	Build with comm-profiling support	NO	YE
AMReX_MEM_PROFILE	Build with memory-profiling support	NO	YE
AMReX_TP_PROFILE	Third-party profiling options	IGNORE	CR
AMReX_TESTING	Build for testing –sets MultiFab initial data to NaN	NO	YE
AMReX_MPI_THREAD_MULTIPLE	Concurrent MPI calls from multiple threads	NO	YE
AMReX_PROFPARSER	Build with profile parser support	NO	YE
AMReX_ROCTX	Build with roctx markup profiling support	NO	YE
AMReX_FPE	Build with Floating Point Exceptions checks	NO	YE
AMReX_ASSERTIONS	Build with assertions turned on	NO	YE

Table 3.2 – continued from previous page

Variable Name	Description	Default	Po
AMReX_BOUND_CHECK	Enable bound checking in Array4 class	NO	YE
AMReX_EXPORT_DYNAMIC	Enable backtrace on macOS	NO (unless Darwin)	YE
AMReX_SENSEI	Enable the SENSEI in situ infrastucture	NO	YE
AMReX_NO_SENSEI_AMR_INST	Disables the instrumentation in amrex::Amr	NO	YE
AMReX_CONDUIT	Enable Conduit support	NO	YE
AMReX_ASCENT	Enable Ascent support	NO	YE
AMReX_HYPRE	Enable HYPRE interfaces	NO	YE
AMReX_PETSC	Enable PETSc interfaces	NO	YE
AMReX_SUNDIALS	Enable SUNDIALS interfaces	NO	YE
AMReX_HDF5	Enable HDF5-based I/O	NO	YE
AMReX_HDF5_ZFP	Enable compression with ZFP in HDF5-based I/O	NO	YE
AMReX_PLOTFILE_TOOLS	Build and install plotfile postprocessing tools	NO	YE
AMReX_ENABLE_TESTS	Enable CTest suite	NO	YE
AMReX_DIFFERENT_COMPILER	Allow an app to use a different compiler	NO	YE
AMReX_INSTALL	Generate Install Targets	YES	YE
AMReX_PROBINIT	Enable support for probin file	Platform dependent	YE

The option `CMAKE_BUILD_TYPE=Debug` implies `AMReX_ASSERTIONS=YES`. In order to turn off assertions in debug mode, `AMReX_ASSERTIONS=NO` must be set explicitly while invoking CMake.

The `CMAKE_C_COMPILER`, `CMAKE_CXX_COMPILER`, and `CMAKE_Fortran_COMPILER` options are used to tell CMake which compiler to use for the compilation of C, C++, and Fortran sources respectively. If those options are not set by the user, CMake will use the system default compilers.

The options `CMAKE_Fortran_FLAGS` and `CMAKE_CXX_FLAGS` allow the user to set their own compilation flags for Fortran and C++ source files respectively. If `CMAKE_Fortran_FLAGS`/ `CMAKE_CXX_FLAGS` are not set by the user, they will be initialized with the value of the environmental variables `FFLAGS`/ `CXXFLAGS`. If neither `FFLAGS`/ `CXXFLAGS` nor `CMAKE_Fortran_FLAGS`/ `CMAKE_CXX_FLAGS` are defined, AMReX default flags are used.

For a detailed explanation of GPU support in AMReX CMake, refer to section *Building GPU Support*.

3.3.2 CMake and macOS

While not strictly necessary when using homebrew on macOS, it is highly recommended that the user specifies `-DCMAKE_C_COMPILER=$(which gcc-X) -DCMAKE_CXX_COMPILER=$(which g++-X)` (where X is the GCC version installed by homebrew) when using gfortran. This is because homebrew's CMake defaults to the Clang C/C++ compiler. Normally Clang plays well with gfortran, but if there are some issues, we recommend telling CMake to use gcc for C/C++ also.

3.3.3 Importing AMReX into your CMake project

In order to import AMReX into your CMake project, you need to include the following line in the appropriate CMakeLists.txt file:

```
find_package(AMReX)
```

Calls to `find_package(AMReX)` will find a valid installation of AMReX, if present, and import its settings and targets into your CMake project. Imported AMReX targets can be linked to any of your targets, after they have been made available following a successful call to `find_package(AMReX)`, by including the following line in the appropriate CMakeLists.txt file:

```
target_link_libraries( <your-target-name> PUBLIC AMReX::<amrex-target-name> )
```

In the above snippet, <amrex-target-name> is any of the targets listed in the table below.

Table 3.3: AMReX targets available for import.

Target name	Description
amrex	AMReX library
Flags_CXX	C++ flags preset (interface)
Flags_Fortran	Fortran flags preset (interface)
Flags_FPE	Floating Point Exception flags (interface)

The options used to configure the AMReX build may result in certain parts, or components, of the AMReX source code to be excluded from compilation. For example, setting -DAMReX_LINEAR_SOLVERS=no at configure time prevents the compilation of AMReX linear solvers code. Your CMake project can check which component is included in the AMReX library via *find_package*:

```
find_package(AMReX REQUIRED <components-list>)
```

The keyword REQUIRED in the snippet above will cause a fatal error if AMReX is not found, or if it is found but the components listed in <components-list> are not include in the installation. A list of AMReX component names and related configure options are shown in the table below.

Table 3.4: AMReX components.

Option	Component
AMReX_SPACEDIM	1D, 2D, 3D
AMReX_PRECISION	DOUBLE, SINGLE
AMReX_FORTRAN	FORTRAN
AMReX_PIC	PIC
AMReX_MPI	MPI
AMReX_OMP	OMP
AMReX_GPU_BACKEND	CUDA, HIP, SYCL
AMReX_FORTRAN_INTERFACES	FINTERFACES
AMReX_LINEAR_SOLVERS	LSOLVERS
AMReX_AMRDATA	AMRDATA
AMReX_AMRLEVEL	AMRLEVEL
AMReX_EB	EB
AMReX_PARTICLES	PARTICLES
AMReX_PARTICLES_PRECISION	PDOUBLE, PSINGLE
AMReX_BASE_PROFILE	BASEP
AMReX_TINY_PROFILE	TINYP
AMReX_TRACE_PROFILE	TRACEP
AMReX_COMM_PROFILE	COMMP
AMReX_MEM_PROFILE	MEMP
AMReX_PROFPARSER	PROFPARSER
AMReX_FPE	FPE
AMReX_ASSERTIONS	ASSERTIONS
AMReX_SENSEI	SENSEI
AMReX_CONDUIT	CONDUIT
AMReX_ASCENT	ASCENT
AMReX_HYPRE	HYPRE
AMReX_PLOTFILE_TOOLS	PFTOOLS

As an example, consider the following CMake code:

```
find_package(AMReX REQUIRED 3D EB)
target_link_libraries( Foo PUBLIC AMReX::amrex )
```

The code in the snippet above checks whether an AMReX installation with 3D and Embedded Boundary support is available on the system. If so, AMReX is linked to target `Foo` and AMReX flags preset is used to compile `Foo`'s C++ sources. If no AMReX installation is found or if the available one was built without 3D or Embedded Boundary support, a fatal error is issued.

You can tell CMake to look for the AMReX library in non-standard paths by setting the environmental variable `AMReX_ROOT` to point to the AMReX installation directory or by adding `-DAMReX_ROOT=<path/to/amrex/installation/directory>` to the `cmake` invocation. More details on `find_package` can be found here.

3.4 AMReX on Windows

The AMReX team does development on Linux machines, from laptops to supercomputers. Many people also use AMReX on Macs without issues.

We do not officially support AMReX on Windows, and many of us do not have access to any Windows machines. However, we believe there are no fundamental issues for it to work on Windows.

(1) AMReX mostly uses standard C++17. We run continous integration tests on Windows with MSVC and Clang compilers.

(2) We use POSIX signal handling when floating point exceptions, segmentation faults, etc. happen. This capability is not supported on Windows.

(3) Memory profiling is an optional feature in AMReX that is not enabled by default. It reads memory system information from the OS to give us a summary of our memory usage. This is not supported on Windows.

3.5 Spack

AMReX can be installed using the scientific software package manager Spack. Spack supports multiple versions and configurations of AMReX across a wide variety of platforms and environments. To learn more about Spack visit http://www.spack.io. For system requirements and installation instructions please see https://spack.readthedocs.io/.

Once Spack has been downloaded and the Spack environment enabled, AMReX can be installed with the command,

```
spack install amrex
```

This will install the latest release of AMReX and required dependencies if needed.

AMReX can be built in several combinations of versions and configurations. Available options can be viewed by typing,

```
spack info amrex
```

For example, suppose we want to install the development version of AMReX for a two dimensional simulation with Cuda support for Cuda Architecture `sm_60`. Then we would use the install commands,

```
spack install amrex@develop dimensions=2 +cuda cuda_arch=60
```

CHAPTER

FOUR

BASICS

In this chapter, we present the basics of AMReX. The implementation source codes are in `amrex/Src/Base/`. Note that AMReX classes and functions are in namespace `amrex`. For clarity, we usually drop `amrex::` in the example codes here. It is also assumed that headers have been properly included. We recommend you study the tutorial in `amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX1_C` while reading this chapter. After reading this chapter, one should be able to develop single-level parallel codes using AMReX. It should also be noted that this is not a comprehensive reference manual.

4.1 Dimensionality

As we have mentioned in *Building AMReX*, the dimensionality of AMReX must be set at compile time. A macro, `AMREX_SPACEDIM`, is defined to be the number of spatial dimensions. C++ codes can also use the `amrex::SpaceDim` variable. Fortran codes can use either the macro and preprocessing or do

```
use amrex_fort_module, only : amrex_spacedim
```

The coordinate directions are zero based.

4.2 Vector, Array, GpuArray, Array1D, Array2D, and Array3D

`Vector` class in `AMReX_Vector.H` is derived from `std::vector`. The main difference between `Vector` and `std::vector` is that `Vector::operator[]` provides bound checking when compiled with `DEBUG=TRUE`.

`Array` class in `AMReX_Array.H` is simply an alias to `std::array`. AMReX also provides `GpuArray`, a trivial type that works on both host and device. (It was added when the minimal requirement for C++ standard was C++11, for which `std::array` does not work on device.) It also works when compiled just for CPU. Besides `GpuArray`, AMReX also provides GPU safe `Array1D`, `Array2D` and `Array3D` that are 1, 2 and 3-dimensional fixed size arrays, respectively. These three class templates can have non-zero based indexing.

4.3 Real

AMReX can be compiled to use either double precision (which is the default) or single precision. `amrex::Real` is typedef'd to either **`double`** or **`float`**. C codes can use `amrex_real`. They are defined in `AMReX_REAL.H`. The data type is accessible in Fortran codes via

```
use amrex_fort_module, only : amrex_real
```

In C++, AMReX also provides a user literal `_rt` so that one can have a proper type for constants (e.g., `2.7_rt`).

4.4 Long

AMReX defines a 64 bit integer type `amrex::Long` that is an alias to **`long`** on Unix-like systems and **`long long`** on Windows. In C, the type alias is `amrex_long`. In Fortran, one can use `amrex_long` defined in `amrex_fort_module`.

4.5 ParallelDescriptor

AMReX users do not need to use MPI directly. Parallel communication is often handled by the data abstraction classes (e.g.,MultiFab; section on *FabArray, MultiFab and iMultiFab*). In addition, AMReX has provided namespace `ParallelDescriptor` in `AMReX_ParallelDescriptor.H`. The frequently used functions are

```
int myproc = ParallelDescriptor::MyProc();  // Return the rank

int nprocs = ParallelDescriptor::NProcs();  // Return the number of processes

if (ParallelDescriptor::IOProcessor()) {
    // Only the I/O process executes this
}

int ioproc = ParallelDescriptor::IOProcessorNumber();  // I/O rank

ParallelDescriptor::Barrier();

// Broadcast 100 ints from the I/O Processor
Vector<int> a(100);
ParallelDescriptor::Bcast(a.data(), a.size(),
                    ParallelDescriptor::IOProcessorNumber())

// See AMReX_ParallelDescriptor.H for many other Reduce functions
ParallelDescriptor::ReduceRealSum(x);
```

Additionally, `amrex_paralleldescriptor_module` in `Src/Base/AMReX_ParallelDescriptor_F.F90` provides a number of functions for Fortran.

4.6 ParallelContext

Users can also use groups of MPI subcommunicators to perform simultaneous physics calculations. These comms are managed by AMReX's `ParallelContext` in `AMReX_ParallelContext.H`. It maintains a stack of `MPI_Comm` handlers. A global comm is placed in the `ParallelContext` stack during AMReX's initialization and additional subcommunicators can be handled by adding comms with `push(MPI_Comm)` and removed using `pop()`. This creates a hierarchy of `MPI_Comm` objects that can be used to split work as the user sees fit. Note that `ParallelDescriptor` by default uses AMReX's base comm, independent of the status of the `ParallelContext` stack.

`ParallelContext` also tracks and returns information about the local (most recently added) and global `MPI_Comm`. The most common access functions are given below. See `AMReX_ParallelContext.H` for a full listing of the available functions.

```
MPI_Comm subCommA = ....;
MPI_Comm subCommB = ....;
// Add a communicator to ParallelContext.
// After these pushes, subCommB becomes the
//     "local" communicator.
ParallelContext::push(subCommA);
ParallelContext::push(subCommB);

// Get Global and Local communicator (subCommB).
MPI_Comm globalComm = ParallelContext::CommunicatorAll();
MPI_Comm localComm  = ParallelContext::CommunicatorSub();

// Get local number of ranks and global IO Processor Number.
int localRanks = ParallelContext::NProcsSub();
int globalIO     = ParallelContext::IOProcessorNumberAll();

if (ParallelContext::IOProcessorSub()) {
    // Only the local I/O process executes this
}

// Translation of global rank to local communicator rank.
// Returns MPI_UNDEFINED if comms do not overlap.
int localRank = ParallelContext::global_to_local_rank(globalrank);

// Translations of MPI rank IDs using integer arrays.
// Returns MPI_UNDEFINED if comms do not overlap.
ParallelContext::global_to_local_rank(local_array, global_array, n);
ParallelContext::local_to_global_rank(global_array, local_array, n);

// Remove the last added subcommunicator.
// This would make "subCommA" the new local communicator.
// Note: The user still needs to free "subCommB".
ParallelContext::pop();
```

4.7 Print

AMReX provides classes in AMReX_Print.H for printing messages to standard output or any C++ ostream. The main reason one should use them instead of std::cout is that messages from multiple processes or threads do not get mixed up. Below are some examples.

```
Print() <<  "x = " << x << "\n"; // Print on I/O processor

Real pi = std::atan(1.0)*4.0;
// Print on rank 3 with precision of 17 digits
// SetPrecision does not modify cout's floating-point decimal precision setting.
Print(3).SetPrecision(17) << pi << "\n";

int oldprec = std::cout.precision(10);
Print() << pi << "\n";  // Print with 10 digits

AllPrint() << "Every process prints\n";  // Print on every process

std::ofstream ofs("my.txt", std::ofstream::out);
Print(ofs) << "Print to a file" << std::endl;
ofs.close();

AllPrintToFile("file.") << "Each process appends to its own file (e.g., file.3)\n";
```

It should be emphasized that Print() without any argument only prints on the I/O process. A common mistake in using it for debug printing is one forgets that for non-I/O processes to print we should use AllPrint() or Print(rank).

4.8 ParmParse

ParmParse in AMReX_ParmParse.H is a class providing a database for the storage and retrieval of command-line and input-file arguments. When amrex::Initialize(int& argc, char**& argv) is called, the first command-line argument after the executable name (if there is one, and it does not contain the character '=' or start with '-') is taken to be the inputs file, and the contents of the file are used to initialize the ParmParse database. The rest of the command-line arguments are also parsed by ParmParse, with the exception of those following a '--' which signals command line sharing (see section *Sharing the Command Line*).

4.8.1 Inputs File

The format of the inputs file is a series of definitions in the form of prefix.name = value value .... For each line, text after # are comments. Here is an example inputs file.

```
nsteps    = 100               # integer
nsteps    = 1000              # nsteps appears a second time
dt        = 0.03              # floating point number
ncells    = 128 64 32         # a list of 3 ints
xrange    = -0.5 0.5          # a list of 2 reals
title     = "Three Kingdoms"  # a string
hydro.cfl = 0.8               # with prefix, hydro
```

The following code shows how to use ParmParse to get/query the values.

```
ParmParse pp;

int nsteps = 0;
pp.query("nsteps", nsteps);
amrex::Print() << nsteps << "\n";  // 1000

Real dt;
pp.get("dt", dt);  // runtime error if dt is not in inputs

Vector<int> numcells;
// The variable name 'numcells' can be different from parameter name 'ncells'.
pp.getarr("ncells", numcells);
amrex::Print() << numcells.size() << "\n";  // 3

Vector<Real> xr {-1.0, 1.0};
if (!queryarr("xrange", xr)) {
    amrex::Print() << "Cannot find xrange in inputs, "
                   << "so the default {-1.0,1.0} will be used\n";
}

std::string title;
pp.query("title", title);  // query string

ParmParse pph("hydro");  // with prefix 'hydro'
Real cfl;
pph.get("cfl", cfl);    // get parameter with prefix
```

Note that when there are multiple definitions for a parameter `ParmParse` by default returns the last one. The difference between `query` and `get` should also be noted. It is a runtime error if `get` fails to get the value, whereas `query` returns an error code without generating a runtime error that will abort the run.

4.8.2 Overriding Parameters with Command-Line Arguments

It is sometimes convenient to override parameters with command-line arguments without modifying the inputs file. The command-line arguments after the inputs file are added later than the file to the database and are therefore used by default. For example, to change the value of `ncells` and `hydro.cfl`, one can run with:

```
myexecutable myinputsfile ncells="64 32 16" hydro.cfl=0.9
```

4.8.3 Setting Parameter Values Inside Functions

An application code may want to set values or defaults that differ from the those in AMReX in a function. This is accomplished in two steps:

- First, define a function that sets the variable(s).
- Second, pass the name of that function to `amrex::Initialize`.

The example function below sets variable values using two different approaches to highlight subtle differences in implementation:

```
void add_par () {
   ParmParse pp("eb2");

   // `variable_one` can be overridden by an inputs file and/or command line argument.
   if(not pp.contains("variable_one")) {
      pp.add("variable_one",false);
   }

   // The inputs file or command line arguments for `variable_two` are ignored.
   pp.add("variable_two",false);
};
```

First this function, `add_par`, declares a `ParmParse` object that will be used to set variables. In the next section of code, we check if the value for `variable_one` has already been set elsewhere before writing to it. This approach prevents the function from overriding a value set in the inputs file or at the command line. In the next section, we write a value to `variable_two` without a conditional statement. In this case, we will ignore values for `variable_two` set in the inputs file or as a command line argument —effectively overriding them with the value set here in the function.

In the second step, we pass the name of the function we defined to `amrex::Initialize`. In the example above the function was called `add_par`, and therefore we write,

```
amrex::Initialize(argc, argv, true, MPI_COMM_WORLD, add_par);
```

Now AMReX will use the user defined function to appropriately set the desired values.

4.8.4 Sharing the Command Line

In some cases we want AMReX to only read some of the command line arguments – this happens, for example, when we are going to use AMReX in cooperation with another code package and that code also takes arguments.

Consider:

```
main2d.gnu.exe inputs amrex.v=1 amrex.fpe_trap_invalid=1 -- -tao_monitor
```

In this example, AMReX will parse the inputs file and the optional AMReX command line arguments, but will ignore arguments after the double dashes.

4.8.5 Command Line Flags

AMReX allows application codes to parse flags such as `-h` or `--help` while still making use of ParmParse for parsing other runtime parameters but only if it is the first argument after the executable. If the first argument following the executable name begins with a dash, AMReX will initialize without reading any parameters and the application code may then parse the command line and handle those cases. Several built in functions are available to help do this. They are briefly introduced in the table below.

Table 4.1: AMReX functions for parsing the command line.

Function	Type	Purpose
`amrex::get_command()`	String	Get the entire command line.
`amrex::get_argument_count()`	Int	Get the number of command line arguments after the executable.
`amrex:get_command_argument(int n)`	String	Returns the n-th argument after the exectuable.

4.9 Parser

AMReX provides a parser in `AMReX_Parser.H` that can be used at runtime to evaluate mathematical expressions given in the form of string. It supports +, -, *, /, ** (power), ^ (power), `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `abs`, `floor`, `ceil` and `fmod`. The minimum and maximum of two numbers can be computed with `min` and `max`, respectively. It supports the Heaviside step function, `heaviside(x1,x2)` that gives `0`, `x2`, `1`, for `x1 < 0`, `x1 = 0` and `x1 > 0`, respectively. It also supports the Bessel function of the first kind of order `n` `jn(n,x)`. There is `if(a,b,c)` that gives `b` or `c` depending on the value of `a`. A number of comparison operators are supported, including <, >, ==, !=, <=, and >=. The Boolean results from comparison can be combined by `and` and `or`, and they hold the value `1` for true and `0` for false. The precedence of the operators follows the convention of the C and C++ programming languages. Here is an example of using the parser.

```
Parser parser("if(x>a and x<b, sin(x)*cos(y)*if(z<0, 1.0, exp(-z)), .3*c**2)");
parser.setConstant(a, ...);
parser.setConstant(b, ...);
parser.setConstant(c, ...);
parser.registerVariables({"x","y","z"});
auto f = parser.compile<3>();  // 3 because there are three variables.

// f can be used in both host and device code.  It takes 3 arguments in
// this example.  The parser object must be alive for f to be valid.
for (int k = 0; ...) {
  for (int j = 0; ...) {
    for (int i = 0; ...) {
      a(i,j,k) = f(i*dx, j*dy, k*dz);
    }
  }
}
```

Local automatic variables can be defined in the expression. For example,

```
Parser parser("r2=x*x+y*y; r=sqrt(r2); cos(a+r2)*log(r)"
parser.setConstant(a, ...);
parser.registerVariables({"x","y"});
auto f = parser.compile<2>();  // 2 because there are two variables.
```

Note that an assignment to an automatic variable must be terminated with `;`, and one should avoid name conflict between the local variables and the constants set by `setConstant` and the variables registered by `registerVariables`.

Besides `amrex::Parser` for floating point numbers, AMReX also provides `amrex::IParser` for integers. The two parsers have a lot of similarity, but floating point number specific functions (e.g., `sqrt`, `sin`, etc.) are not supported in `IParser`. In addition to `/` whose result truncates towards zero, the integer parser also supports `//` whose result truncates towards negative infinity.

4.10 Initialize and Finalize

As we have mentioned, `Initialize` must be called to initialize the execution environment for AMReX and `Finalize` must be paired with `Initialize` to release the resources used by AMReX. There are two versions of `Initialize`.

```
void Initialize (MPI_Comm mpi_comm,
                 std::ostream& a_osout = std::cout,
                 std::ostream& a_oserr = std::cerr,
                 ErrorHandler a_errhandler = nullptr);

void Initialize (int& argc, char**& argv, bool build_parm_parse=true,
                 MPI_Comm mpi_comm = MPI_COMM_WORLD,
                 const std::function<void()>& func_parm_parse = {},
                 std::ostream& a_osout = std::cout,
                 std::ostream& a_oserr = std::cerr,
                 ErrorHandler a_errhandler = nullptr);
```

`Initialize` tests if MPI has been initialized. If MPI has been initialized, AMReX will duplicate the `MPI_Comm` argument. If not, AMReX will initialize MPI and ignore the `MPI_Comm` argument.

Both versions have two optional `std::ostream` parameters, one for standard output in `Print` (section *Print*) and the other for standard error, and they can be accessed with functions `OutStream()` and `ErrorStream()`. Both versions can also take an optional error handler function. If it is provided by the user, AMReX will use it to handle errors and signals. Otherwise, AMReX will use its own function for error and signal handling.

The first version of `Initialize` does not parse the command line options, whereas the second version will build ParmParse database (section *ParmParse*) unless `build_parm_parse` parameter is `false`. In the second version, one can pass a function that adds ParmParse parameters to the database instead of reading from command line or input file.

Because many AMReX classes and functions (including destructors inserted by the compiler) do not function properly after **amrex**`:Finalize` is called, it's best to put the codes between `amrex::Initialize` and `amrex::Finalize` into its scope (e.g., a pair of curly braces or a separate function) to make sure resources are properly freed.

4.11 Example of AMR Grids

In block-structured AMR, there is a hierarchy of logically rectangular grids. The computational domain on each AMR level is decomposed into a union of rectangular domains. Fig. 4.1 below shows an example of AMR with three total levels. In the AMReX numbering convention, the coarsest level is level 0. The coarsest grid (*black*) covers the domain with 16^2 cells. Bold lines represent grid boundaries. There are two intermediate resolution grids (*blue*) at level 1 and the cells are a factor of two finer than those at level 0. The two finest grids (*red*) are at level 2 and the cells are a factor of two finer than the level 1 cells. There are 1, 2 and 2 Boxes on levels 0, 1, and 2, respectively. Note that there is no direct parent-child connection. In this chapter, we will focus on single levels.

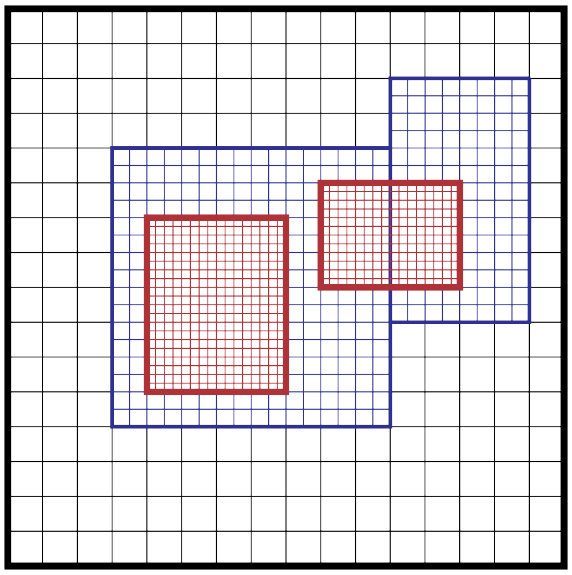


Fig. 4.1: Example of AMR grids. There are three levels in total. There are 1, 2 and 2 Boxes on levels 0, 1, and 2, respectively.

4.12 Box, IntVect and IndexType

`Box` in AMReX_Box.H is the data structure for representing a rectangular domain in indexing space. In Fig. 4.1, there are 1, 2 and 2 Boxes on levels 0, 1 and 2, respectively. `Box` is a dimension-dependent class. It has lower and upper corners (represented by `IntVect`) and an index type (represented by `IndexType`). A `Box` contains no floating-point data.

4.12.1 IntVect

`IntVec` is a dimension-dependent class representing an integer vector in `AMREX_SPACEDIM`-dimensional space. An `IntVect` can be constructed as follows,

```
IntVect iv(AMREX_D_DECL(19, 0, 5));
```

Here `AMREX_D_DECL` is a macro that expands `AMREX_D_DECL(19,0,5)` to either `19` or `19, 0` or `19, 0, 5` depending on the number of dimensions. The data can be accessed via `operator[]`, and the internal data pointer can be returned by function `getVect`. For example

```
for (int idim = 0; idim < AMREX_SPACEDIM; ++idim) {
    amrex::Print() << "iv[" << idim << "] = " << iv[idim] << "\n";
}
const int * p = iv.getVect();  // This can be passed to Fortran/C as an array
```

The class has a static function `TheZeroVector()` returning the zero vector, `TheUnitVector()` returning the unit vector, and `TheDimensionVector (int dir)` returning a reference to a constant `IntVect` that is zero except in the `dir`-direction. Note the direction is zero-based. `IntVect` has a number of relational operators, ==, !=, <, <=, > , and >= that can be used for lexicographical comparison (e.g., key of `std::map`), and a class `IntVect::shift_hasher` that can be used as a hash function (e.g., for `std::unordered_map`). It also has various arithmetic operators. For example,

```
IntVect iv(AMREX_D_DECL(19, 0, 5));
IntVect iv2(AMREX_D_DECL(4, 8, 0));
iv += iv2;  // iv is now (23,8,5)
iv *= 2;    // iv is now (46,16,10);
```

In AMR codes, one often needs to do refinement and coarsening on `IntVect`. The refinement operation can be done with the multiplication operation. However, the coarsening requires care because of the rounding towards zero behavior of integer division in Fortran, C and C++. For example `int i = -1/2` gives `i = 0`, and what we want is usually `i = -1`. Thus, one should use the coarsen functions:

```
IntVect iv(AMREX_D_DECL(127,127,127));
IntVect coarsening_ratio(AMREX_D_DECL(2,2,2));
iv.coarsen(2);                  // Coarsen each component by 2
iv.coarsen(coarsening_ratio);  // Component-wise coarsening
const auto& iv2 = amrex::coarsen(iv, 2); // Return an IntVect w/o modifying iv
IntVect iv3 = amrex::coarsen(iv, coarsening_ratio); // iv not modified
```

Finally, we note that `operator<<` is overloaded for `IntVect` and therefore one can call

```
amrex::Print() << iv << "\n";
std::cout << iv << "\n";
```

4.12.2 IndexType

This class defines an index as being cell based or node based in each dimension. The default constructor defines a cell based type in all directions. One can also construct an `IndexType` with an `IntVect` with zero and one representing cell and node, respectively.

```
// Node in x-direction and cell based in y and z-directions
// (i.e., x-face of numerical cells)
IndexType xface(IntVect{AMREX_D_DECL(1,0,0)});
```

The class provides various functions including

```
// True if the IndexType is cell based in all directions.
bool cellCentered () const;

// True if the IndexType is cell based in dir-direction.
bool cellCentered (int dir) const;

// True if the IndexType is node based in all directions.
bool nodeCentered () const;

// True if the IndexType is node based in dir-direction.
bool nodeCentered (int dir) const;
```

Index type is a very important concept in AMReX. It is a way of representing the notion of indices i and $i + 1/2$.

4.12.3 Box

A Box is an abstraction for defining discrete regions of AMREX_SPACEDIM-dimensional indexing space. Boxes have an IndexType and two IntVects representing the lower and upper corners. Boxes can exist in positive and negative indexing space. Typical ways of defining a Box are

```
IntVect lo(AMREX_D_DECL(64,64,64));
IntVect hi(AMREX_D_DECL(127,127,127));
IndexType typ({AMREX_D_DECL(1,1,1)});
Box cc(lo,hi);        // By default, Box is cell based.
Box nd(lo,hi+1,typ);  // Construct a nodal Box.
Print() << "A cell-centered Box " << cc << "\n";
Print() << "An all nodal Box    " << nd << "\n";
```

Depending the dimensionality, the output of the code above is

```
A cell-centered Box ((64,64,64) (127,127,127) (0,0,0))
An all nodal Box    ((64,64,64) (128,128,128) (1,1,1))
```

For simplicity, we will assume it is 3D for the rest of this section. In the output, three integer tuples for each box are the lower corner indices, upper corner indices, and the index types. Note that 0 and 1 denote cell and node, respectively. For each tuple like (64,64,64), the 3 numbers are for 3 directions. The two Boxes in the code above represent different indexing views of the same domain of 64^3 cells. Note that in AMReX convention, the lower side of a cell has the same integer value as the cell centered index. That is if we consider a cell based index represent i, the nodal index with the same integer value represents $i - 1/2$. Fig. 4.2 shows some of the different index types for 2D.

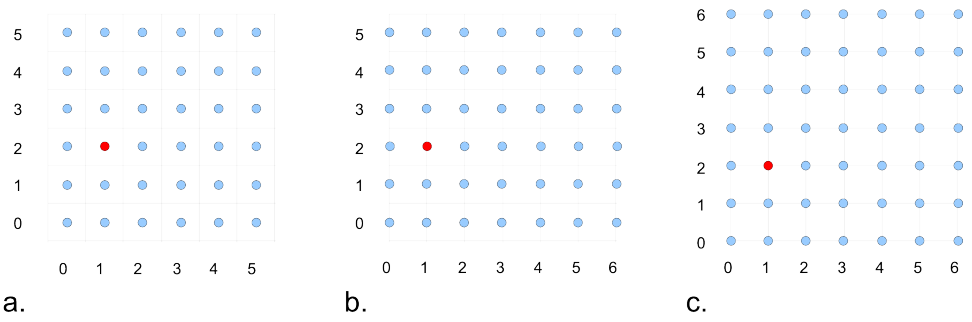


Fig. 4.2: Some of the different index types in two dimensions: (a) cell-centered, (b) x-face-centered (i.e., nodal in x-direction only), and (c) corner/nodal, i.e., nodal in all dimensions.

There are a number of ways of converting a Box from one type to another.

```
Box b0 ({64,64,64}, {127,127,127}); // Index type: (cell, cell, cell)

Box b1 = surroundingNodes(b0);  // A new Box with type (node, node, node)
Print() << b1;                  // ((64,64,64) (128,128,128) (1,1,1))
Print() << b0;                  // Still ((64,64,64) (127,127,127) (0,0,0))

Box b2 = enclosedCells(b1);     // A new Box with type (cell, cell, cell)
if (b2 == b0) {                 // Yes, they are identical.
   Print() << "b0 and b2 are identical!\n";
}
```

(continues on next page)

(continued from previous page)

```
Box b3 = convert(b0, {0,1,0});  // A new Box with type (cell, node, cell)
Print() << b3;                  // ((64,64,64) (127,128,127) (0,1,0))

b3.convert({0,0,1});            // Convert b0 to type (cell, cell, node)
Print() << b3;                  // ((64,64,64) (127,127,128) (0,0,1))

b3.surroundingNodes();          //  Exercise for you
b3.enclosedCells();             //  Exercise for you
```

The internal data of `Box` can be accessed via various member functions. Examples are

```
const IntVect& smallEnd () const;  // Get the small end of the Box
int bigEnd (int dir) const;        // Get the big end in dir direction
const int* loVect () const&;       // Get a const pointer to the lower end
const int* hiVect () const&;       // Get a const pointer to the upper end
```

Boxes can be refined and coarsened. Refinement or coarsening does not change the index type. Some examples are shown below.

```
Box ccbx ({16,16,16}, {31,31,31});
ccbx.refine(2);
Print() << ccbx;                  // ((32,32,32) (63,63,63) (0,0,0))
Print() << ccbx.coarsen(2);       // ((16,16,16) (31,31,31) (0,0,0))

Box ndbx ({16,16,16}, {32,32,32}, {1,1,1});
ndbx.refine(2);
Print() << ndbx;                  // ((32,32,32) (64,64,64) (1,1,1))
Print() << ndbx.coarsen(2);       // ((16,16,16) (32,32,32) (1,1,1))

Box facebx ({16,16,16}, {32,31,31}, {1,0,0});
facebx.refine(2);
Print() << facebx;                // ((32,32,32) (64,63,63) (1,0,0))
Print() << facebx.coarsen(2);     // ((16,16,16) (32,31,31) (1,0,0))

Box uncoarsenable ({16,16,16}, {30,30,30});
Print() << uncoarsenable.coarsen(2); // ((8,8,8), (15,15,15));
Print() << uncoarsenable.refine(2);  // ((16,16,16), (31,31,31));
                                     // Different from the original!
```

Note that the behavior of refinement and coarsening depends on the index type. A refined `Box` covers the same physical domain as the original `Box`, and a coarsened `Box` also covers the same physical domain if the original `Box` is coarsenable. `Box uncoarsenable` in the example above is considered uncoarsenable because its coarsened version does not cover the same physical domain in the AMR context.

Boxes can grow in one or all directions. There are a number of grow functions. Some are member functions of the `Box` class and others are free functions in the `amrex` namespace.

The `Box` class provides the following member functions testing if a `Box` or `IntVect` is contained within this `Box`. Note that it is a runtime error if the two Boxes have different types.

```
bool contains (const Box& b) const;
bool strictly_contains (const Box& b) const;
bool contains (const IntVect& p) const;
bool strictly_contains (const IntVect& p) const;
```

Another very common operation is the intersection of two Boxes like in the following examples.

```
Box b0 ({16,16,16}, {31,31,31});
Box b1 ({ 0, 0,30}, {23,23,63});
if (b0.intersects(b1)) {                    // true
    Print() << "b0 and b1 intersect.\n";
}

Box b2 = b0 & b1;      // b0 and b1 unchanged
Print() << b2;         // ((16,16,30) (23,23,31) (0,0,0))

Box b3 = surroundingNodes(b0) & surroundingNodes(b1); // b0 and b1 unchanged
Print() << b3;         // ((16,16,30) (24,24,32) (1,1,1))

b0 &= b2;              // b2 unchanged
Print() << b0;         // ((16,16,30) (23,23,31) (0,0,0))

b0 &= b3;              // Runtime error because of type mismatch!
```

4.13 Dim3 and XDim3

`Dim3` and `XDim3` are plain structs with three fields,

```
struct Dim3 { int x; int y; int z; };
struct XDim3 { Real x; Real y; Real z; };
```

One can convert an `IntVect` to `Dim3`,

```
IntVect iv(...);
Dim3 d3 = iv.dim3();
```

`Dim3` always has three fields even when AMReX is built for 1D or 2D. For the example above, the extra fields are set to zero. Given a `Box`, one can get its lower and upper bounds and use them to write dimension agnostic loops.

```
Box bx(...);
Dim3 lo = lbound(bx);
Dim3 hi = ubound(bx);
for     (int k = lo.z; k <= hi.z; ++k) {
  for   (int j = lo.y; j <= hi.y; ++j) {
    for (int i = lo.x; i <= hi.x; ++i) {
    }
  }
}
```

One can also call function `Dim3 length(Box const&)` to return the length of a Box.

4.14 RealBox and Geometry

A RealBox stores the physical location in floating-point numbers of the lower and upper corners of a rectangular domain.

The Geometry class in AMReX_Geometry.H describes problem domain and coordinate system for rectangular problem domains. A Geometry object can be constructed with

```
explicit Geometry (const Box&     dom,
                   const RealBox* rb     = nullptr,
                   int            coord  = -1,
                   int*           is_per = nullptr) noexcept;

Geometry (const Box& dom, const RealBox& rb, int coord,
          Array<int,AMREX_SPACEDIM> const& is_per) noexcept;
```

Here the constructors take a cell-centered Box specifying the indexing space domain, a RealBox specifying the physical domain, an int specifying coordinate system type, and an int pointer or array specifying periodicity. If a RealBox is not given in the first constructor, AMReX will construct one based on ParmParse parameters, geometry.prob_lo / geometry.prob_hi / geometry.prob_extent, where each of the parameter is an array of AMREX_SPACEDIM real numbers. See the section on *Problem Definition* for more details about how to specify these.

The argument for coordinate system is an integer type with valid values being 0 (Cartesian), or 1 (cylindrical), or 2 (spherical). If it is invalid as in the case of the default argument value of the first constructor, AMReX will query the ParmParse database for geometry.coord_sys and use it if one is found. If it cannot find the parameter, the coordinate system is set to 0 (i.e., Cartesian coordinates).

The Geometry class has the concept of periodicity. An argument can be passed specifying periodicity in each dimension. If it is not given in the first constructor, the domain is assumed to be non-periodic unless there is the ParmParse integer array parameter geometry.is_periodic with 0 denoting non-periodic and 1 denoting periodic. Below is an example of defining a Geometry for a periodic rectangular domain of $[-1.0, 1.0]$ in each direction discretized with 64 numerical cells in each direction.

```
int n_cell = 64;

// This defines a Box with n_cell cells in each direction.
Box domain(IntVect{AMREX_D_DECL(       0,        0,        0)},
           IntVect{AMREX_D_DECL(n_cell-1, n_cell-1, n_cell-1)});

// This defines the physical box, [-1,1] in each direction.
RealBox real_box({AMREX_D_DECL(-1.0,-1.0,-1.0)},
                 {AMREX_D_DECL( 1.0, 1.0, 1.0)});

// This says we are using Cartesian coordinates
int coord = 0;

// This sets the boundary conditions to be doubly or triply periodic
Array<int,AMREX_SPACEDIM> is_periodic {AMREX_D_DECL(1,1,1)};

// This defines a Geometry object
Geometry geom(domain, real_box, coord, is_periodic);
```

A Geometry object can return various information of the physical domain and the indexing space domain. For example,

```
const auto problo = geom.ProbLoArray(); // Lower corner of the physical
                                        // domain.  The return type is
                                        // GpuArray<Real,AMREX_SPACEDIM>.
Real yhi = geom.ProbHi(1);              // y-direction upper corner
const auto dx = geom.CellSizeArray();   // Cell size for each direction.
const Box& domain = geom.Domain();      // Index domain
bool is_per = geom.isPeriodic(0);       // Is periodic in x-direction?
if (geom.isAllPeriodic()) {}            // Periodic in all direction?
if (geom.isAnyPeriodic()) {}            // Periodic in any direction?
```

4.15 BoxArray

BoxArray is a class in AMReX_BoxArray.H for storing a collection of Boxes on a single AMR level. One can make a BoxArray out of a single Box and then chop it into multiple Boxes.

```
Box domain(IntVect{0,0,0}, IntVect{127,127,127});
BoxArray ba(domain);  // Make a new BoxArray out of a single Box
Print() << "BoxArray size is " << ba.size() << "\n";  // 1
ba.maxSize(64);       // Chop into boxes of 64^3 cells
Print() << ba;
```

The output is like below,

```
(BoxArray maxbox(8)
       m_ref->m_hash_sig(0)
((0,0,0) (63,63,63) (0,0,0)) ((64,0,0) (127,63,63) (0,0,0))
((0,64,0) (63,127,63) (0,0,0)) ((64,64,0) (127,127,63) (0,0,0))
((0,0,64) (63,63,127) (0,0,0)) ((64,0,64) (127,63,127) (0,0,0))
((0,64,64) (63,127,127) (0,0,0)) ((64,64,64) (127,127,127) (0,0,0)) )
```

It shows that ba now has 8 Boxes, and it also prints out each Box.

In AMReX, BoxArray is a global data structure. It holds all the Boxes in a collection, even though a single process in a parallel run only owns some of the Boxes via domain decomposition. In the example above, a 4-process run may divide the work and each process owns say 2 Boxes (see section on *DistributionMapping*). Each process can then allocate memory for the floating point data on the Boxes it owns (see sections on *FabArray, MultiFab and iMultiFab* & *BaseFab, FArrayBox, IArrayBox, and Array4*).

BoxArray has an indexing type, just like Box. Each Box in a BoxArray has the same type as the BoxArray itself. In the following example, we show how one can convert BoxArray to a different type.

```
BoxArray cellba(Box(IntVect{0,0,0}, IntVect{63,127,127}));
cellba.maxSize(64);
BoxArray faceba = cellba;       // Make a copy
faceba.convert(IntVect{0,0,1}); // convert to index type (cell, cell, node)
// Return an all node BoxArray
const BoxArray& nodeba = amrex::convert(faceba, IntVect{1,1,1});
Print() << cellba[0] << "\n";  // ((0,0,0) (63,63,63) (0,0,0))
Print() << faceba[0] << "\n";  // ((0,0,0) (63,63,64) (0,0,1))
Print() << nodeba[0] << "\n";  // ((0,0,0) (64,64,64) (1,1,1))
```

As shown in the example above, BoxArray has an **operator[]** that returns a Box given an index. It should be emphasized that there is a difference between its behavior and the usual behavior of an subscript operator one might expect.

The subscript operator in BoxArray returns by **value instead of reference**. This means code like below is meaningless because it modifies a temporary return value.

```
ba[3].coarsen(2);  // DO NOT DO THIS!  Doesn't do what one might expect.
```

BoxArray has a number of member functions that allow the Boxes to be modified. For example,

```
BoxArray& refine (int refinement_ratio);   // Refine each Box in BoxArray
BoxArray& refine (const IntVect& refinement_ratio);
BoxArray& coarsen (int refinement_ratio);  // Coarsen each Box in BoxArray
BoxArray& coarsen (const IntVect& refinement_ratio);
```

We have mentioned at the beginning of this section that BoxArray is a global data structure storing Boxes shared by all processes. The operation of a deep copy is thus undesirable because it is expensive and the extra copy wastes memory. The implementation of the BoxArray class uses std::shared_ptr to an internal container holding the actual Box data. Thus making a copy of BoxArray is a quite cheap operation. The conversion of types and coarsening are also cheap because they can share the internal data with the original BoxArray. In our implementation, function refine does create a new deep copy of the original data. Also note that a BoxArray and its variant with a different type share the same internal data is an implementation detail. We discuss this so that the users are aware of the performance and resource cost. Conceptually we can think of them as completely independent of each other.

```
BoxArray ba(...);  // original BoxArray
BoxArray ba2 = ba; // a copy that shares the internal data with the original
ba2.coarsen(2);    // Modify the copy
// The original copy is unmodified even though they share internal data.
```

For advanced users, AMReX provides functions performing the intersection of a BoxArray and a Box. These functions are much faster than a naive implementation of performing intersection of the Box with each Box in the BoxArray. If one needs to perform those intersections, functions amrex::intersect, BoxArray::intersects and BoxArray::intersections should be used.

4.16 DistributionMapping

DistributionMapping is a class in AMReX_DistributionMapping.H that describes which process owns the data living on the domains specified by the Boxes in a BoxArray. Like BoxArray, there is an element for each Box in DistributionMapping, including the ones owned by other parallel processes. One can construct a DistributionMapping object given a BoxArray,

```
DistributionMapping dm {ba};
```

or by simply making a copy,

```
DistributionMapping dm {another_dm};
```

Note that this class is built using std::shared_ptr. Thus making a copy is relatively cheap in terms of performance and memory resources. This class has a subscript operator that returns the process ID at a given index.

By default, DistributionMapping uses an algorithm based on space filling curve to determine the distribution. One can change the default via the ParmParse parameter DistributionMapping.strategy. KNAPSACK is a common choice that is optimized for load balance. One can also explicitly construct a distribution. The DistributionMapping class allows the user to have complete control by passing an array of integers that represent the mapping of grids to processes.

```
DistributionMapping dm;   // empty object
Vector<int> pmap {...};
// The user fills the pmap array with the values specifying owner processes
dm.define(pmap);  // Build DistributionMapping given an array of process IDs.
```

4.17 BaseFab, FArrayBox, IArrayBox, and Array4

AMReX is a block-structured AMR framework. Although AMR introduces irregularity to the data and algorithms, there is regularity at the block/Box level because each is still logically rectangular, and the data structure at the Box level is conceptually simple. BaseFab is a class template for multi-dimensional array-like data structure on a Box. The template parameter is typically basic types such as Real, int or char. The dimensionality of the array is AMREX_SPACEDIM *plus one*. The additional dimension is for the number of components. The data are internally stored in a contiguous block of memory in Fortran array order (i.e., column-major order) for $(x, y, z, \text{component})$, and each component also occupies a contiguous block of memory because of the ordering. For example, a BaseFab<Real> with 4 components defined on a three-dimensional Box(IntVect{-4,8,32},IntVect{32,64,48}) is like a Fortran array of real(amrex_real), dimension(-4:32,8:64,32:48,0:3). Note that the convention in C++ part of AMReX is the component index is zero based. The code for constructing such an object is as follows,

```
Box bx(IntVect{-4,8,32}, IntVect{32,64,48});
int numcomps = 4;
BaseFab<Real> fab(bx,numcomps);
```

Most applications do not use BaseFab directly, but utilize specialized classes derived from BaseFab. The most common types are FArrayBox in AMReX_FArrayBox.H derived from BaseFab<Real> and IArrayBox in AMReX_IArrayBox.H derived from BaseFab<int>.

These derived classes also obtain many BaseFab member functions via inheritance. We now show some common usages of these functions. To get the Box where a BaseFab or its derived object is defined, one can call

```
const Box& box() const;
```

To the number of component, one can call

```
int nComp() const;
```

To get a pointer to the array data, one can call

```
T* dataPtr(int n=0);      // Data pointer to the nth component
                          // T is template parameter (e.g., Real)
const T* dataPtr(int n=0) const; // const version
```

The typical usage of the returned pointer is then to pass it to a Fortran or C function that works on the array data (see the section on *Fortran and C++ Kernels*). BaseFab has several functions that set the array data to a constant value. Two examples are as follows.

```
void setVal(T x);         // Set all data to x
// Set the sub-region specified by bx to value x starting from component
// nstart.  ncomp is the total number of component to be set.
void setVal(T x, const Box& bx, int nstart, int ncomp);
```

One can copy data from one BaseFab to another.

```
BaseFab<T>& copy (const BaseFab<T>& src, const Box& srcbox, int srccomp,
                  const Box& destbox, int destcomp, int numcomp);
```

Here the function copies the data from the region specified by `srcbox` in the source `BaseFab src` into the region specified by `destbox` in the destination BaseFab that invokes the function call. Note that although `srcbox` and `destbox` may be different, they must be the same size, shape and index type, otherwise a runtime error occurs. The user also specifies how many components (`int numcomp`) are copied starting at component srccomp in src and stored starting at component destcomp. BaseFab has functions returning the minimum or maximum value.

```
T min (int comp=0) const;  // Minimum value of given component.
T min (const Box& subbox, int comp=0) const; // Minimum value of given
                                             // component in given subbox.
T max (int comp=0) const;  // Maximum value of given component.
T max (const Box& subbox, int comp=0) const; // Maximum value of given
                                             // component in given subbox.
```

`BaseFab` also has many arithmetic functions. Here are some examples using FArrayBox.

```
Box box(IntVect{0,0,0}, IntVect{63,63,63});
int ncomp = 2;
FArrayBox fab1(box, ncomp);
FArrayBox fab2(box, ncomp);
fab1.setVal(1.0);    // Fill fab1 with 1.0
fab1.mult(10.0, 0);  // Multiply component 0 by 10.0
fab2.setVal(2.0);    // Fill fab2 with 2.0
Real a = 3.0;
fab2.saxpy(a, fab1); // For both components, fab2 <- a * fab1 + fab2
```

These floating point operation functions are templated with parameter `RunOn` specifying where they run, `RunOn::Host` or `RunOn::Device`. When AMReX is built just for CPU, the template parameter has a default value of `RunOn::Host` so that the user does not need to specify it for backward compatibility, and if `RunOn::Device` is provided it will be ignored. However, when AMReX is built with GPU support, one must specify where to run for these `BaseFab` functions. For example,

```
fab1.setVal<RunOn::Host>(1.0);    // Fill fab1 with 1.0
fab1.mult<RunOn::Device>(10.0, 0);  // Multiply component 0 by 10.0
```

For more complicated expressions that are not supported, one can write Fortran or C/C++ functions for those (see the section on *Fortran and C++ Kernels*). In C++, one can use `Array4`, which is a class template for accessing `BaseFab` data in a more array like manner using `operator()`. Below is an example of using `Array4`.

```
FArrayBox afab(...), bfab(...);
IArrayBox ifab(...);
Array4<Real> const& a = afab.array();
Array4<Real const> const b = bfab.const_array();
Array4<int const> m = ifab.array();
Dim3 lo = lbound(a);
Dim3 hi = ubound(a);
int nc = a.nComp();
for (int n = 0; n < nc; ++n) {
  for     (int k = lo.z; k <= hi.z; ++k) {
    for   (int j = lo.y; j <= hi.y; ++j) {
      for (int i = lo.x; i <= hi.x; ++i) {
```

(continues on next page)

(continued from previous page)

```
        if (m(i,j,k) > 0) {
          a(i,j,k,n) *= 2.0;
        } else {
          a(i,j,k,n) = 2.0*a(i,j,k,n) + 0.5*(b(i-1,j,k,n)+b(i+1,j,k,n));
        }
      }
    }
  }
}
```

Note that `operator()` of `Array4` takes either three or four arguments. The optional fourth argument has a default value of zero. The two `const`s in `Array4<Real const> const&` have different meaning. The first `const` inside `<>` means the data accessed via `Array4` is read-only, whereas the second `const` means the `Array4` object itself cannot be modified to point to other data. In the example above, neither `m(i,j,k) = 0` nor `b(i,j,k) = 0.0` is allowed. However one is allowed to do `m = ifab2.array()` to assign `m` again, but not to `b`. The behavior is in some sense similar to `double const * const p`.

`BaseFab` and its derived classes are containers for data on `Box`. Recall that `Box` has various types (see the section on *Box, IntVect and IndexType*). The examples in this section so far use the default cell based type. However, some functions will result in a runtime error if the types mismatch. For example.

```
Box ccbx ({16,16,16}, {31,31,31});           // cell centered box
Box ndbx ({16,16,16}, {31,31,31}, {1,1,1});  // nodal box
FArrayBox ccfab(ccbx);
FArrayBox ndfab(ndbx);
ccfab.setVal(0.0);
ndfab.copy(ccfab);   // runtime error due to type mismatch
```

Because it typically contains a lot of data, BaseFab's copy constructor and copy assignment operator are disabled to prevent performance degradation. However, BaseFab does provide a move constructor. In addition, it also provides a constructor for making an alias of an existing object. Here is an example using `FArrayBox`.

```
FArrayBox orig_fab(box, 4);  // 4-component FArrayBox
// Make a 2-component FArrayBox that is an alias of orig_fab
// starting from component 1.
FArrayBox alias_fab(orig_fab, amrex::make_alias, 1, 2);
```

In this example, the alias `FArrayBox` has only two components even though the original one has four components. The alias has a sliced component view of the original `FArrayBox`. This is possible because of the array ordering. However, it is not possible to slice in the real space (i.e., the first `AMREX_SPACEDIM` dimensions). Note that no new memory is allocated in constructing the alias and the alias contains a non-owning pointer. It should be emphasized that the alias will contain a dangling pointer after the original `FArrayBox` reaches its end of life. One can also construct an alias `BaseFab` given an `Array4`,

```
Array4<Real> const a = orig_fab.array();
FArrayBox alias_fab(a);
```

4.18 FabArray, MultiFab and iMultiFab

FabArray<FAB> is a class template in AMReX_FabArray.H for a collection of FABs on the same AMR level associated with a BoxArray (see the section on *BoxArray*). The template parameter FAB is usually BaseFab<T> or its derived classes (e.g., FArrayBox). However, FabArray can also be used to hold other data structures. To construct a FabArray, a BoxArray must be provided because the FabArray is intended to hold *grid* data defined on a union of rectangular regions embedded in a uniform index space. For example, a FabArray object can be used to hold data for one level as in Fig. 4.1.

FabArray is a parallel data structure in which the data (i.e., FAB) are distributed among parallel processes. For each process, a FabArray contains only the FAB objects owned by that process, and the process operates only on its local data. For operations that require data owned by other processes, remote communications are involved. Thus, the construction of a FabArray requires a DistributionMapping (see the section on *DistributionMapping*) that specifies which process owns which Box. For level 2 (*red*) in Fig. 4.1, there are two Boxes. Suppose there are two parallel processes, and we use a DistributionMapping that assigns one Box to each process. Then the FabArray on each process is built on the BoxArray with both Boxes, but contains only the FAB associated with its process.

In AMReX, there are some specialized classes derived from FabArray. The iMultiFab class in AMReX_iMultiFab.H is derived from FabArray<IArrayBox>. The most commonly used FabArray kind class is MultiFab in AMReX_MultiFab.H derived from FabArray<FArrayBox>. In the rest of this section, we use MultiFab as example. However, these concepts are equally applicable to other types of FabArrays. There are many ways to define a MultiFab. For example,

```
// ba is BoxArray
// dm is DistributionMapping
int ncomp = 4;
int ngrow = 1;
MultiFab mf(ba, dm, ncomp, ngrow);
```

Here we define a MultiFab with 4 components and 1 ghost cell. A MultiFab contains a number of FArrayBoxes (see the section on *BaseFab, FArrayBox, IArrayBox, and Array4*) defined on Boxes grown by the number of ghost cells (1 in this example). That is the Box in the FArrayBox is not exactly the same as in the BoxArray. If the BoxArray has a Box{(7,7,7) (15,15,15)}, the one used for constructing FArrayBox will be Box{(6,6,6) (16,16,16)} in this example. For cells in FArrayBox, we call those in the original Box **valid cells** and the grown part **ghost cells**. Note that FArrayBox itself does not have the concept of ghost cells. Ghost cells are a key concept of MultiFab, however, that allows for local operations on ghost cell data originated from remote processes. We will discuss how to fill ghost cells with data from valid cells later in this section. MultiFab also has a default constructor. One can define an empty MultiFab first and then call the define function as follows.

```
MultiFab mf;
// ba is BoxArray
// dm is DistributionMapping
int ncomp = 4;
int ngrow = 1;
mf.define(ba, dm, ncomp, ngrow);
```

Given an existing MultiFab, one can also make an alias MultiFab as follows.

```
// orig_mf is an existing MultiFab
int start_comp = 3;
int num_comps = 1;
MultiFab alias_mf(orig_mf, amrex::make_alias, start_comp, num_comps);
```

Here the first integer parameter is the starting component in the original MultiFab that will become component 0 in the alias MultiFab and the second integer parameter is the number of components in the alias. It's a runtime error if

the sum of the two integer parameters is greater than the number of the components in the original MultiFab. Note that the alias MultiFab has exactly the same number of ghost cells as the original MultiFab.

We often need to build new MultiFabs that have the same `BoxArray` and `DistributionMapping` as a given MultiFab. Below is an example of how to achieve this.

```
// mf0 is an already defined MultiFab
const BoxArray& ba = mf0.boxArray();
const DistributionMapping& dm = mf0.DistributionMap();
int ncomp = mf0.nComp();
int ngrow = mf0.nGrow();
MultiFab mf1(ba,dm,ncomp,ngrow);  // new MF with the same ncomp and ngrow
MultiFab mf2(ba,dm,ncomp,0);      // new MF with no ghost cells
// new MF with 1 component and 2 ghost cells
MultiFab mf3(mf0.boxArray(), mf0.DistributionMap(), 1, 2);
```

As we have repeatedly mentioned in this chapter that `Box` and `BoxArray` have various index types. Thus, `MultiFab` also has an index type that is obtained from the `BoxArray` used for defining the `MultiFab`. It should be noted again that index type is a very important concept in AMReX. Let's consider an example of a finite-volume code, in which the state is defined as cell averaged variables and the fluxes are defined as face averaged variables.

```
// ba is cell-centered BoxArray
// dm is DistributionMapping
int ncomp = 3;  // Suppose the system has 3 components
int ngrow = 0;  // no ghost cells
MultiFab state(ba, dm, ncomp, ngrow);
MultiFab xflux(amrex::convert(ba, IntVect{1,0,0}), dm, ncomp, 0);
MultiFab yflux(amrex::convert(ba, IntVect{0,1,0}), dm, ncomp, 0);
MultiFab zflux(amrex::convert(ba, IntVect{0,0,1}), dm, ncomp, 0);
```

Here all `MultiFab`s use the same `DistributionMapping`, but their `BoxArray`s have different index types. The state is cell-based, whereas the fluxes are on the faces. Suppose the cell based `BoxArray` contains a `Box{(8,8,16), (15,15,31)}`. The state on that `Box` is conceptually a Fortran Array with the dimension of `(8:15,8:15,16:31,0:2)`. The fluxes are arrays with slightly different indices. For example, the x-direction flux for that `Box` has the dimension of `(8:16,8:15,16:31,0:2)`. Note there is an extra element in x-direction.

The `MultiFab` class provides many functions performing common arithmetic operations on a `MultiFab` or between `MultiFab`s built with the *same* `BoxArray` and `DistributionMap`. For example,

```
Real dmin = mf.min(3);    // Minimum value in component 3 of MultiFab mf
                          // no ghost cells included
Real dmax = mf.max(3,1); // Maximum value in component 3 of MultiFab mf
                          // including 1 ghost cell
mf.setVal(0.0);           // Set all values to zero including ghost cells

MultiFab::Add(mfdst, mfsrc, sc, dc, nc, ng);  // Add mfsrc to mfdst
MultiFab::Copy(mfdst, mfsrc, sc, dc, nc, ng); // Copy from mfsrc to mfdst
// MultiFab mfdst: destination
// MultiFab mfsrc: source
// int      sc   : starting component index in mfsrc for this operation
// int      dc   : starting component index in mfdst for this operation
// int      nc   : number of components for this operation
// int      ng   : number of ghost cells involved in this operation
//                 mfdst and mfsrc may have more ghost cells
```

We refer the reader to `amrex/Src/Base/AMReX_MultiFab.H` and `amrex/Src/Base/AMReX_FabArray.H` for more

details. It should be noted again it is a runtime error if the two MultiFabs passed to functions like MultiFab::Copy are not built with the *same* BoxArray (including index type) and DistributionMapping.

It is usually the case that the Boxes in the BoxArray used for building a MultiFab are non-intersecting except that they can be overlapping due to nodal index type. However, MultiFab can have ghost cells, and in that case FArrayBoxes are defined on Boxes larger than the Boxes in the BoxArray. Parallel communication is then needed to fill the ghost cells with valid cell data from other FArrayBoxes possibly on other parallel processes. The function for performing this type of communication is FillBoundary.

```
MultiFab mf(...parameters omitted...);
Geometry geom(...parameters omitted...);
mf.FillBoundary();                      // Fill ghost cells for all components
                                        // Periodic boundaries are not filled.
mf.FillBoundary(geom.periodicity());  // Fill ghost cells for all components
                                        // Periodic boundaries are filled.
mf.FillBoundary(2, 3);         // Fill 3 components starting from component 2
mf.FillBoundary(geom.periodicity(), 2, 3);
```

Note that FillBoundary does not modify any valid cells. Also note that MultiFab itself does not have the concept of periodic boundary, but Geometry has, and we can provide that information so that periodic boundaries can be filled as well. You might have noticed that a ghost cell could overlap with multiple valid cells from different FArrayBoxes in the case of nodal index type. In that case, it is unspecified that which valid cell's value is used to fill the ghost cell. It ought to be the case the values in those overlapping valid cells are the same up to roundoff errors. If a ghost cell does not overlap with any valid cells, its value will not be modified by FillBoundary.

Another type of parallel communication is copying data from one MultiFab to another MultiFab with a different BoxArray or the same BoxArray with a different DistributionMapping. The data copy is performed on the regions of intersection. The most generic interface for this is

```
mfdst.ParallelCopy(mfsrc, compsrc, compdst, ncomp, ngsrc, ngdst, period, op);
```

Here mfdst and mfsrc are destination and source MultiFabs, respectively. Parameters compsrc, compdst, and ncomp are integers specifying the range of components. The copy is performed on ncomp components starting from component compsrc of mfsrc and component compdst of mfdst. Parameters ngsrc and ngdst specify the number of ghost cells involved for the source and destination, respectively. Parameter period is optional, and by default no periodic copy is performed. Like FillBoundary, one can use Geometry::periodicity() to provide the periodicity information. The last parameter is also optional and is set to FabArrayBase::COPY by default. One could also use FabArrayBase::ADD. This determines whether the function copies or adds data from the source to the destination. Similar to FillBoundary, if a destination cell has multiple cells as source, it is unspecified that which source cell is used in FabArrayBase::COPY, and, for FabArrayBase::ADD, the multiple values are all added to the destination cell. This function has two variants, in which the periodicity and operation type are also optional.

```
mfdst.ParallelCopy(mfsrc, period, op);  // mfdst and mfsrc must have the same
                                        // number of components
mfdst.ParallelCopy(mfsrc, compsrc, compdst, ncomp, period, op);
```

Here the number of ghost cells involved is zero, and the copy is performed on all components if unspecified (assuming the two MultiFabs have the same number of components).

Both ParallelCopy(...) and FillBoundary(...) are blocking calls. They will only return when the communication is completed and the destination MultiFab is guaranteed to be properly updated. AMReX also provides non-blocking versions of these calls to allow users to overlap communication with calculation and potentially improve overall application performance.

The non-blocking calls are used by calling the ***_nowait(...) function to begin the comm operation, followed by the ***_finish() function at a later time to complete it. For example:

```
mfA.ParallelCopy_nowait(mfsrc, period, op);

// ... Any overlapping calc work here on other data, e.g.
mfB.setVal(0.0);

mfA.ParallelCopy_finish();

mfB.FillBoundary_nowait(period);
// ... Overlapping work here
mfB.FillBoundary_finish();
```

All function signatures of the blocking calls are also available in the non-blocking calls and should be used in the *nowait* function. The *finish* functions take no parameters, as the required data is stored during *nowait* and retrieved. Users that choose to use non-blocking calls must ensure the calls are properly used to avoid race conditions, which typically means not interacting with the MultiFab between the `_nowait` and `_finish` calls.

4.19 MFIter and Tiling

In this section, we will first show how `MFIter` works without tiling. Then we will introduce the concept of logical tiling. Finally we will show how logical tiling can be launched via `MFIter`.

4.19.1 MFIter without Tiling

In the section on *FabArray, MultiFab and iMultiFab*, we have shown some of the arithmetic functionalities of `MultiFab`, such as adding two MultiFabs together. In this section, we will show how you can operate on the `MultiFab` data with your own functions. AMReX provides an iterator, `MFIter` for looping over the FArrayBoxes in MultiFabs. For example,

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi) // Loop over grids
{
    // This is the valid Box of the current FArrayBox.
    // By "valid", we mean the original ungrown Box in BoxArray.
    const Box& box = mfi.validbox();

    // A reference to the current FArrayBox in this loop iteration.
    FArrayBox& fab = mf[mfi];

    // Obtain Array4 from FArrayBox.  We can also do
    //     Array4<Real> const& a = mf.array(mfi);
    Array4<Real> const& a = fab.array();

    // Call function f1 to work on the region specified by box.
    // Note that the whole region of the Fab includes ghost
    // cells (if there are any), and is thus larger than or
    // equal to "box".
    f1(box, a);
}
```

Here function `f1` might be something like below,

```
void f1 (Box const& bx, Array4<Real> const& a)
{
   const auto lo = lbound(bx);
   const auto hi = ubound(bx);
   for     (int k = lo.z; k <= hi.z; ++k) {
     for   (int j = lo.y; j <= hi.y; ++j) {
       for (int i = lo.x; i <= hi.x; ++i) {
         a(i,j,k) = ...
       }
     }
   }
}
```

`MFIter` only loops over grids owned by this process. For example, suppose there are 5 Boxes in total and processes 0 and 1 own 2 and 3 Boxes, respectively. That is the MultiFab on process 0 has 2 FArrayBoxes, whereas there are 3 FArrayBoxes on process 1. Thus the numbers of iterations of MFIter are 2 and 3 on processes 0 and 1, respectively.

In the example above, `MultiFab` is assumed to have a single component. If it has multiple components, we can call `int nc = mf.nComp()` or `int nc = a.nComp()` to get the number of components.

There is only one `MultiFab` in the example above. Below is an example of working with multiple MultiFabs. Note that these two MultiFabs are not necessarily built on the same `BoxArray`. But they must have the same `DistributionMapping`, and their BoxArrays are typically related (e.g., they are different due to index types).

```
// U and F are MultiFabs
for (MFIter mfi(F); mfi.isValid(); ++mfi) // Loop over grids
{
    const Box& box = mfi.validbox();

    Array4<Real const> const& u = U.const_array(mfi);
    Array4<Real      > const& f = F.array(mfi);

    f2(box, u, f);
}
```

Here function `f2` might be something like below,

```
void f1 (Box const& bx, Array4<Real const> const& u,
         Array4<Real> const& f)
{
   const auto lo = lbound(bx);
   const auto hi = ubound(bx);
   const int nf = f.nComp();
   for (int n = 0; n < nf; ++n) {
     for     (int k = lo.z; k <= hi.z; ++k) {
       for   (int j = lo.y; j <= hi.y; ++j) {
         for (int i = lo.x; i <= hi.x; ++i) {
           f(i,j,k,n) = ... u(i,j,k,n) ...
         }
       }
     }
   }
}
```

4.19.2 MFIter with Tiling

Tiling, also known as cache blocking, is a well known loop transformation technique for improving data locality. This is often done by transforming the loops into tiling loops that iterate over tiles and element loops that iterate over the data elements within a tile. For example, the original loops might look like this in Fortran

```
do k = kmin, kmax
  do j = jmin, jmax
    do i = imin, imax
      A(i,j,k) = B(i+1,j,k)+B(i-1,j,k)+B(i,j+1,k)+B(i,j-1,k) &
                 +B(i,j,k+1)+B(i,j,k-1)-6.0d0*B(i,j,k)
    end do
  end do
end do
```

And the manually tiled loops might look like

```
jblocksize = 11
kblocksize = 16
jblocks = (jmax-jmin+jblocksize-1)/jblocksize
kblocks = (kmax-kmin+kblocksize-1)/kblocksize
do kb = 0, kblocks-1
  do jb = 0, jblocks-1
    do k = kb*kblocksize, min((kb+1)*kblocksize-1,kmax)
      do j = jb*jblocksize, min((jb+1)*jblocksize-1,jmax)
        do i = imin, imax
          A(i,j,k) = B(i+1,j,k)+B(i-1,j,k)+B(i,j+1,k)+B(i,j-1,k) &
                     +B(i,j,k+1)+B(i,j,k-1)-6.0d0*B(i,j,k)
        end do
      end do
    end do
  end do
end do
```

As we can see, to manually tile individual loops is very labor-intensive and error-prone for large applications. AMReX has incorporated the tiling construct into `MFIter` so that the application codes can get the benefit of tiling easily. An `MFIter` loop with tiling is almost the same as the non-tiling version. The first example in (see the previous section on *MFIter without Tiling*) requires only two minor changes:

1. passing `true` when defining `MFIter` to indicate tiling;
2. calling `tilebox` instead of `validbox` to obtain the work region for the loop iteration.

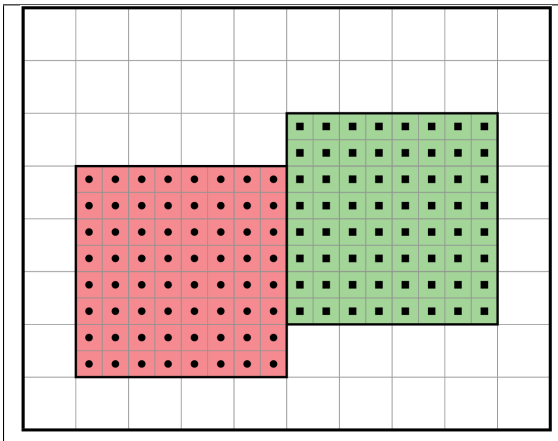
```
//               * true *  turns on tiling
for (MFIter mfi(mf,true); mfi.isValid(); ++mfi) // Loop over tiles
{
    //                   tilebox() instead of validbox()
    const Box& box = mfi.tilebox();

    FArrayBox& fab = mf[mfi];
    Array4<Real> const& a = fab.array();
    f1(box, a);
}
```

The second example in the previous section on *MFIter without Tiling* also requires only two minor changes.

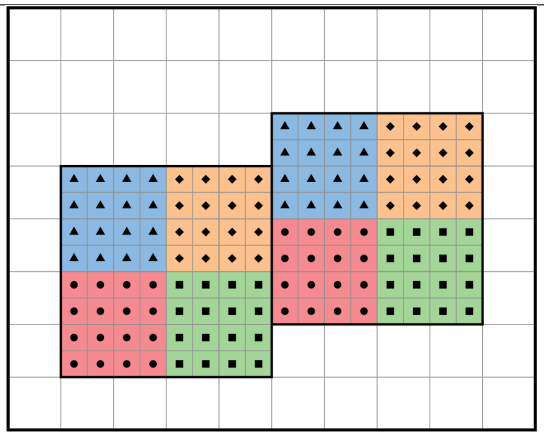
```
//              * true *  turns on tiling
for (MFIter mfi(F,true); mfi.isValid(); ++mfi) // Loop over tiles
{
    //                   tilebox() instead of validbox()
    const Box& box = mfi.tilebox();

    Array4<Real const> const& u = U.const_array(mfi);
    Array4<Real      > const& f = F.array(mfi);
    f2(box, u, f);
}
```

The kernels functions like `f1` and `f2` in the two examples here usually require very little changes.

Table 4.2: Comparison of `MFIter` with (right) and without (left) tiling.



Example of cell-centered valid boxes. There are two valid boxes in this example. Each has 8^2 cells.



Example of cell-centered tile boxes. Each grid is *logically* broken into 4 tiles, and each tile as 4^2 cells. There are 8 tiles in total.

Table 4.2 shows an example of the difference between `validbox` and `tilebox`. In this example, there are two grids of cell-centered index type. The function `validbox` always returns a `Box` for the valid region of an `FArrayBox` no matter whether or not tiling is enabled, whereas the function `tilebox` returns a `Box` for a tile. (Note that when tiling is disabled, `tilebox` returns the same `Box` as `validbox`.) The number of loop iteration is 2 in the non-tiling version, whereas in the tiling version the kernel function is called 8 times.

It is important to use the correct `Box` when implementing tiling, especially if the box is used to define a work region inside of the loop. For example:

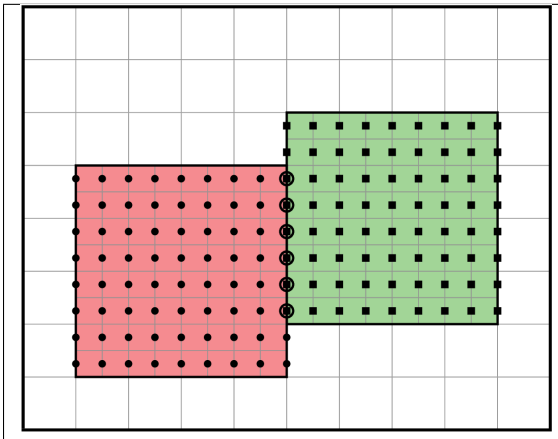
```
// MFIter loop with tiling on.
for (MFIter mfi(mf,true); mfi.isValid(); ++mfi)
{
    Box bx = mfi.validbox();     // Gets box of entire, untiled region.
    calcOverBox(bx);             // ERROR! Works on entire box, not tiled box.
                                 // Other iterations will redo many of the same cells.
}
```

The tile size can be explicitly set when defining `MFIter`.

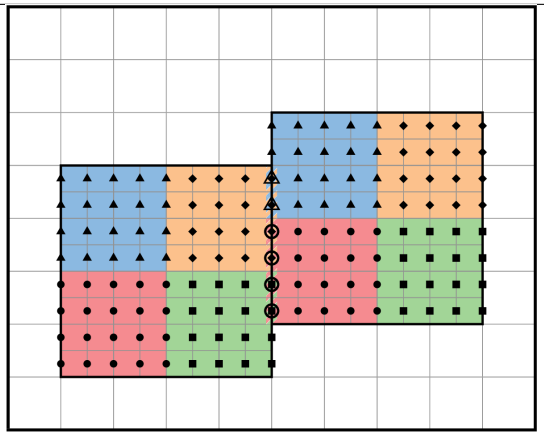
```
// No tiling in x-direction. Tile size is 16 for y and 32 for z.
for (MFIter mfi(mf,IntVect(1024000,16,32)); mfi.isValid(); ++mfi) {...}
```

An `IntVect` is used to specify the tile size for every dimension. A tile size larger than the grid size simply means tiling is disabled in that direction. AMReX has a default tile size `IntVect{1024000,8,8}` in 3D and no tiling in 2D. This is used when tile size is not explicitly set but the tiling flag is on. One can change the default size using `ParmParse` (section *ParmParse*) parameter `fabarray.mfiter_tile_size`.

Table 4.3: Comparison of `MFIter` with (right) and without (left) tiling, for face-centered nodal indexing.



Example of face valid boxes. There are two valid boxes in this example. Each has 9×8 points. Note that points in one `Box` may overlap with points in the other `Box`. However, the memory locations for storing floating point data of those points do not overlap, because they belong to separate FArrayBoxes.



Example of face tile boxes. Each grid is *logically* broken into 4 tiles as indicated by the symbols. There are 8 tiles in total. Some tiles have 5×4 points, whereas others have 4×4 points. Points from different Boxes may overlap, but points from different tiles of the same Box do not.

Dynamic tiling, which runs one box per OpenMP thread, is also available. This is useful when the underlying work cannot benefit from thread parallelization. Dynamic tiling is implemented using the `MFItInfo` object and requires the `MFIter` loop to be defined in an OpenMP parallel region:

```
// Dynamic tiling, one box per OpenMP thread.
// No further tiling details,
//   so each thread works on a single tilebox.
#ifdef AMREX_USE_OMP
#pragma omp parallel
#endif
    for (MFIter mfi(mf,MFItInfo().SetDynamic(true)); mfi.isValid(); ++mfi)
    {
        const Box& bx = mfi.validbox();
        ...
    }
```

Dynamic tiling also allows explicit definition of a tile size:

```
// Dynamic tiling, one box per OpenMP thread.
// No tiling in x-direction. Tile size is 16 for y and 32 for z.
#ifdef AMREX_USE_OMP
#pragma omp parallel
#endif
    for (MFIter mfi(mf,MFItInfo().SetDynamic(true).EnableTiling(1024000,16,32)); mfi.
↪isValid(); ++mfi)
    {
        const Box& bx = mfi.tilebox();
        ...
    }
```

Usually `MFIter` is used for accessing multiple MultiFabs like the second example, in which two MultiFabs, `U` and `F`, use `MFIter` via `operator[]`. These different MultiFabs may have different BoxArrays. For example, `U` might be cell-centered, whereas `F` might be nodal in x-direction and cell in other directions. The `MFIter::validbox` and `tilebox` functions return Boxes of the same type as the `MultiFab` used in defining the `MFIter` (`F` in this example). Table 4.3 illustrates an example of non-cell-centered valid and tile boxes. Besides `validbox` and `tilebox`, `MFIter` has a number of functions returning various Boxes. Examples include,

```
Box fabbox() const;       // Return the Box of the FArrayBox

// Return grown tile box.  By default it grows by the number of
// ghost cells of the MultiFab used for defining the MFIter.
Box growntilebox(int ng=-1000000) const;

// Return tilebox with provided nodal flag as if the MFIter
// is constructed with MultiFab of such flag.
Box tilebox(const IntVect& nodal_flag);
```

It should be noted that the function `growntilebox` does not grow the tile Box like a normal `Box`. Growing a `Box` normally means the Box is extended in every face of every dimension. However, the function `growntilebox` only extends the tile Box in such a way that tiles from the same grid do not overlap. This is the basic design principle of these various tiling functions. Tiling is a way of domain decomposition for work sharing. Overlapping tiles is undesirable because work would be wasted and for multi-threaded codes race conditions could occur.

Table 4.4: Comparing growing cell-type and face-type tile boxes.

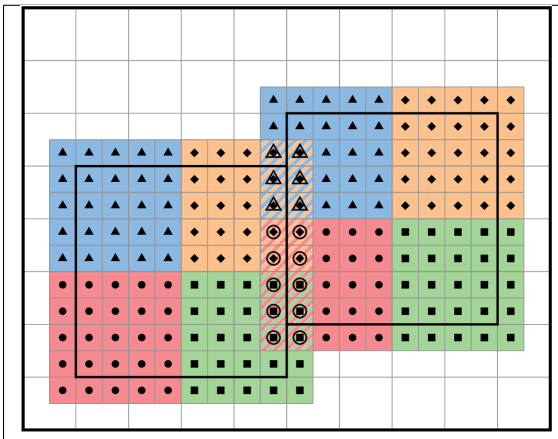
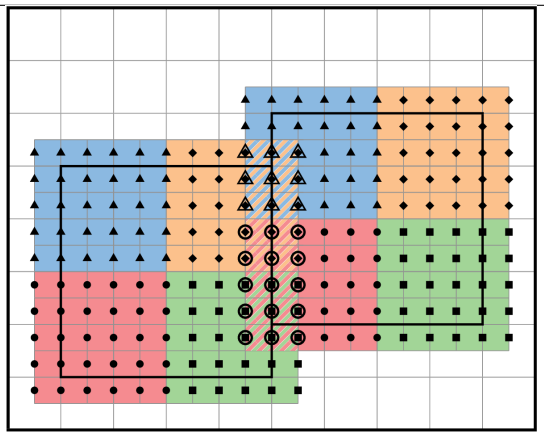
	
Example of cell-centered grown tile boxes. As indicated by symbols, there are 8 tiles and four in each grid in this example. Tiles from the same grid do not overlap. But tiles from different grids may overlap.	Example of face type grown tile boxes. As indicated by symbols, there are 8 tiles and four in each grid in this example. Tiles from the same grid do not overlap even though they have face index type.

Table 4.4 illustrates an example of `growntilebox`. These functions in `MFIter` return `Box` by value. There are three ways of using these functions.

```
const Box& bx = mfi.validbox();  // const& to temporary object is legal

// Make a copy if Box needs to be modified later.
// Compilers can optimize away the temporary object.
Box bx2 = mfi.validbox();
bx2.surroundingNodes();

Box&& bx3 = mfi.validbox(); // bound to the return value
bx3.enclosedCells();
```

But `Box& bx = mfi.validbox()` is not legal and will not compile.

Finally it should be emphasized that tiling should not be used when running on GPUs because of kernel launch overhead.

4.19.3 Multiple MFIters

To avoid some common bugs, it is not allowed to have multiple active `MFIter` objects like below by default.

```
for (MFIter mfi1(...); ...) {
    for (MFIter mfi2(...); ...) {
    }
}
```

```
call amrex_mfiter_build(mf1, ...)
call amrex_mfiter_build(mf2, ...)
```

The will results in an assertion failure at runtime. To disable the assertion, one could call

```
int old_flag = amrex::MFIter::allowMultipleMFIters(true);
```

```
logical :: old_flag
old_flag = amrex_mfiter_allow_multiple(.true.)
```

4.20 Fortran and C++ Kernels

In the section on *MFIter and Tiling*, we have shown that a typical pattern for working with MultiFabs is to use `MFIter` to iterate over the data. In each iteration, a kernel function is called to work on the data and the work region is specified by a `Box`. When tiling is used, the work region is a tile. The tiling is logical in the sense that there is no data layout transformation. The kernel function still gets the whole arrays in `FArrayBox`es, even though it is supposed to work on a tile region of the arrays. We have shown examples of writing kernels in C++ in the previous section. Fortran is also often used for writing these kernels because of its native multi-dimensional array support. To C++, these kernel functions are C functions, whose function signatures are typically declared in a header file named `*_f.H` or `*_F.H`. We recommend the users to follow this convention. Examples of these function declarations are as follows.

```
#include <AMReX_BLFort.H>
#ifdef __cplusplus
extern "C"
{
#endif
    void f1(const int*, const int*, amrex_real*, const int*, const int*);
    void f2(const int*, const int*,
            const amrex_real*, const int*, const int*, const int*
            amrex_real*, const int*, const int*, const int*);
#ifdef __cplusplus
}
#endif
```

These Fortran functions take C pointers and view them as multi-dimensional arrays of the shape specified by the additional integer arguments. Note that Fortran takes arguments by reference unless the **value** keyword is used. So an integer argument on the Fortran side matches an integer pointer on the C++ side. Thanks to Fortran 2003, function name mangling is easily achieved by declaring the Fortran function as **bind**(c).

AMReX provides many macros for passing an FArrayBox's data into Fortran/C. For example

```
for (MFIter mfi(mf,true); mfi.isValid(); ++mfi)
{
    const Box& box = mfi.tilebox();
    f(BL_TO_FORTRAN_BOX(box),
      BL_TO_FORTRAN_ANYD(mf[mfi]));
}
```

Here `BL_TO_FORTRAN_BOX` takes a `Box` and provides two **int** * s specifying the lower and upper bounds of the Box. `BL_TO_FORTRAN_ANYD` takes an `FArrayBox` returned by `mf[mfi]` and the preprocessor turns it into `Real` *, **int** *, **int** *, where `Real` * is the data pointer that matches real array argument in Fortran, the first **int** * (which matches an integer argument in Fortran) specifies the lower bounds, and the second **int** * the upper bounds of the spatial dimensions of the array. An example of the Fortran function is shown below,

```
subroutine f(lo, hi, u, ulo, uhi) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: lo(3),hi(3),ulo(3),uhi(3)
  real(amrex_real),intent(inout)::u(ulo(1):uhi(1),ulo(2):uhi(2),ulo(3):uhi(3))
end subroutine f
```

Here, the size of the integer arrays is 3, the maximal number of spatial dimensions. If the actual spatial dimension is less than 3, the values in the degenerate dimensions are set to zero. So the Fortran function interface does not have to change according to the spatial dimensionality, and the bound of the third dimension of the data array simply becomes `0:0`. With the data passed by `BL_TO_FORTRAN_BOX` and `BL_FORTRAN_ANYD`, this version of Fortran function interface works for any spatial dimensions. If one wants to write a special version just for 2D and would like to use 2D arrays, one can use

```
subroutine f2d(lo, hi, u, ulo, uhi) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: lo(2),hi(2),ulo(2),uhi(2)
  real(amrex_real),intent(inout)::u(ulo(1):uhi(1),ulo(2):uhi(2))
end subroutine f2d
```

Note that this does not require any changes in the C++ part, because when C++ passes an integer pointer pointing to an array of three integers Fortran can treat it as a 2-element integer array.

Another commonly used macro is `BL_TO_FORTRAN`. This macro takes an `FArrayBox` and provides a real pointer for the floating point data array and a number of integer scalars for the bounds. However, the number of the integers depends on the dimensionality. More specifically, there are 6 and 4 integers for 2D and 3D, respectively. The first half of the integers are the lower bounds for each spatial dimension and the second half the upper bounds. For example,

```
subroutine f2d(u, ulo1, ulo2, uhi1, uhi2) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: ulo1, ulo2, uhi1, uhi2
  real(amrex_real),intent(inout)::u(ulo1:uhi1,ulo2:uhi2)
end subroutine f2d

subroutine f3d(u, ulo1, ulo2, ulo3, uhi1, uhi2, uhi3) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: ulo1, ulo2, ulo3, uhi1, uhi2, uhi3
  real(amrex_real),intent(inout)::u(ulo1:uhi1,ulo2:uhi2,ulo3:uhi3)
end subroutine f3d
```

Here for simplicity we have omitted passing the tile Box.

Usually `MultiFab`s have multiple components. Thus we often also need to pass the number of component into Fortran functions. We can obtain the number by calling the `MultiFab::nComp()` function, and pass it to Fortran. We can also use the `BL_TO_FORTRAN_FAB` macro that is similar to `BL_TO_FORTRAN_ANYD` except that it provides an additional `int *` for the number of components. The Fortran function matching `BL_TO_FORTRAN_FAB(fab)` is then like below,

```
subroutine f(u, ulo, uhi,nu) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: lo(3),hi(3),ulo(3),uhi(3),nu
  real(amrex_real),intent(inout)::u(ulo(1):uhi(1),ulo(2):uhi(2),ulo(3):uhi(3),nu)
end subroutine f
```

There is a potential type safety issue when calling Fortran functions from C++. If there is a mismatch between the function declaration on the C++ side and the function definition in Fortran, the compiler cannot catch it. For example

```
// function declaration
extern "C" {
    void f (amrex_real* x);
}

for (MFIter mfi(mf,true); mfi.isValid(); ++mfi)
{
    f(mf[mfi].dataPtr());
}

! Fortran definition
subroutine f(x,y) bind(c)
    implicit none
    integer x, y
end subroutine f
```

The code above will compile without errors even though the number of arguments and types don't match.

To help detect this kind of issues, AMReX provides a type check tool. Note that it only works when GCC is used. In the directory an AMReX based code is compiled, type

```
make typecheck
```

Extra arguments used in a usual AMReX build (e.g., USE_MPI=TRUE DIM=2) can be added. When it finishes, the output may look like,

```
Function my_f in main_F.H vs. Fortran procedure in f.f90
    number of arguments 1 does NOT match 2.
    arg #1: C type ['double', 'pointer'] does NOT match Fortran type ('INTEGER 4',
↪'pointer', 'x').
22 functions checked, 1 error(s) found.  More details can be found in tmp_build_dir/t/3d.
↪gnu.DEBUG.EXE/amrex_typecheck.ou.
```

It should be noted that Fortran by default passes argument by reference. In the example output above, `pointer` in `Fortran type ('INTEGER 4', 'pointer', 'x')` means it's a reference to argument (i.e., C pointer), not a Fortran pointer.

The type check tool has known limitations. For a function to be checked by the tool in the GNU make build system, the declaration must be in a header file named `*_f.H` or `*_F.H`, and the header file must be in the `CEXE_headers` make variable. The headers are preprocessed first by cpp as C language, and is then parsed by pycparser (https://pypi.python.org/pypi/pycparser) that needs to be installed on your system. Because pycparser is a C parser, C++ parts of the headers (e.g., `extern "C" {`) need to be hidden with macro `#ifdef __cplusplus`. Headers like `AMReX_BLFort.H` can be used as a C header, but most other AMReX headers cannot and should be hidden by `#ifdef __cplusplus` if they are included. More details can be found at `amrex/Docs/Readme.typecheck`. Despite these limitations, it is recommended to use the type check tool and report issues to us.

Although Fortran has native multi-dimensional array, we recommend writing kernels in C++ because of performance portability for CPU and GPU. AMReX provides a multi-dimensional array type of syntax, similar to Fortran, that is readable and easy to implement. We have demonstrated how to use `Array4` in previous sections. Because of its importance, we will summarize its basic usage again with the example below.

```
void f (Box const& bx, FArrayBox const& sfab, FArrayBox& dfab)
{
    const Dim3 lo = amrex::lbound(bx);
    const Dim3 hi = amrex::ubound(bx);
```

(continues on next page)

(continued from previous page)

```
    Array4<Real const> const& src = sfab.const_array();
    Array4<Real      > const& dst = dfab2.array();

    for         (int k = lo.z; k <= hi.z; ++k) {
        for     (int j = lo.y; j <= hi.y; ++j) {
            AMREX_PRAGMA_SIMD
            for (int i = lo.x; i <= hi.x; ++i) {
                dst(i,j,k) = 0.5*(src(i,j,k)+src(i+1,j,k));
            }
        }
    }
}

for (MFIter mfi(mf1,true); mfi.isValid(); ++mfi)
{
    const Box& box = mfi.tilebox();
    f(box, mf1[mfi], mf2[mfi]);
}
```

A `Box` and two `FArrayBox`es are passed to a C++ kernel function. In the function, `amrex::lbound` and `amrex::ubound` are called to get the start and end of the loops from `Box::smallEnd()` and `Box::bigEnd` of `bx`. Both functions return a `amrex::Dim3`, a trivial type containing three integers. The individual components are accessed by using `.x`, `.y` and `.z`, as shown in the **`for`** loops.

`BaseFab::array()` is called to obtain an `Array4` object that is designed as an independent, **`operator()`** based accessor to the `BaseFab` data. `Array4` is an AMReX class that contains a pointer to the `FArrayBox` data and two `Dim3` structs that contain the bounds of the `FArrayBox`. The bounds are stored to properly translate the three dimensional coordinates to the appropriate location in the one-dimensional array. `Array4`'s **`operator()`** can also take a fourth integer to access across states of the `FArrayBox`. When AMReX is built for 1D or 2D, it can be used by passing *0* to the missing dimensions.

The `AMREX_PRAGMA_SIMD` macro is placed in the innermost loop to notify the compiler that loop iterations are independent and it is safe to vectorize the loop. This should be done whenever possible to achieve the best performance. Be aware: the macro generates a compiler dependent pragma, so their exact effect on the resulting code is also compiler dependent. It should be emphasized that using the `AMREX_PRAGMA_SIMD` macro on loops that are not safe for vectorization may lead to errors, so if unsure about the independence of the iterations of a loop, test and verify before adding the macro.

These loops should usually use `i <= hi.x`, not `i < hi.x`, when defining the loop bounds. If not, the highest index cells will be left out of the calculation.

4.21 ParallelFor

In the examples so far, we have explicitly written out the for loops when we iterate over a `Box`. AMReX also provides function templates for writing these in a concise and performance portable way like below,

```
#ifdef AMREX_USE_OMP
#pragma omp parallel if (Gpu::notInLaunchRegion())
#endif
  for (MFIter mfi(mfa,TilingIfNotGPU()); mfi.isValid(); ++mfi)
  {
```

(continues on next page)

(continued from previous page)

```
    const Box& bx = mfi.tilebox();
    Array4<Real> const& a = mfa[mfi].array();
    Array4<Real const> const& b = mfb[mfi].const_array();
    Array4<Real const> const& c = mfc[mfi].const_array();
    ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k)
    {
      a(i,j,k) += b(i,j,k) * c(i,j,k);
    });
  }
```

Here, `ParallelFor` takes two arguments. The first argument is a `Box` specifying the iteration index space, and the second argument is a C++ lambda function that works on cell `(i,j,k)`. Variables a, b and c in the lambda function are captured by value from the enclosing scope. The code above is performance portable. It works with and without GPU support. When AMReX is built with GPU support, AMREX_GPU_DEVICE indicates that the lambda function is a device function and `ParallelFor` launches a GPU kernel to do the work. When it is built without GPU support, AMREX_GPU_DEVICE has no effects whatsoever. More details on `ParallelFor` will be presented in section *Launching C++ nested loops*. It should be emphasized that `ParallelFor` does not start an OpenMP parallel region. The OpenMP parallel region will be started by the pragma above the `MFIter` loop if it is built with OpenMP and without enabling GPU. Tiling is turned off if GPU is enabled so that more parallelism is exposed to GPU kernels. Also note that when tiling is off, `tilebox` returns `validbox`.

There are other versions of `ParallelFor`,

```
// 1D for loop
ParallelFor(N, [=] AMREX_GPU_DEVICE (int i) { ... });

// 4D for loop
ParallelFor(box, numcomps,
            [=] AMREX_GPU_DEVICE (int i, int j, int k, int n) { ... });
```

4.22 Ghost Cells

AMReX uses a `MultiFab` as a container for floating point data on multiple Boxes at a single level of refinement. Each rectangular Box has its own boundaries on the low and high side in each coordinate direction. Each Box within a `MultiFab` can have ghost cells for storing data outside the Box's valid region. This allows us to, e.g., perform stencil-type operations on regular arrays. There are three basic types of boundaries:

1. interior boundary
2. coarse/fine boundary
3. physical boundary

Interior boundary is the border among the grid Boxes themselves. For example, in Fig. 4.1, the two blue grid Boxes on level 1 share an interior boundary that is 10 cells long. For a `MultiFab` with ghost cells on level 1, we can use the `MultiFab::FillBoundary` function introduced in the section on *FabArray, MultiFab and iMultiFab* to fill ghost cells at the interior boundary with valid cell data from other Boxes. `MultiFab::FillBoundary` can optionally fill periodic boundary ghost cells as well.

A coarse/fine boundary is the border between two AMR levels. `FillBoundary` does not fill these ghost cells. These ghost cells on the fine level need to be interpolated from the coarse level data. This is a subject that will be discussed in the section on *FillPatchUtil and Interpolater*.

Note that periodic boundary is not considered a basic type in the discussion here because after periodic transformation it becomes either interior boundary or coarse/fine boundary.

The third type of boundary is the physical boundary at the physical domain. Note that both coarse and fine AMR levels could have grids touching the physical boundary. It is up to the application codes to properly fill the ghost cells at the physical boundary. However, AMReX does provide support for some common operations. See the section on *Boundary Conditions* for a discussion on domain boundary conditions in general, including how to implement physical (non-periodic) boundary conditions.

4.23 Boundary Conditions

This section describes how to implement domain boundary conditions in AMReX. A ghost cell that is outside of the valid region can be thought of as either "interior" (which includes periodic and coarse-fine ghost cells), or "physical". Physical boundary conditions can occur on domain boundaries and can be characterized as inflow, outflow, slip/no-slip walls, etc., and are ultimately linked to mathematical Dirichlet or Neumann conditions.

The basic idea behind physical boundary conditions is as follows:

- Create a `BCRec` object, which is essentially a multidimensional integer array of `2*DIM` components. Each component defines a boundary condition type for the lo/hi side of the domain, for each direction. See `amrex/Src/Base/AMReX_BC_TYPES.H` for common physical and mathematical types. Below is an example of setting up a `Vector<BCRec>` for multiple components before the call to ghost cell routines.

```
// Set up BC; see ``amrex/Src/Base/AMReX_BC_TYPES.H`` for supported types
Vector<BCRec> bc(phi.nComp());
for (int n = 0; n < phi.nComp(); ++n)
{
    for (int idim = 0; idim < AMREX_SPACEDIM; ++idim)
    {
        if (geom.isPeriodic(idim))
        {
            bc[n].setLo(idim, BCType::int_dir); // interior
            bc[n].setHi(idim, BCType::int_dir);
        }
        else
        {
            bc[n].setLo(idim, BCType::foextrap); // first-order extrapolation
            bc[n].setHi(idim, BCType::foextrap);
        }
    }
}
```

`amrex::BCType` has the following types,

int_dir
: Interior, including periodic boundary

ext_dir
: "External Dirichlet". It is the user's responsibility to write a routine to fill ghost cells (more details below).

foextrap
: "First Order Extrapolation" First order extrapolation from last cell in interior.

reflect_even
: Reflection from interior cells with sign unchanged, $q(-i) = q(i)$.

reflect_odd
Reflection from interior cells with sign changed, $q(-i) = -q(i)$.

user_1, user_2 and user_3
"User". It is the user's responsibility to write a routine to fill ghost cells (more details below).

- For external Dirichlet and user boundaries, the user needs to provide a callable object like below.

```
struct MyExtBCFill {
    AMREX_GPU_DEVICE
    void operator() (const IntVect& iv, Array4<Real> const& dest,
                     const int dcomp, const int numcomp,
                     GeometryData const& geom, const Real time,
                     const BCRec* bcr, const int bcomp,
                     const int orig_comp) const
    {
        // external Dirichlet or user BC for cell iv
    }
};
```

  Here, for the CPU build, the AMREX_GPU_DEVICE macro has no effect whatsoever, whereas for the GPU build, this marks the operator as a GPU device function.

- It is the user's responsibility to have a consistent definition of what the ghost cells represent. A common option used in AMReX codes is to fill the domain ghost cells with the value that lies on the boundary (as opposed to another common option where the value in the ghost cell represents an extrapolated value based on the boundary condition type). Then in our stencil based "work" codes, we also pass in the `BCRec` object and use modified stencils near the domain boundary that know the value in the first ghost cell represents the value on the boundary.

Depending on the level of complexity of your code, there are various options for filling domain boundary ghost cells.

For single-level codes built from `amrex/Src/Base` (excluding the `amrex/Src/AmrCore` and `amrex/Src/Amr` source code directories), you will have single-level MultiFabs filled with data in the valid region where you need to fill the ghost cells on each grid.

```
MultiFab mf;
Geometry geom;
Vector<BCRec> bc;
Real time;

// ...

// fills interior and periodic domain boundary ghost cells
mf.FillBoundary(geom.periodicity());

// fills physical domain boundary ghost cells for a cell-centered multifab
if (not geom.isAllPeriodic()) {
    GpuBndryFuncFab<MyExtBCFill> bf(MyExtBCFill{});
    PhysBCFunct<GpuBndryFuncFab<MyExtBCFill> > physbcf(geom, bc, bf);
    physbcf(mf, 0, mf.nComp(), mf.nGrowVector(), time, 0);
}
```

4.24 Masks

Given an index (i,j,k), we often need to know its relationship with other points and levels (e.g., whether this point on a coarse level is covered by a fine level, whether this ghost point is outside coarse/fine boundary, etc.). AMReX provides various functions for creating masks for this type of purposes.

4.24.1 Owner Mask

AMReX supports various index types such as face, edge and node, besides cell centered type. For non-cell types, two boxes could overlap. For example, a nodal index (i,j,k) could exist in more than one FArrayBox of a nodal MultiFab. AMReX provides a function to create an owner mask, where the owner is the grid with the lowest grid number containing the data. This has a number of use cases. The nodal data for the same nodal point on different FArrayBoxes may be out of sync. We can use MultiFab::OverrideSync and an owner mask to sync up the data with owners overriding non-owners.

```
MultiFab mf(...); // non-cell-centered
auto mask = amrex::OwnerMask(mf, geom.periodicity());
mf.OverrideSync(*mask, geom.periodicity());
```

To compute the dot product of two nodal MultiFabs, we can use a mask to avoid double counting.

```
MultiFab mf1(...);
MultiFab mf2(...);
auto mask = amrex::OwnerMask(mf1, geom.periodicity());
Real result = MultiFab::Dot(*mask, mf1, 0, mf2, 0, 1, 0);
```

4.24.2 Overlap Mask

For the synchronization example mentioned previously, maybe instead of overriding, we want to do averaging. This can be achieved with an overlap mask indicating how many duplicates are in each point. The code below shows how the MultiFab::AverageSync function is implemented in AMReX.

```
MultiFab mf(...); // non-cell-centered
auto mask = mf.OverlapMask(geom.periodicity());
mask->invert(1.0, 0, 1);
mf.WeightedSync(*mask, geom.periodicity());
```

4.24.3 Point Mask

The FabArray class has a member function BuildMask that can be used to set masks indicating the type of points (e.g., valid, outside the domain, etc.). For example,

```
iMultiFab mask(ba, dm, 1, nghost);
int a = 10; // ghost points covered by valid points
int b = 11; // ghost points not covered by valid points
int c = 12; // outside physical domain
int d = 13; // interior points (i.e., valid points)
mask.BuildMask(geom.Domain(), geom.periodicity(), a, b, c, d);
```

4.24.4 Fine Mask

AMReX provides a number of makeFineMask functions that can be useful for multi-level AMR calculations. For example, we may want to compute the infinity norm on a coarse AMR level without including data from cells covered by fine level grids.

```
int coarse_value = 1;
int fine_value = 0;
iMultiFab mask = makeFineMask(coarse_mf, fine_boxarray, refine_ratio,
                              coarse_value, fine_value);
Real result = coarse_mf.norminf(mask);
```

4.25 Memory Allocation

Some constructors of MultiFab, FArrayBox, etc. can take an Arena argument for memory allocation. This is usually not important for CPU codes, but very important for GPU codes. We will present more details in *Memory Allocation* in Chapter GPU.

AMReX has a Fortran module, amrex_mempool_module that can be used to allocate memory for Fortran pointers. The reason that such a module exists in AMReX is that memory allocation is often very slow in multi-threaded OpenMP parallel regions. AMReX amrex_mempool_module provides a much faster alternative approach, in which each thread has its own memory pool. Here are examples of using the module.

```
use amrex_mempool_module, only : amrex_allocate, amrex_deallocate
real(amrex_real), pointer, contiguous :: a(:,:,:), b(:,:,:,:)
integer :: lo1, hi1, lo2, hi2, lo3, hi3, lo(4), hi(4)
! lo1 = ...
! a(lo1:hi1, lo2:hi2, lo3:hi3)
call amrex_allocate(a, lo1, hi1, lo2, hi2, lo3, hi3)
! b(lo(1):hi(1),lo(2):hi(2),lo(3):hi(3),lo(4):hi(4))
call amrex_allocate(b, lo, hi)
! ......
call amrex_deallocate(a)
call amrex_deallocate(b)
```

The downside of this is we have to use **pointer** instead of **allocatable**. This means we must explicitly free the memory via amrex_deallocate and we need to declare the pointers as **contiguous** for performance reason. Also, we often pass the Fortran pointer to a procedure with explicit array argument to get rid of the pointerness completely.

4.26 Abort, Assertion and Backtrace

amrex::Abort(const char * message) is used to terminate a run usually when something goes wrong. This function takes a message and writes it to stderr. Files named like Backtrace.1 (where 1 means process 1) are produced containing backtrace information of the call stack. In Fortran, we can call amrex_abort from the amrex_error_module, which takes a Fortran character variable with assumed size (i.e., len=*) as a message. A ParmParse runtime boolean parameter amrex.throw_handling (which is defaulted to 0, i.e., false) can be set to 1 (i.e., true) so that AMReX will throw an exception instead of aborting.

AMREX_ASSERT is a macro that takes a Boolean expression. For debug build (e.g., DEBUG=TRUE using the GNU Make build system), if the expression at runtime is evaluated to false, amrex::Abort will be called and the run is thus terminated. For optimized build (e.g., DEBUG=FALSE using the GNU Make build system), the AMREX_ASSERT statement

is removed at compile time and thus has no effect at runtime. We often use this as a means of putting debug statement in the code without adding any extra cost for production runs. For example,

```
AMREX_ASSERT(mf.nGrow() > 0 && mf.nComp() == mf2.nComp());
```

Here for debug build we like to assert that `MultiFab mf` has ghost cells and it also has the same number of components as `MultiFab mf2`. If we always want the assertion, we can use `AMREX_ALWAYS_ASSERT`. The assertion macros have a `_WITH_MESSAGE` variant that will print a message when assertion fails. For example,

```
AMREX_ASSERT_WITH_MESSAGE(mf.boxArray() == mf2.boxArray(),
                          "These two mfs must have the same BoxArray");
```

Backtrace files are produced by AMReX signal handler by default when segfault occurs or `Abort` is called. If the application does not want AMReX to handle this, `ParmParse` parameter *amrex.signal_handling=0* can be used to disable it.

CHAPTER

FIVE

GRIDDING AND LOAD BALANCING

AMReX provides a great deal of generality when it comes to how to decompose the computational domain into individual logically rectangular grids, and how to distribute those grids to MPI ranks. We use the phrase "load balancing" here to refer to the combined process of grid creation (and re-creation when regridding) and distribution of grids to MPI ranks.

Even for single-level calculations, AMReX provides the flexibility to have different size grids, more than one grid per MPI rank, and different strategies for distributing the grids to MPI ranks.

For multi-level calculations, the same principles for load balancing apply as in single-level calculations, but there is additional complexity in how to tag cells for refinement and how to create the union of grids at levels > 0 where that union most likely does not cover the computational domain.

See *Grid Creation* for grids are created, i.e. how the `BoxArray` on which `MultiFabs` will be built is defined at each level.

See *Load Balancing* for the strategies AMReX supports for distributing grids to MPI ranks, i.e. defining the `DistributionMapping` with which `MultiFabs` at that level will be built.

We also note that we can create separate grids, and map them in different ways to MPI ranks, for different types of data in a single calculation. We refer to this as the "dual grid approach" and the most common usage is to load balance mesh and particle data separately. See *Dual Grid Approach* for more about this approach.

When running on multicore machines with OpenMP, we can also control the distribution of work by setting the size of grid tiles (by defining `fabarray_mfiter.tile_size`), and if relevant, of particle tiles (by defining `particle.tile_size`). We can also specify the strategy for assigning tiles to OpenMP threads. See *MFIter with Tiling* for more about tiling.

5.1 Grid Creation

To run an AMReX-based application you must specify the domain size by specifying `n_cell` – this is the number of cells spanning the domain in each coordinate direction at level 0.

Users often specify `max_grid_size` as well. The default load balancing algorithm then divides the domain in every direction so that each grid is no longer than `max_grid_size` in that direction. If not specified by the user, `max_grid_size` defaults to 128 in 2D and 32 in 3D (in each coordinate direction).

Another popular input is `blocking_factor`. The value of `blocking_factor` constrains grid creation in that in that each grid must be divisible by `blocking_factor`. Note that both the domain (at each level) and `max_grid_size` must be divisible by `blocking_factor`, and that `blocking_factor` must be either 1 or a power of 2 (otherwise the gridding algorithm would not in fact create grids divisible by `blocking_factor` because of how `blocking_factor` is used in the gridding algorithm).

If not specified by the user, `blocking_factor` defaults to 8 in each coordinate direction. The typical purpose of `blocking_factor` is to ensure that the grids will be sufficiently coarsenable for good multigrid performance.

There is one more default behavior to be aware of. There is a boolean `refine_grid_layout` that defaults to true but can be over-ridden at run-time. If `refine_grid_layout` is true and the number of grids created is less than the number of processors (Ngrids < Nprocs), then grids will be further subdivided until Ngrids >= Nprocs.

Caveat: if subdividing the grids to achieve Ngrids >= Nprocs would violate the `blocking_factor` criterion then additional grids are not created and the number of grids will remain less than the number of processors

Note that `n_cell` must be given as three separate integers, one for each coordinate direction.

However, `max_grid_size` and `blocking_factor` can be specified as a single value applying to all coordinate directions, or as separate values for each direction.

- If `max_grid_size` (or `blocking_factor`) is specified as multiple integers then the first integer applies to level 0, the second to level 1, etc. If you don't specify as many integers as there are levels, the final value will be used for the remaining levels.
- If different values of `max_grid_size` (or `blocking_factor`) are wanted for each coordinate direction, then `max_grid_size_x`, `max_grid_size_y` and `max_grid_size_z` (or `blocking_factor_x`, `blocking_factor_y` and `blocking_factor_z`) must be used. If you don't specify as many integers as there are levels, the final value will be used for the remaining levels.

Additional notes:

- To create identical grids of a specific size, e.g. of length m in each direction, then set `max_grid_size` = m and `blocking_factor` = m.
- Note that `max_grid_size` is just an upper bound; with `n_cell = 48` and `max_grid_size = 32`, we will typically have one grid of length 32 and one of length 16.

The grid creation process at level 0 proceeds as follows (if not using the KD-tree approach):

1. The domain is initially defined by a single grid of size `n_cell`.
2. If `n_cell` is greater than `max_grid_size` then the grids are subdivided until each grid is no longer than `max_grid_size` cells on each side. The `blocking_factor` criterion (ie that the length of each side of each grid is divisible by `blocking_factor` in that direction) is satisfied during this process.
3. Next, if `refine_grid_layout = true` and there are more processors than grids at this level, then the grids at this level are further divided until Ngrids >= Nprocs (unless doing so would violate the `blocking_factor` criterion).

The creation of grids at levels > 0 begins by tagging cells at the coarser level and follows the Berger-Rigoutsos clustering algorithm with the additional constraints of satisfying the `blocking_factor` and `max_grid_size` criteria. An additional parameter becomes relevant here: the "grid efficiency", specified as `amr.grid_eff` in the inputs file. This threshold value, which defaults to 0.7 (or 70%), is used to ensure that grids do not contain too large a fraction of untagged cells. We note that the grid creation process attempts to satisfy the `amr.grid_eff` constraint but will not do so if it means violating the `blocking_factor` criterion.

Users often like to ensure that coarse/fine boundaries are not too close to tagged cells; the way to do this is to set `amr.n_error_buf` to a large integer value (the default is 1). This parameter is used to increase the number of tagged cells before the grids are defined; if cell "*(i,j,k)*" satisfies the tagging criteria, then, for example, if `amr.n_error_buf` is 3, all cells in the 7x7x7 box from lower corner "*(i-3,j-3,k-3)*" to "*(i+3,j+3,k+3)*" will be tagged.

5.2 Dual Grid Approach

In AMReX-based applications that have both mesh data and particle data, the mesh work and particle work have very different requirements for load balancing.

Rather than using a combined work estimate to create the same grids for mesh and particle data, we have the option to pursue a "dual grid" approach.

With this approach the mesh (`MultiFab`) and particle (`ParticleContainer`) data are allocated on different `BoxArrays` with different `DistributionMappings`.

This enables separate load balancing strategies to be used for the mesh and particle work.

The cost of this strategy, of course, is the need to copy mesh data onto temporary `MultiFabs` defined on the particle `BoxArrays` when mesh-particle communication is required.

5.3 Load Balancing

The process of load balancing is typically independent of the process of grid creation; the inputs to load balancing are a given set of grids with a set of weights assigned to each grid. (The exception to this is the KD-tree approach in which the grid creation process is governed by trying to balance the work in each grid.)

Single-level load balancing algorithms are sequentially applied to each AMR level independently, and the resulting distributions are mapped onto the ranks taking into account the weights already assigned to them (assign heaviest set of grids to the least loaded rank). Note that the load of each process is measured by how much memory has already been allocated, not how much memory will be allocated. Therefore the following code is not recommended because it tends to generate non-optimal distributions.

```
for (int lev = 0; lev < nlevels; ++lev) {
    // build DistributionMapping for Level lev
}
for (int lev = 0; lev < nlevels; ++lev) {
    // build MultiFabs for Level lev
}
```

Instead, one should do,

```
for (int lev = 0; lev < nlevels; ++lev) {
    // build DistributionMapping for Level lev
    // build MultiFabs for Level lev
}
```

Distribution options supported by AMReX include the following; the default is SFC:

- Knapsack: the default weight of a grid in the knapsack algorithm is the number of grid cells, but AMReX supports the option to pass an array of weights – one per grid – or alternatively to pass in a MultiFab of weights per cell which is used to compute the weight per grid.
- SFC: enumerate grids with a space-filling Z-morton curve, then partition the resulting ordering across ranks in a way that balances the load.
- Round-robin: sort grids and assign them to ranks in round-robin fashion – specifically FAB i is owned by CPU i%N where N is the total number of MPI ranks.

CHAPTER

SIX

AMRCORE SOURCE CODE

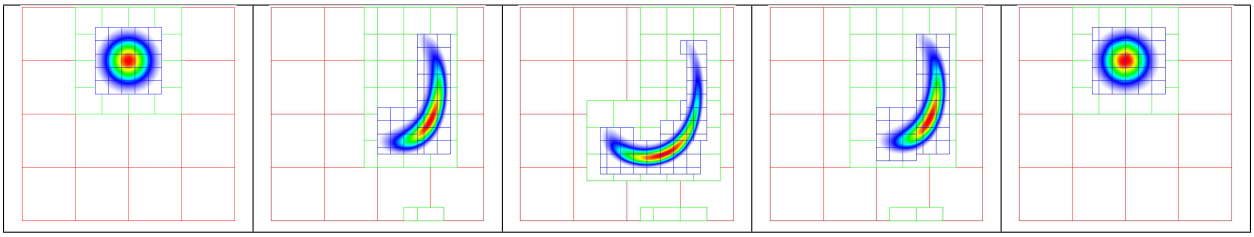
In this Chapter we give an overview of functionality contained in the `amrex/Src/AmrCore` source code. This directory contains source code for the following:

- Storing information about the grid layout and processor distribution mapping at each level of refinement.
- Functions to create grids at different levels of refinement, including tagging operations.
- Operations on data at different levels of refinement, such as interpolation and restriction operators.
- Flux registers used to store and manipulate fluxes at coarse-fine interfaces.
- Particle support for AMR (see *Particles*).

There is another source directory, `amrex/Src/Amr/`, which contains additional classes used to manage the time-stepping for AMR simulations. However, it is possible to build a fully adaptive, subcycling-in-time simulation code without these additional classes.

In this Chapter, we restrict our use to the `amrex/Src/AmrCore` source code and present a tutorial that performs an adaptive, subcycling-in-time simulation of the advection equation for a passively advected scalar. The accompanying tutorial code is available in `amrex-tutorials/ExampleCodes/Amr/Advection_AmrCore` with build/run directory `Exec/SingleVortex`. In this example, the velocity field is a specified function of space and time, such that an initial Gaussian profile is displaced but returns to its original configuration at the final time. The boundary conditions are periodic and we use a refinement ratio of $r = 2$ between each AMR level. The results of the simulation in two-dimensions are depicted in the Table showing the *SingleVortex Tutorial*.

Table 6.1: Time sequence ($t = 0, 0.5, 1, 1.5, 2$ s) of advection of a Gaussian profile using the SingleVortex tutorial. The analytic velocity field distorts the profile, and then restores the profile to the original configuration. The red, green, and blue boxes indicate grids at AMR levels $\ell = 0, 1$, and 2.



6.1 AmrCore Source Code: Details

Here we provide more information about the source code in `amrex/Src/AmrCore`.

6.1.1 AmrMesh and AmrCore

For single-level simulations (see e.g., `amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX1_C/main.cpp`) the user needs to build `Geometry`, `DistributionMapping`, and `BoxArray` objects associated with the simulation. For simulations with multiple levels of refinement, the `AmrMesh` class can be thought of as a container to store arrays of these objects (one for each level), and information about the current grid structure.

`amrex/Src/AmrCore/AMReX_AmrMesh.cpp/H` contains the `AmrMesh` class. The protected data members are:

```
protected:
    int             verbose;
    int             max_level;       // Maximum allowed level.
    Vector<IntVect> ref_ratio;        // Refinement ratios [0:finest_level-1]

    int             finest_level;    // Current finest level.

    Vector<IntVect> n_error_buf;      // Buffer cells around each tagged cell.
    Vector<IntVect> blocking_factor; // Blocking factor in grid generation
                                     // (by level).
    Vector<IntVect> max_grid_size;    // Maximum allowable grid size (by level).
    Real            grid_eff;        // Grid efficiency.
    int             n_proper;        // # cells required for proper nesting.

    bool use_fixed_coarse_grids;
    int  use_fixed_upto_level;
    bool refine_grid_layout;         // chop up grids to have the number of
                                     // grids no less the number of procs

    Vector<Geometry>            geom;
    Vector<DistributionMapping> dmap;
    Vector<BoxArray>            grids;
```

The following parameters are frequently set via the inputs file or the command line. Their usage is described in the section on *Grid Creation*

Table 6.2: AmrCore parameters

Variable	Value	Default
amr.verbose	int	0
amr.max_level	int	none
amr.max_grid_size	ints	32 in 3D, 128 in 2D
amr.n_proper	int	1
amr.grid_eff	Real	0.7
amr.n_error_buf	int	1
amr.blocking_factor	int	8
amr.refine_grid_layout	int	true

AMReX_AmrCore.cpp/H contains the pure virtual class `AmrCore`, which is derived from the `AmrMesh` class. AmrCore does not actually have any data members, just additional member functions, some of which override the base class

AmrMesh.

There are no pure virtual functions in `AmrMesh`, but there are 5 pure virtual functions in the `AmrCore` class. Any applications you create must implement these functions. The tutorial code Amr/Advection_AmrCore provides sample implementation in the derived class `AmrCoreAdv`.

```
//! Tag cells for refinement.  TagBoxArray tags is built on level lev grids.
virtual void ErrorEst (int lev, TagBoxArray& tags, Real time,
                       int ngrow) override = 0;

//! Make a new level from scratch using provided BoxArray and DistributionMapping.
//! Only used during initialization.
virtual void MakeNewLevelFromScratch (int lev, Real time, const BoxArray& ba,
                                      const DistributionMapping& dm) override = 0;

//! Make a new level using provided BoxArray and DistributionMapping and fill
//  with interpolated coarse level data.
virtual void MakeNewLevelFromCoarse (int lev, Real time, const BoxArray& ba,
                                     const DistributionMapping& dm) = 0;

//! Remake an existing level using provided BoxArray and DistributionMapping
//  and fill with existing fine and coarse data.
virtual void RemakeLevel (int lev, Real time, const BoxArray& ba,
                          const DistributionMapping& dm) = 0;

//! Delete level data
virtual void ClearLevel (int lev) = 0;
```

Refer to the `AmrCoreAdv` class in the `amrex-tutorials/ExampleCodes/Amr/AmrCore_Advection/Source` code for a sample implementation.

6.1.2 TagBox, and Cluster

These classes are used in the grid generation process. The `TagBox` class is essentially a data structure that marks which cells are "tagged" for refinement. `Cluster` (and `ClusterList` contained within the same file) are classes that help sort tagged cells and generate a grid structure that contains all the tagged cells. These classes and their member functions are largely hidden from any application codes through simple interfaces such as `regrid` and `ErrorEst` (a routine for tagging cells for refinement).

6.1.3 FillPatchUtil and Interpolater

Many codes, including the Advection_AmrCore example, contain an array of MultiFabs (one for each level of refinement), and then use "fillpatch" operations to fill temporary MultiFabs that may include a different number of ghost cells. Fillpatch operations fill all cells, valid and ghost, from actual valid data at that level, space-time interpolated data from the next-coarser level, neighboring grids at the same level, and domain boundary conditions (for examples that have non-periodic boundary conditions). Note that at the coarsest level, the interior and domain boundary (which can be periodic or prescribed based on physical considerations) need to be filled. At the non-coarsest level, the ghost cells can also be interior or domain, but can also be at coarse-fine interfaces away from the domain boundary. `AMReX_FillPatchUtil.cpp/H` contains two primary functions of interest.

1. `FillPatchSingleLevel()` fills a `MultiFab` and its ghost region at a single level of refinement. The routine is flexible enough to interpolate in time between two MultiFabs associated with different times.

2. `FillPatchTwoLevels()` fills a `MultiFab` and its ghost region at a single level of refinement, assuming there is an underlying coarse level. This routine is flexible enough to interpolate the coarser level in time first using `FillPatchSingleLevel()`.

Note that `FillPatchSingleLevel()` and `FillPatchTwoLevels()` call the single-level routines `MultiFab::FillBoundary` and `FillDomainBoundary()` to fill interior, periodic, and physical boundary ghost cells. In principle, you can write a single-level application that calls `FillPatchSingleLevel()` instead of using `MultiFab::FillBoundary` and `FillDomainBoundary()`.

A `FillPatchUtil` uses an `Interpolator`. This is largely hidden from application codes. AMReX_Interpolater.cpp/H contains the virtual base class `Interpolater`, which provides an interface for coarse-to-fine spatial interpolation operators. The fillpatch routines described above require an Interpolater for FillPatchTwoLevels(). Within AMReX_Interpolater.cpp/H are the derived classes:

- `NodeBilinear`
- `CellBilinear`
- `CellConservativeLinear`
- `CellConservativeProtected`
- `CellConservativeQuartic`
- `CellQuadratic`
- `PCInterp`
- `FaceLinear`
- `FaceDivFree`

These Interpolaters can be executed on CPU or GPU, with certain limitations:

- `CellConservativeProtected` only works in 2D and 3D.
- `CellQuadratic` only works in 2D and 3D.
- `CellConservativeQuartic` only works with a refinement ratio of 2.
- `FaceDivFree` only works in 2D and 3D and with a refinement ratio of 2.

6.1.4 Using FluxRegisters

AMReX_FluxRegister.cpp/H contains the class `FluxRegister`, which is derived from the class `BndryRegister` (in `amrex/Src/Boundary/AMReX_BndryRegister`). In the most general terms, a FluxRegister is a special type of BndryRegister that stores and manipulates data (most often fluxes) at coarse-fine interfaces. A simple usage scenario comes from a conservative discretization of a hyperbolic system:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \mathbf{F} \rightarrow \frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n}}{\Delta t} = \frac{F_{i+1/2,j} - F_{i-1/2,j}}{\Delta x} + \frac{F_{i,j+1/2} - F_{i,j-1/2}}{\Delta y}.$$

Consider a two-level, two-dimensional simulation. A standard methodology for advancing the solution in time is to first advance the coarse grid solution ignoring the fine level, and then advance the fine grid solution using the coarse level only to supply boundary conditions. At the coarse-fine interface, the area-weighted fluxes from the fine grid advance do not in general match the underlying flux from the coarse grid face, resulting in a lack of global conservation. Note that for subcycling-in-time algorithms (where for each coarse grid advance, the fine grid is advanced r times using a coarse grid time step reduced by a factor of r, where r is the refinement ratio), the coarse grid flux must be compared to the area *and* time-weighted fine grid fluxes. A `FluxRegister` accumulates and ultimately stores the net difference in fluxes between the coarse grid and fine grid advance over each face over a given coarse time step. The simplest

possible synchronization step is to modify the coarse grid solution in coarse cells immediately adjacent to the coarse-fine interface are updated to account for the mismatch stored in the FluxRegister. This can be done "simply" by taking the coarse-level divergence of the data in the FluxRegister using the `reflux` function.

The Fortran routines that perform the actual floating point work associated with incrementing data in a `FluxRegister` are contained in the files AMReX_FLUXREG_F.H and AMReX_FLUXREG_xD.F.

6.1.5 AmrParticles and AmrParGDB

The AmrCore/ directory contains derived classes for dealing with particles in a multi-level framework. The description of the base classes are given in the chapter on *Particles*.

AMReX_AmrParticles.cpp/H contains the classes `AmrParticleContainer` and `AmrTracerParticleContainer`, which are derived from the classes `ParticleContainer` (in `amrex/Src/Particle/AMReX_Particles`) and `TracerParticleContainer` (in `amrex/Src/Particle/AMReX_TracerParticles`).

AMReX_AmrParGDB.cpp/H contains the class `AmrParGDB`, which is derived from the class `ParGDBBase` (in `amrex/Src/Particle/AMReX_ParGDB`).

6.2 Example: Advection_AmrCore

6.2.1 The Advection Equation

We seek to solve the advection equation on a multi-level, adaptive grid structure:

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{U}).$$

The velocity field is a specified divergence-free (so the flow field is incompressible) function of space and time. The initial scalar field is a Gaussian profile. To integrate these equations on a given level, we use a simple conservative update,

$$\frac{\phi^{n+1}_{i,\,j} - \phi^{n}_{i,\,j}}{\Delta t} = \frac{(\phi u)^{n+1/2}_{i+1/2,\,j} - (\phi u)^{n+1/2}_{i-1/2,\,j}}{\Delta x} + \frac{(\phi v)^{n+1/2}_{i,\,j+1/2} - (\phi v)^{n+1/2}_{i,\,j-1/2}}{\Delta y},$$

where the velocities on faces are prescribed functions of space and time, and the scalars on faces are computed using a Godunov advection integration scheme. The fluxes in this case are the face-centered, time-centered "ϕu" and "ϕv" terms.

We use a subcycling-in-time approach where finer levels are advanced with smaller time steps than coarser levels, and then synchronization is later performed between levels. More specifically, the multi-level procedure can most easily be thought of as a recursive algorithm in which, to advance level ℓ, $0 \le \ell \le \ell_{\max}$, the following steps are taken:

- Advance level ℓ in time by one time step, Δt^{ℓ}, as if it is the only level. If $\ell > 0$, obtain boundary data (i.e. fill the level ℓ ghost cells) using space- and time-interpolated data from the grids at $\ell - 1$ where appropriate.
- If $\ell < \ell_{\max}$
  - Advance level $(\ell + 1)$ for r time steps with $\Delta t^{\ell+1} = \frac{1}{r}\Delta t^{\ell}$.
  - Synchronize the data between levels ℓ and $\ell + 1$.

Specifically, for a 3-level simulation, depicted graphically in the figure showing the *Schematic of subcycling-in-time algorithm.* above:

1. Integrate $\ell = 0$ over Δt.

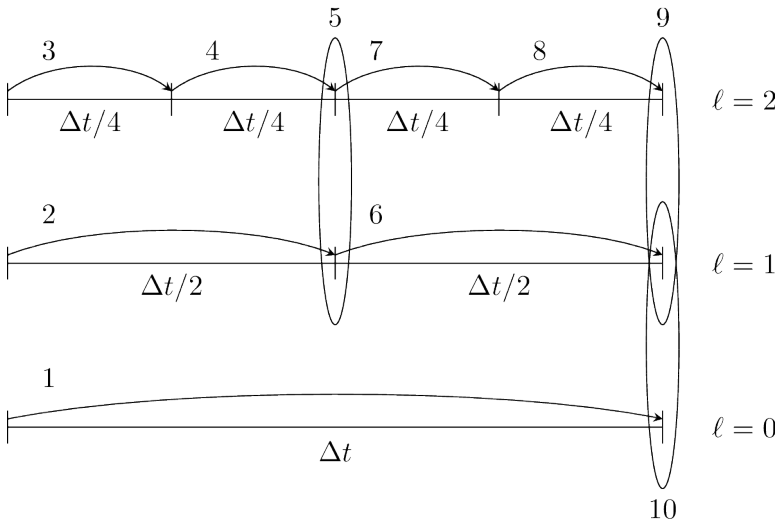


Fig. 6.1: Schematic of subcycling-in-time algorithm.

2. Integrate $\ell = 1$ over $\Delta t/2$.
3. Integrate $\ell = 2$ over $\Delta t/4$.
4. Integrate $\ell = 2$ over $\Delta t/4$.
5. Synchronize levels $\ell = 1, 2$.
6. Integrate $\ell = 1$ over $\Delta t/2$.
7. Integrate $\ell = 2$ over $\Delta t/4$.
8. Integrate $\ell = 2$ over $\Delta t/4$.
9. Synchronize levels $\ell = 1, 2$.
10. Synchronize levels $\ell = 0, 1$.

For the scalar field, we keep track volume and time-weighted fluxes at coarse-fine interfaces. We accumulate area and time-weighted fluxes in `FluxRegister` objects, which can be thought of as special boundary FABsets associated with coarse-fine interfaces. Since the fluxes are area and time-weighted (and sign-weighted, depending on whether they come from the coarse or fine level), the flux registers essentially store the extent by which the solution does not maintain conservation. Conservation only happens if the sum of the (area and time-weighted) fine fluxes equals the coarse flux, which in general is not true.

The idea behind the level $\ell/(\ell+1)$ synchronization step is to correct for sources of mismatch in the composite solution:

1. The data at level ℓ that underlie the level $\ell + 1$ data are not synchronized with the level $\ell + 1$ data. This is simply corrected by overwriting covered coarse cells to be the average of the overlying fine cells.
2. The area and time-weighted fluxes from the level ℓ faces and the level $\ell + 1$ faces do not agree at the $\ell/(\ell + 1)$ interface, resulting in a loss of conservation. The remedy is to modify the solution in the coarse cells immediately next to the coarse-fine interface to account for the mismatch stored in the flux register (computed by taking the coarse-level divergence of the flux register data).

6.2.2 Code Structure

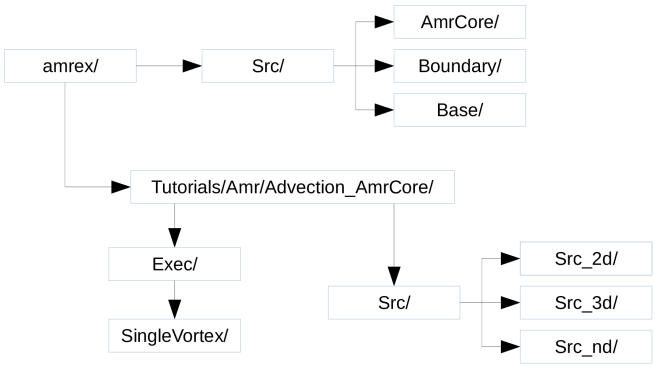


Fig. 6.2: Source code tree for the AmrAdvection_AmrCore example.

The figure shows the *Source code tree for the AmrAdvection_AmrCore example.*

- amrex/Src/
  - Base/ Base amrex library.
  - Boundary/ An assortment of classes for handling boundary data.
  - AmrCore/ AMR data management classes, described in more detail above.
- `Advection_AmrCore/Src` Source code specific to this example. Most notably is the `AmrCoreAdv` class, which is derived from `AmrCore`. The subdirectories `Src_2d` and `Src_3d` contain dimension specific routines. `Src_nd` contains dimension-independent routines.
- Exec Contains a makefile so a user can write other examples besides SingleVortex.
- SingleVortex Build the code here by editing the GNUmakefile and running make. There is also problem-specific source code here used for initialization or specifying the velocity field used in this simulation.

Here is a high-level pseudo-code of the flow of the program:

```
/* Advection_AmrCore Pseudocode */
main()
  AmrCoreAdv amr_core_adv; // build an AmrCoreAdv object
  amr_core_adv.InitData()  // initialize data all all levels
    AmrCore::InitFromScratch()
    AmrMesh::MakeNewGrids()
    AmrMesh::MakeBaseGrids() // define level 0 grids
    AmrCoreAdv::MakeNewLevelFromScratch()
    /* allocate phi_old, phi_new, t_new, and flux registers */
    initdata()  // fill phi
    if (max_level > 0) {
          do {
        AmrMesh::MakeNewGrids()
          /* construct next finer grid based on tagging criteria */
        AmrCoreAdv::MakeNewLevelFromScratch()
             /* allocate phi_old, phi_new, t_new, and flux registers */
```

(continues on next page)

(continued from previous page)

```
            initdata()  // fill phi
      } while (finest_level < max_level);
    }
  amr_core_adv.Evolve()
    loop over time steps {
      ComputeDt()
      timeStep() // advance a level
        /* check regrid conditions and regrid if necessary */
        Advance()
          /* copy phi into a MultiFab and fill ghost cells */
          /* advance phi */
          /* update flux registers */
        if (lev < finest_level) {
          timeStep() // recursive call to advance the next-finer level "r" times
            /* check regrid conditions and regrid if necessary */
            Advance()
              /* copy phi into a MultiFab and fill ghost cells */
              /* advance phi */
              /* update flux registers */
          reflux() // synchronize lev and lev+1 using FluxRegister divergence
          AverageDown() // set covered coarse cells to be the average of fine
        }
    }
```

6.2.3 The AmrCoreAdv Class

This example uses the class `AmrCoreAdv`, which is derived from the class `AmrCore` (which is derived from `AmrMesh`). The function definitions/implementations are given in AmrCoreAdv.H/cpp.

6.2.4 FluxRegisters

The function `AmrCoreAdv::Advance()` calls the Fortran subroutine, `advect` (in `./Src_xd/Adv_xd.f90`). `advect` computes and returns the time-advanced state as well as the fluxes used to update the state. These fluxes are used to set or increment the flux registers.

```
// increment or decrement the flux registers by area and time-weighted fluxes
// Note that the fluxes have already been scaled by dt and area
// In this example we are solving phi_t = -div(+F)
// The fluxes contain, e.g., F_{i+1/2,j} = (phi*u)_{i+1/2,j}
// Keep this in mind when considering the different sign convention for updating
// the flux registers from the coarse or fine grid perspective
// NOTE: the flux register associated with flux_reg[lev] is associated
// with the lev/lev-1 interface (and has grid spacing associated with lev-1)
if (do_reflux) {
   if (flux_reg[lev+1]) {
      for (int i = 0; i < BL_SPACEDIM; ++i) {
          flux_reg[lev+1]->CrseInit(fluxes[i],i,0,0,fluxes[i].nComp(), -1.0);
      }
   }
   if (flux_reg[lev]) {
```

(continues on next page)

(continued from previous page)

```
      for (int i = 0; i < BL_SPACEDIM; ++i) {
          flux_reg[lev]->FineAdd(fluxes[i],i,0,0,fluxes[i].nComp(), 1.0);
      }
   }
}
```

The synchronization is performed at the end of `AmrCoreAdv::timeStep`:

```
if (do_reflux)
{
    // update lev based on coarse-fine flux mismatch
    flux_reg[lev+1]->Reflux(*phi_new[lev], 1.0, 0, 0, phi_new[lev]->nComp(),
                            geom[lev]);
}

AverageDownTo(lev); // average lev+1 down to lev
```

6.2.5 Regridding

The regrid function belongs to the `AmrCore` class (it is virtual – in this tutorial we use the instance in `AmrCore`).

At the beginning of each time step, we check whether we need to regrid. In this example, we use a `regrid_int` and keep track of how many times each level has been advanced. When any given particular level $\ell < \ell_{\max}$ has been advanced a multiple of `regrid_int`, we call the `regrid` function.

```
void
AmrCoreAdv::timeStep (int lev, Real time, int iteration)
{
    if (regrid_int > 0)  // We may need to regrid
    {
        // regrid changes level "lev+1" so we don't regrid on max_level
        if (lev < max_level && istep[lev])
        {
            if (istep[lev] % regrid_int == 0)
            {
                // regrid could add newly refine levels
                // (if finest_level < max_level)
                // so we save the previous finest level index
        int old_finest = finest_level;
        regrid(lev, time);

                // if there are newly created levels, set the time step
        for (int k = old_finest+1; k <= finest_level; ++k) {
            dt[k] = dt[k-1] / MaxRefRatio(k-1);
        }
        }
    }
    }
```

Central to the regridding process is the concept of "tagging" which cells need refinement. `ErrorEst` is a pure virtual function of `AmrCore`, so each application code must contain an implementation. In AmrCoreAdv.cpp the ErrorEst func-

tion is essentially an interface to a Fortran routine that tags cells (in this case, `state_error` in `Src_nd/Tagging_nd.f90`). Note that this code uses tiling.

```
// tag all cells for refinement
// overrides the pure virtual function in AmrCore
void
AmrCoreAdv::ErrorEst (int lev, TagBoxArray& tags, Real time, int ngrow)
{
    static bool first = true;
    static Vector<Real> phierr;

    // only do this during the first call to ErrorEst
    if (first)
    {
    first = false;
        // read in an array of "phierr", which is the tagging threshold
        // in this example, we tag values of "phi" which are greater than phierr
        // for that particular level
        // in subroutine state_error, you could use more elaborate tagging, such
        // as more advanced logical expressions, or gradients, etc.
    ParmParse pp("adv");
    int n = pp.countval("phierr");
    if (n > 0) {
        pp.getarr("phierr", phierr, 0, n);
    }
    }

    if (lev >= phierr.size()) return;

    const int clearval = TagBox::CLEAR;
    const int   tagval = TagBox::SET;

    const Real* dx      = geom[lev].CellSize();
    const Real* prob_lo = geom[lev].ProbLo();

    const MultiFab& state = *phi_new[lev];

#ifdef AMREX_USE_OMP
#pragma omp parallel
#endif
    {
        Vector<int>  itags;

    for (MFIter mfi(state,true); mfi.isValid(); ++mfi)
    {
        const Box& tilebox  = mfi.tilebox();

            TagBox&     tagfab  = tags[mfi];

        // We cannot pass tagfab to Fortran because it is BaseFab<char>.
        // So we are going to get a temporary integer array.
            // set itags initially to 'untagged' everywhere
            // we define itags over the tilebox region
```

(continues on next page)

(continued from previous page)

```
        tagfab.get_itags(itags, tilebox);

            // data pointer and index space
        int*        tptr    = itags.dataPtr();
        const int*  tlo     = tilebox.loVect();
        const int*  thi     = tilebox.hiVect();

            // tag cells for refinement
        state_error(tptr,  ARLIM_3D(tlo), ARLIM_3D(thi),
            BL_TO_FORTRAN_3D(state[mfi]),
            &tagval, &clearval,
            ARLIM_3D(tilebox.loVect()), ARLIM_3D(tilebox.hiVect()),
            ZFILL(dx), ZFILL(prob_lo), &time, &phierr[lev]);
        //
        // Now update the tags in the TagBox in the tilebox region
            // to be equal to itags
        //
        tagfab.tags_and_untags(itags, tilebox);
    }
    }
}
```

The `state_error` subroutine in `Src_nd/Tagging_nd.f90` in this example is simple:

```
subroutine state_error(tag,tag_lo,tag_hi, &
                       state,state_lo,state_hi, &
                       set,clear,&
                       lo,hi,&
                       dx,problo,time,phierr) bind(C, name="state_error")

  implicit none

  integer          :: lo(3),hi(3)
  integer          :: state_lo(3),state_hi(3)
  integer          :: tag_lo(3),tag_hi(3)
  double precision :: state(state_lo(1):state_hi(1), &
                            state_lo(2):state_hi(2), &
                            state_lo(3):state_hi(3))
  integer          :: tag(tag_lo(1):tag_hi(1), &
                          tag_lo(2):tag_hi(2), &
                          tag_lo(3):tag_hi(3))
  double precision :: problo(3),dx(3),time,phierr
  integer          :: set,clear

  integer          :: i, j, k

  ! Tag on regions of high phi
  do       k = lo(3), hi(3)
     do    j = lo(2), hi(2)
        do i = lo(1), hi(1)
           if (state(i,j,k) .ge. phierr) then
              tag(i,j,k) = set
```

(continues on next page)

(continued from previous page)

```
            endif
          enddo
        enddo
    enddo

end subroutine state_error
```

6.2.6 FillPatch

This example has two functions, `AmrCoreAdv::FillPatch` and `AmrCoreAdv::CoarseFillPatch`, that make use of functions in AmrCore/AMReX_FillPatchUtil.

In `AmrCoreAdv::Advance`, we create a temporary `MultiFab` called `Sborder`, which is essentially ϕ but with ghost cells filled in. The valid and ghost cells are filled in from actual valid data at that level, space-time interpolated data from the next-coarser level, neighboring grids at the same level, or domain boundary conditions (for examples that have non-periodic boundary conditions).

```
MultiFab Sborder(grids[lev], dmap[lev], S_new.nComp(), num_grow);
FillPatch(lev, time, Sborder, 0, Sborder.nComp());
```

Several other calls to fillpatch routines are hidden from the user in the regridding process.

CHAPTER

SEVEN

AMR SOURCE CODE

The source code in `amrex/Src/Amr` contains a number of classes, most notably `Amr`, `AmrLevel`, and `LevelBld`. These classes provide a more well developed set of tools for writing AMR codes than the classes created for the Advection_AmrCore tutorial.

- The `Amr` class is derived from `AmrCore`, and manages data across the entire AMR hierarchy of grids.
- The `AmrLevel` class is a pure virtual class for managing data at a single level of refinement.
- The `LevelBld` class is a pure virtual class for defining variable types and attributes.

Many of our mature, public application codes contain derived classes that inherit directly from `AmrLevel`. These include:

- The `Castro` class in our compressible astrophysics code, CASTRO, (available in the AMReX-Astro/Castro github repository)
- The `Nyx` class in our computational cosmology code, Nyx (available in the AMReX-Astro/Nyx github repository).
- Our incompressible Navier-Stokes code, IAMR (available in the AMReX-codes/IAMR github repository) has a pure virtual class called `NavierStokesBase` that inherits from `AmrLevel`, and an additional derived class `NavierStokes`.
- Our low Mach number combustion code PeleLM (available in the AMReX-Combustion/PeleLM github repository) contains a derived class `PeleLM` that also inherits from `NavierStokesBase` (but does not use `NavierStokes`).

The tutorial code in `amrex-tutorials/ExampleCodes/Amr/Advection_AmrLevel` gives a simple example of a class derived from `AmrLevel` that can be used to solve the advection equation on a subcycling-in-time AMR hierarchy. Note that example is essentially the same as the Advection AmrCore tutorial and documentation in the chapter on *AmrCore Source Code*, except now we use the provided libraries in `amrex/Src/Amr`.

The tutorial code also contains a `LevelBldAdv` class (derived from `LevelBld` in the Source/Amr directory). This class is used to define variable types (how many, nodality, interlevel interpolation stencils, etc.).

7.1 Amr Class

The `Amr` class is designed to manage parts of the computation which do not belong on a single level, like establishing and updating the hierarchy of levels, global timestepping, and managing the different `AmrLevels`. Most likely you will not need to derive any classes from `Amr`. Our mature application codes use this base class without any derived classes.

One of the most important data members is an array of `AmrLevels` - the `Amr` class calls many functions from the `AmrLevel` class to do things like advance the solution on a level, compute a time step to be used for a level, etc.

7.2 AmrLevel Class

Pure virtual functions include:

- `computeInitialDt` Compute an array of time steps for each level of refinement. Called at the beginning of the simulation.
- `computeNewDt` Compute an array of time steps for each level of refinement. Called at the end of a coarse level advance.
- `advance` Advance the grids at a level.
- `post_timestep` Work after at time step at a given level. In this tutorial we do the AMR synchronization here.
- `post_regrid` Work after regridding. In this tutorial we redistribute particles.
- `post_init` Work after initialization. In this tutorial we perform AMR synchronization.
- `initData` Initialize the data on a given level at the beginning of the simulation.
- `init` There are two versions of this function used to initialize data on a level during regridding. One version is specifically for the case where the level did not previously exist (a newly created refined level).
- `errorEst` Perform the tagging at a level for refinement.

7.2.1 StateData

The most important data managed by the `AmrLevel` is an array of `StateData`, which holds the scalar fields, etc., in the boxes that together make up the level.

`StateData` is a class that essentially holds a pair of MultiFabs: one at the old time and one at the new time. AMReX knows how to interpolate in time between these states to get data at any intermediate point in time. The main data that we care about in our applications codes (such as the fluid state) will be stored as `StateData`. Essentially, data is made `StateData` if we need it to be stored in checkpoints/plotfiles, and/or we want it to be automatically interpolated when we refine. An `AmrLevel` stores an array of `StateData` (in a C++ array called `state`). We index this array using integer keys (defined via an `enum` in, e.g., `AmrLevelAdv.H`):

```
enum StateType { Phi_Type = 0,
                 NUM_STATE_TYPE };
```

In our tutorial code, we use the function `AmrLevelAdv::variableSetUp` to tell our simulation about the `StateData` (e.g., how many variables, ghost cells, nodality, etc.). Note that if you have more than one `StateType`, each of the different `StateData` carried in the state array can have different numbers of components, ghost cells, boundary conditions, etc. This is the main reason we separate all this data into separate `StateData` objects collected together in an indexable array.

7.3 LevelBld Class

The `LevelBld` class is a pure virtual class for defining variable types and attributes. To more easily understand its usage, refer to the derived class, `LevelBldAdv` in the tutorial. The `variableSetUp` and `variableCleanUp` are implemented, and in this tutorial call routines in the `AmrLevelAdv` class, e.g.,

```
void
AmrLevelAdv::variableSetUp ()
{
```

(continues on next page)

(continued from previous page)

```
    BL_ASSERT(desc_lst.size() == 0);

    // Get options, set phys_bc
    read_params();

    desc_lst.addDescriptor(Phi_Type,IndexType::TheCellType(),
                           StateDescriptor::Point,0,NUM_STATE,
                           &cell_cons_interp);

    int lo_bc[BL_SPACEDIM];
    int hi_bc[BL_SPACEDIM];
    for (int i = 0; i < BL_SPACEDIM; ++i) {
        lo_bc[i] = hi_bc[i] = INT_DIR;   // periodic boundaries
    }

    BCRec bc(lo_bc, hi_bc);

    StateDescriptor::BndryFunc bndryfunc(nullfill);
    bndryfunc.setRunOnGPU(true);  // I promise the bc function will launch gpu kernels.

    desc_lst.setComponent(Phi_Type, 0, "phi", bc,
                          bndryfunc);
}
```

We see how to define the `StateType`, including nodality, whether or not we want the variable to represent a point in time or an interval over time (useful for returning the time associated with data), the number of ghost cells, number of components, and the interlevel interpolation (See AMReX_Interpolator for various interpolation types). We also see how to specify physical boundary functions by providing a function (in this case, `nullfill` since we are not using physical boundary conditions), where `nullfill` is defined in `Src/bc_nullfill.cpp` in the tutorial source code.

7.4 Example: Advection_AmrLevel

The Advection_AmrLevel example is documented in detail here in the AMReX tutorial documentation.

The `Src` subdirectory contains source code that is specific to this example. Most notably is the `AmrLevelAdv` class, which is derived from the base `AmrLevel` class, and the `LevelBldAdv` class, derived from the base `LevelBld` class as described above. The subdirectory `Src/Src_K` contain GPU kernels.

The `Exec` subdirectory contains two examples: `SingleVortex` and `UniformVelocity`. Each subdirectory contains problem-specific source code used for initialization using a Fortran subroutine (`Prob.f90`) and specifying the velocity fields used in this simulation (`face_velocity_2d_K.H` and `face_velocity_3d_K.H` for the 2-D and 3-D problem, respectively). Build the code here by editing the `GNUmakefile` and running `make`.

The pseudocode for the main program is given below.

```
/* Advection_AmrLevel Pseudocode */
main()
  Amr amr;
  amr.init()
  loop {
    amr.coarseTimeStep()
      /* compute dt */
```

(continues on next page)

(continued from previous page)

```
      timeStep()
        amr_level[level]->advance()
        /* call timeStep r times for next-finer level */
        amr_level[level]->post_timestep() // AMR synchronization
      postCoarseTimeStep()
      /* write plotfile and checkpoint */
  }
  /* write final plotfile and checkpoint */
```

7.5 Particles

There is an option to turn on passively advected particles. In the `GNUmakefile`, add the line `USE_PARTICLES = TRUE` and build the code (do a `make realclean first`). In the inputs file, add the line `adv.do_tracers = 1`. When you run the code, within each plotfile directory there will be a subdirectory called "Tracer".

Copy the files from `amrex/Tools/Py_util/amrex_particles_to_vtp` into the run directory and type, e.g.,

```
python amrex_binary_particles_to_vtp.py plt00000 Tracer
```

To generate a vtp file you can open with ParaView (Refer to the chapter on *Visualization*).

CHAPTER

EIGHT

FORK-JOIN

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 AMReX Fork-Join 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.

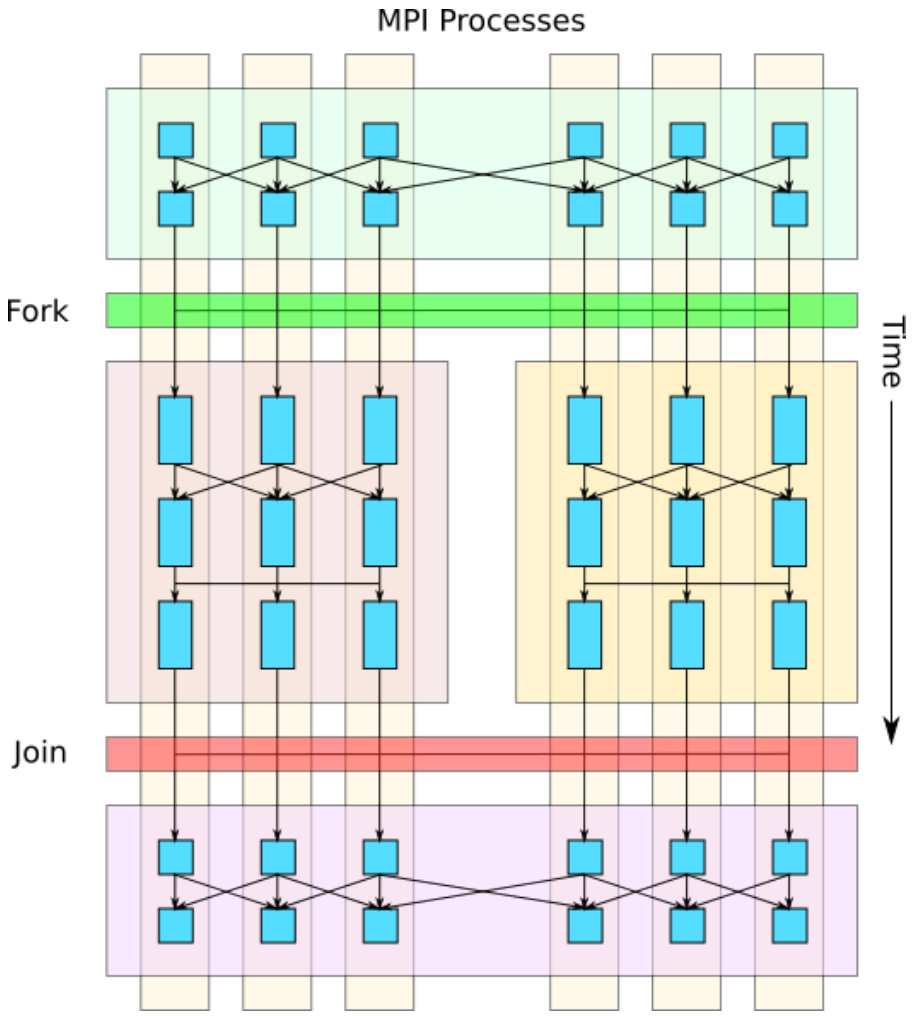


Fig. 8.1: Example of a fork-join operation where the parent task's MPI processes (ranks) are split into two independent child tasks that execute in parallel and then join to resume execution of the parent task.

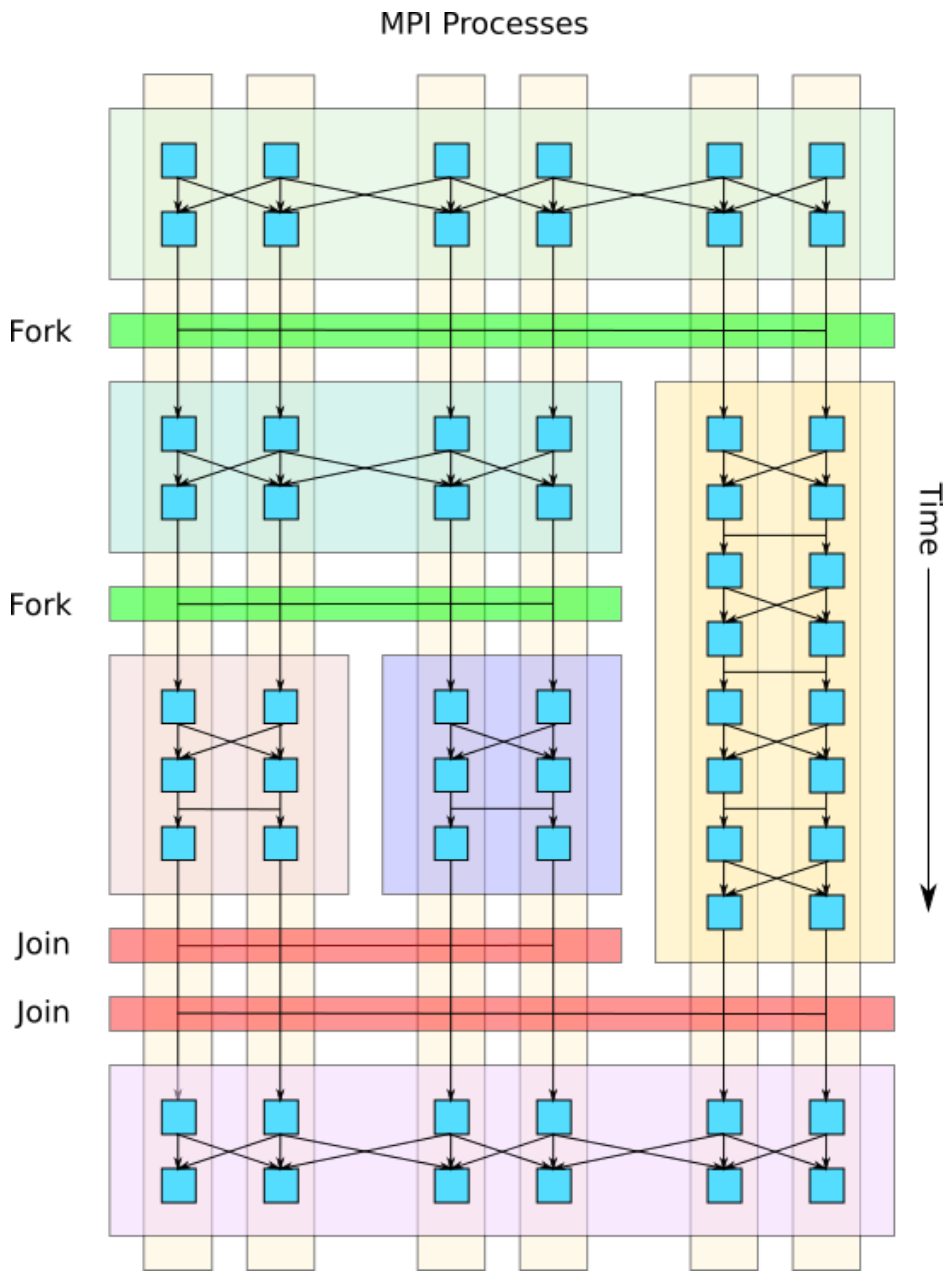


Fig. 8.2: Example of nested fork-join operations where a child task is further split into more subtasks.

CHAPTER

NINE

I/O (PLOTFILE, CHECKPOINT)

In this chapter, we will discuss parallel I/O capabilities for mesh data in AMReX. The section on *Particle IO* will discuss I/O for particle data.

9.1 Plotfile

AMReX has its own native plotfile format. Many visualization tools are available for AMReX plotfiles (see the chapter on *Visualization*). AMReX provides the following two functions for writing a generic AMReX plotfile. Many AMReX application codes may have their own plotfile routines that store additional information such as compiler options, git hashes of the source codes and `ParmParse` runtime parameters.

```
void WriteSingleLevelPlotfile (const std::string &plotfilename,
                               const MultiFab &mf,
                               const Vector<std::string> &varnames,
                               const Geometry &geom,
                               Real time,
                               int level_step);

void WriteMultiLevelPlotfile (const std::string &plotfilename,
                              int nlevels,
                              const Vector<const MultiFab*> &mf,
                              const Vector<std::string> &varnames,
                              const Vector<Geometry> &geom,
                              Real time,
                              const Vector<int> &level_steps,
                              const Vector<IntVect> &ref_ratio);
```

`WriteSingleLevelPlotfile` is for single level runs and `WriteMultiLevelPlotfile` is for multiple levels. The name of the plotfile is specified by the plotfilename argument. This is the top level directory name for the plotfile. In AMReX convention, the plotfile name consist of letters followed by numbers (e.g., plt00258). `amrex::Concatenate` is a useful helper function for making such strings.

```
int istep = 258;
const std::string& pfname = amrex::Concatenate("plt",istep); // plt00258

// By default there are 5 digits, but we can change it to say 4.
const std::string& pfname2 = amrex::Concatenate("plt",istep,4); // plt0258

istep =1234567;  // Having more than 5 digits is OK.
const std::string& pfname3 = amrex::Concatenate("plt",istep); // plt1234567
```

The argument `mf` above (`MultiFab` for single level and `Vector<const MultiFab*>` for multi-level) is the data to be written to the disk. Note that many visualization tools expect this to be cell-centered data. So for nodal data, we need to convert them to cell-centered data through some kind of averaging. Also note that if you have data at each AMR level in several MultiFabs, you need to build a new MultiFab at each level to hold all the data on that level. This involves local data copy in memory and is not expected to significantly increase the total wall time for writing plotfiles. For the multi-level version, the function expects `Vector<const MultiFab*>`, whereas the multi-level data are often stored as `Vector<std::unique_ptr<MultiFab>>`. AMReX has a helper function for this and one can use it as follows,

```
WriteMultiLevelPlotfile(......, amrex::GetVecOfConstPtrs(mf), ......);
```

The argument `varnames` has the names for each component of the MultiFab data. The size of the Vector should be equal to the number of components. The argument `geom` is for passing `Geometry` objects that contain the physical domain information. The argument `time` is for the time associated with the data. The argument `level_step` is for the current time step associated with the data. For multi-level plotfiles, the argument `nlevels` is the total number of levels, and we also need to provide the refinement ratio via an `Vector` of size nlevels-1.

We note that AMReX does not overwrite old plotfiles if the new plotfile has the same name. The old plotfiles will be renamed to new directories named like plt00350.old.46576787980.

9.2 Async Output

AMReX provides the ability to print MultiFabs, plotfiles and particle data asynchronously. Asynchronous output works by creating a copy of the data at the time of the call, which is written to disk by a persistent thread created during AMReX's initialization. This allows the calculation to continue immediately, which can drastically reduce walltime spent writing to disk.

If the number of output files is less than the number of MPI ranks, AMReX's async output requires MPI to be initialized with THREAD_MULTIPLE support. THREAD_MULTIPLE support allows multiple unique threads to run unique MPI calls simultaneously. This support is required to allow AMReX applications to perform MPI work while the Async Output concurrently pings ranks to signal that they can safely begin writing to their assigned files. However, THREAD_MULTIPLE can introduce additional overhead as each threads' MPI operations must be scheduled safely around each other. Therefore, AMReX uses a lower level of support, SERIALIZED, by default and applications have to turn on THREAD_MULTIPLE support.

To turn on Async Output, use the input flag `amrex.async_out=1`. The number of output files can also be set, using `amrex.async_out_nfiles`. The default number of files is `64`. If the number of ranks is larger than the number of files, THREAD_MULTIPLE must be turned on by adding `MPI_THREAD_MULTIPLE=TRUE` to the GNUMakefile. Otherwise, AMReX will throw an error.

Async Output works for a wide range of AMReX calls, including:

- `amrex::WriteSingleLevelPlotfile()`
- `amrex::WriteMultiLevelPlotfile()`
- `amrex::WriteMLMF()`
- `VisMF::AsyncWrite()`
- `ParticleContainer::Checkpoint()`
- `ParticleContainer::WritePlotFile()`
- `Amr::writePlotFile()`
- `Amr::writeSmallPlotFile()`
- `Amr::checkpoint()`

- `AmrLevel::writePlotFile()`
- `StateData::checkPoint()`
- `FabSet::write()`

Be aware: when using Async Output, a thread is spawned and exclusively used to perform output throughout the runtime. As such, you may oversubscribe resources if you launch an AMReX application that assigns all available hardware threads in another way, such as OpenMP. If you see any degradation when using Async Output and OpenMP, try using one less thread in `OMP_NUM_THREADS` to prevent oversubscription and get more consistent results.

9.3 HDF5 Plotfile

Besides AMReX's native plotfile, applications can also write plotfile in the HDF5 format, which is a cross-platform, self-describing file format. The HDF5 plotfiles store the same information as the native format, and has the additional compression capability that can reduce the file size. Currently supported compression libraries include SZ and ZFP.

To enable HDF5 output, AMReX must be compiled and linked to an HDF5 library with parallel I/O support, by adding `USE_HDF5=TRUE` and `HDF5_HOME=/path/to/hdf5/install/dir` to the GNUMakefile. many HPC systems have an HDF5 module available that can be loaded with `module load hdf5` or `module load cray-hdf5-parallel`. To download and compile HDF5 from source code, please go to HDF5 Download webpage and follow the instructions (latest version is recommended and remember to turn on parallel I/O).

Following are two functions for writing a generic AMReX plotfile in HDF5 format, which are very similar to the AMReX native write functions.

```
void WriteSingleLevelPlotfileHDF5 (const std::string &plotfilename,
                                   const MultiFab &mf,
                                   const Vector<std::string> &varnames,
                                   const Geometry &geom,
                                   Real t,
                                   int level_step,
                                   const std::string &compression);

void WriteMultiLevelPlotfileHDF5 (const std::string &plotfilename,
                                  int nlevels,
                                  const Vector<const MultiFab*> &mf,
                                  const Vector<std::string> &varnames,
                                  const Vector<Geometry> &geom,
                                  Real time,
                                  const Vector<int> &level_steps,
                                  const Vector<IntVect> &ref_ratio,
                                  const std::string &compression);
```

`WriteSingleLevelPlotfileHDF5` is for single level runs and `WriteMultiLevelPlotfileHDF5` is for multiple levels. Their arguments are the same as the native ones except the last one, which optional, and specifies the compression parameters. These two functions write plotfiles with a Chombo-compatible HDF5 file schema, which can be read by visualization tools such as VisIt and ParaView using their built-in Chombo reader plugin (see the chapter on *Visualization*)

9.3.1 HDF5 Plotfile Compression

To enable data compression on the HDF5 datasets, the corresponding compression library and its HDF5 plugin must be available. To compile SZ or ZFP plugin, please refer to their documentation: H5Z-SZ and H5Z-ZFP, and adding `USE_HDF5_SZ=TRUE`, `SZ_HOME=`, or `USE_HDF5_ZFP=TRUE`, `ZFP_HOME=`, `H5Z_HOME=` to the GNUMakefile.

The string argument `compression` in the above two functions controls whether to enable data compression and its parameters. Currently supported options include:

- **No compression**
  - `None@0`
- **SZ compression**
  - `SZ@/path/to/sz.config`
- **ZFP compression**
  - `ZFP_RATE@rate`
  - `ZFP_PRECISION@precision`
  - `ZFP_ACCURACY@accuracy`
  - `ZFP_REVERSIBLE@reversible`

9.3.2 HDF5 Asynchronous Output

The HDF5 output also comes with its own asynchronous I/O support, which is different from the native async output mentioned in the previous section. To use the HDF5 asynchronous I/O VOL connector, download and compile by following the instructions at vol-async.

Since the HDF5 asynchronous I/O in AMReX does not use double buffering, vol-async must be compiled with `-DENABLE_WRITE_MEMCPY=1` added to `CFLAGS`. When compiling AMReX, add `USE_HDF5_ASYNC = TRUE`, `ABT_HOME=`, `ASYNC_HOME=`, and `MPI_THREAD_MULTIPLE=TRUE` to the GNUMakefile. Refer to `amrex/Tests/HDF5Benchmark/GNUmakefile` for the example usage.

9.3.3 Alternative HDF5 Plotfile Schema

`WriteSingleLevelPlotfileHDF5` and `WriteMultiLevelPlotfileHDF5` write HDF5 plotfiles that store all the data on an AMR level as one 1D HDF5 dataset. Each AMR box's data is linearized and the data of different variables are concatenated, resulting in an interleaved pattern for each variable. This could be undesirable when compression is used, as it may lead to applying the compression algorithm to multiple variables with different value ranges and chararistics, and reduce the compression ratio. To overcome this issue, two additional functions are provided to write each variable into individual HDF5 datasets: `WriteSingleLevelPlotfileHDF5MultiDset` and `WriteMultiLevelPlotfileHDF5MultiDset`. They use the exact same arguments as `WriteSingleLevelPlotfileHDF5` and `WriteMultiLevelPlotfileHDF5`. However, this alternative schema is not yet supported by the visualization tools.

9.4 Checkpoint File

Checkpoint files are used for restarting simulations from where the checkpoints are written. Each application code has its own set of data needed for restart. AMReX provides I/O functions for basic data structures like `MultiFab` and `BoxArray`. These functions can be used to build codes for reading and writing checkpoint files. Since each application code has its own requirement, there is no standard AMReX checkpoint format. However we have provided an example restart capability in the tutorial Advection AmrCore. Refer to the functions `ReadCheckpointFile()` and `WriteCheckpointFile()` in this tutorial.

A checkpoint file is actually a directory with name, e.g., `chk00010` containing a `Header` (text) file, along with subdirectories `Level_0`, `Level_1`, etc. containing the `MultiFab` data at each level of refinement. The `Header` file contains problem-specific data (such as the finest level, simulation time, time step, etc.), along with a printout of the `BoxArray` at each level of refinement.

When starting a simulation from a checkpoint file, a typical sequence in the code could be:

- Read in the `Header` file data (except for the `BoxArray` data).
- For each level of refinement, do the following in order:
  - Read in the `BoxArray`
  - Build a `DistributionMapping`
  - Define any `MultiFab`, `FluxRegister`, etc. objects that are built upon the `BoxArray` and the `DistributionMapping`
  - Read in the `MultiFab` data

We do this one level at a time because when you create a distribution map, it checks how much allocated `MultiFab` data already exists before assigning grids to processors.

Typically a checkpoint file is a directory containing some text files and sub-directories (e.g., `Level_0` and `Level_1`) containing various data. It is a good idea that we fist make these directories ready for subsequently writing to the disk. For example, to build directories `chk00010`, `chk00010/Level_0`, and `chk00010/Level_1`, you could write:

```
const std::string& checkpointname = amrex::Concatenate("chk",10);

amrex::Print() << "Writing checkpoint " << checkpointname << "\n";

const int nlevels = 2;

bool callBarrier = true;

// ---- prebuild a hierarchy of directories
// ---- dirName is built first.  if dirName exists, it is renamed.  then build
// ---- dirName/subDirPrefix_0 .. dirName/subDirPrefix_nlevels-1
// ---- if callBarrier is true, call ParallelDescriptor::Barrier()
// ---- after all directories are built
// ---- ParallelDescriptor::IOProcessor() creates the directories
amrex::PreBuildDirectorHierarchy(checkpointname, "Level_", nlevels, callBarrier);
```

A checkpoint file of AMReX application codes often has a clear text Header file that only the I/O process writes to it using `std::ofstream`. The Header file contains problem-dependent information such as the time, the physical domain size, grids, etc. that are necessary for restarting the simulation. To guarantee that precision is not lost for storing floating point number like time in clear text file, the file stream's precision needs to be set properly. And a stream buffer can also be used. For example,

```
// write Header file
if (ParallelDescriptor::IOProcessor()) {

    VisMF::IO_Buffer io_buffer(VisMF::IO_Buffer_Size);
    std::ofstream HeaderFile;
    HeaderFile.rdbuf()->pubsetbuf(io_buffer.dataPtr(), io_buffer.size());
    std::string HeaderFileName(checkpointname + "/Header");
    HeaderFile.open(HeaderFileName.c_str(), std::ofstream::out   |
                                            std::ofstream::trunc |
                                            std::ofstream::binary);

    if( ! HeaderFile.good()) {
        amrex::FileOpenFailed(HeaderFileName);
    }

    HeaderFile.precision(17);

    // write out title line
    HeaderFile << "Checkpoint file for AmrCoreAdv\n";

    // write out finest_level
    HeaderFile << finest_level << "\n";

    // write out array of istep
    for (int i = 0; i < istep.size(); ++i) {
        HeaderFile << istep[i] << " ";
    }
    HeaderFile << "\n";

    // write out array of dt
    for (int i = 0; i < dt.size(); ++i) {
        HeaderFile << dt[i] << " ";
    }
    HeaderFile << "\n";

    // write out array of t_new
    for (int i = 0; i < t_new.size(); ++i) {
        HeaderFile << t_new[i] << " ";
    }
    HeaderFile << "\n";

    // write the BoxArray at each level
    for (int lev = 0; lev <= finest_level; ++lev) {
        boxArray(lev).writeOn(HeaderFile);
        HeaderFile << '\n';
    }
}
```

`amrex::VisMF` is a class that can be used to perform `MultiFab` I/O in parallel. How many processes are allowed to perform I/O simultaneously can be set via

```
VisMF::SetNOutFiles(64);  // up to 64 processes, which is also the default.
```

The optimal number is of course system dependent. The following code shows how to write a `MultiFab`.

```
// write the MultiFab data to, e.g., chk00010/Level_0/
for (int lev = 0; lev <= finest_level; ++lev) {
    VisMF::Write(phi_new[lev],
                 amrex::MultiFabFileFullPrefix(lev, checkpointname, "Level_", "phi"));
}
```

It should also be noted that all the data including those in ghost cells are written/read by `VisMF::Write/Read`.

For reading the Header file, AMReX can have the I/O process read the file from the disk and broadcast it to others as `Vector<char>`. Then all processes can read the information with `std::istringstream`. For example,

```
std::string File(restart_chkfile + "/Header");

VisMF::IO_Buffer io_buffer(VisMF::GetIOBufferSize());

Vector<char> fileCharPtr;
ParallelDescriptor::ReadAndBcastFile(File, fileCharPtr);
std::string fileCharPtrString(fileCharPtr.dataPtr());
std::istringstream is(fileCharPtrString, std::istringstream::in);

std::string line, word;

// read in title line
std::getline(is, line);

// read in finest_level
is >> finest_level;
GotoNextLine(is);

// read in array of istep
std::getline(is, line);
{
    std::istringstream lis(line);
    int i = 0;
    while (lis >> word) {
        istep[i++] = std::stoi(word);
    }
}

// read in array of dt
std::getline(is, line);
{
    std::istringstream lis(line);
    int i = 0;
    while (lis >> word) {
        dt[i++] = std::stod(word);
    }
}

// read in array of t_new
std::getline(is, line);
{
    std::istringstream lis(line);
```

(continues on next page)

(continued from previous page)

```
    int i = 0;
    while (lis >> word) {
        t_new[i++] = std::stod(word);
    }
}
```

The following code how to read in a `BoxArray`, create a `DistributionMapping`, build `MultiFab` and `FluxRegister` data, and read in a `MultiFab` from a checkpoint file, on a level-by-level basis:

```
for (int lev = 0; lev <= finest_level; ++lev) {

    // read in level 'lev' BoxArray from Header
    BoxArray ba;
    ba.readFrom(is);
    GotoNextLine(is);

    // create a distribution mapping
    DistributionMapping dm { ba, ParallelDescriptor::NProcs() };

    // set BoxArray grids and DistributionMapping dmap in AMReX_AmrMesh.H class
    SetBoxArray(lev, ba);
    SetDistributionMap(lev, dm);

    // build MultiFab and FluxRegister data
    int ncomp = 1;
    int nghost = 0;
    phi_old[lev].define(grids[lev], dmap[lev], ncomp, nghost);
    phi_new[lev].define(grids[lev], dmap[lev], ncomp, nghost);
    if (lev > 0 && do_reflux) {
        flux_reg[lev] = std::make_unique<FluxRegister>(grids[lev], dmap[lev],␣
↪refRatio(lev-1), lev, ncomp);
    }
}

// read in the MultiFab data
for (int lev = 0; lev <= finest_level; ++lev) {
    VisMF::Read(phi_new[lev],
                amrex::MultiFabFileFullPrefix(lev, restart_chkfile, "Level_", "phi"));
}
```

It should be emphasized that calling `VisMF::Read` with an empty `MultiFab` (i.e., no memory allocated for floating point data) will result in a `MultiFab` with a new `DistributionMapping` that could be different from any other existing `DistributionMapping` objects and is not recommended.

CHAPTER

TEN

LINEAR SOLVERS

AMReX supports both single-level solves and composite solves on multiple AMR levels, with the scalar solution to the linear system defined on either cell centers or nodes. AMReX also supports solution of linear systems with embedded boundaries. (See chapter *Embedded Boundaries* for more details on the embedded boundary representation of complex geometry.)

The default solution technique is geometric multigrid, but AMReX includes native BiCGStab solvers for a single level as well as interfaces to the hypre library.

In this Chapter we give an overview of the linear solvers in AMReX that solve linear systems in the canonical form

$$(A\alpha - B\nabla \cdot \beta\nabla)\phi = f, \tag{10.1}$$

where A and B are scalar constants, α and β are scalar fields, ϕ is the unknown, and f is the right-hand side of the equation. Note that Poisson's equation $\nabla^2\phi = f$ is a special case of the canonical form. The solution ϕ is at either cell centers or nodes.

For the cell-centered solver, α, ϕ and f are represented by cell-centered MultiFabs, and β is represented by `AMREX_SPACEDIM` face type MultiFabs, i.e. there are separate MultiFabs for the β coefficient in each coordinate direction.

For the nodal solver, A and α are assumed to be zero, ϕ and f are nodal, and β (which we later refer to as σ) is cell-centered.

In addition to these solvers, AMReX has support for tensor solves used to calculate the viscous terms that appear in the compressible Navier-Stokes equations. In these solves, all components of the velocity field are solved for simultaneously. The tensor solve functionality is only available for cell-centered velocity.

The tutorials in Linear Solvers show examples of using the solvers. The tutorial `amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX3_C` shows how to solve the heat equation implicitly using the solver. The tutorials also show how to add linear solvers into the build system.

10.1 MLMG and Linear Operator Classes

Multi-Level Multi-Grid or `MLMG` is a class for solving the linear system using the geometric multigrid method. The constructor of `MLMG` takes the reference to `MLLinOp`, an abstract base class of various linear operator classes, `MLABecLaplacian`, `MLPoisson`, `MLNodeLaplacian`, etc. We choose the type of linear operator class according to the type the linear system to solve.

- `MLABecLaplacian` for cell-centered canonical form (equation (10.1)).
- `MLPoisson` for cell-centered constant coefficient Poisson's equation $\nabla^2\phi = f$.
- `MLNodeLaplacian` for nodal variable coefficient Poisson's equation $\nabla \cdot (\sigma\nabla\phi) = f$.

The constructors of these linear operator classes are in the form like below

```
MLABecLaplacian (const Vector<Geometry>& a_geom,
                 const Vector<BoxArray>& a_grids,
                 const Vector<DistributionMapping>& a_dmap,
                 const LPInfo& a_info = LPInfo(),
                 const Vector<FabFactory<FArrayBox> const*>& a_factory = {},
                 const int a_ncomp = 1);
```

It takes `Vectors` of `Geometry`, `BoxArray` and `DistributionMapping`. The arguments are `Vectors` because MLMG can do multi-level composite solves. If you are using it for single-level, you can do

```
// Given Geometry geom, BoxArray grids, and DistributionMapping dmap on single level
MLABecLaplacian mlabeclaplacian({geom}, {grids}, {dmap});
```

to let the compiler construct `Vectors` for you. Recall that the classes `Vector`, `Geometry`, `BoxArray`, and `DistributionMapping` are defined in chapter *Basics*. There are two new classes that are optional parameters. `LPInfo` is a class for passing parameters. `FabFactory` is used in problems with embedded boundaries (chapter *Embedded Boundaries*).

After the linear operator is built, we need to set up boundary conditions. This will be discussed later in section *Boundary Conditions*.

10.1.1 Coefficients

Next, we consider the coefficients for equation (10.1). For `MLPoisson`, there are no coefficients to set so nothing needs to be done. For `MLABecLaplacian`, we need to call member functions `setScalars`, `setACoeffs`, and `setBCoeffs`. The `setScalars` function sets the scalar constants A and B

```
void setScalars (Real a, Real b) noexcept;
```

For the general case where α and β are scalar fields, we use

```
void setACoeffs (int amrlev, const MultiFab& alpha);
void setBCoeffs (int amrlev, const Array<MultiFab const*,AMREX_SPACEDIM>&␣
↪beta);
```

For the case where α and/or β are scalar constants, there is the option to use

```
void setACoeffs (int amrlev, Real alpha);
void setBCoeffs (int amrlev, Real beta);
void setBCoeffs (int amrlev, Vector<Real> const& beta);
```

Note, however, that the solver behaviour is the same regardless of which functions you use to set the coefficients. These functions solely copy the constant value(s) to a MultiFab internal to `MLMG` and so no appreciable efficiency gains can be expected.

For `MLNodeLaplacian`, one can set a variable `sigma` with the member function

```
void setSigma (int amrlev, const MultiFab& a_sigma);
```

or a constant `sigma` during declaration or definition

```
MLNodeLaplacian (const Vector<Geometry>& a_geom,
                 const Vector<BoxArray>& a_grids,
                 const Vector<DistributionMapping>& a_dmap,
```

(continues on next page)

(continued from previous page)

```
                const LPInfo& a_info = LPInfo(),
                const Vector<FabFactory<FArrayBox> const*>& a_factory = {},
                Real  a_const_sigma = Real(0.0));

void define (const Vector<Geometry>& a_geom,
             const Vector<BoxArray>& a_grids,
             const Vector<DistributionMapping>& a_dmap,
             const LPInfo& a_info = LPInfo(),
             const Vector<FabFactory<FArrayBox> const*>& a_factory = {},
             Real  a_const_sigma = Real(0.0));
```

Here, setting a constant `sigma` alters the internal behavior of the solver making it more efficient for this special case.

The `int amrlev` parameter should be zero for single-level solves. For multi-level solves, each level needs to be provided with `alpha` and `beta`, or `sigma`. For composite solves, `amrlev` 0 will mean the lowest level for the solver, which is not necessarily the lowest level in the AMR hierarchy. This is so solves can be done on different sections of the AMR hierarchy, e.g. on AMR levels 3 to 5.

After boundary conditions and coefficients are prescribed, the linear operator is ready for an MLMG object like below.

```
MLMG mlmg(mlabeclaplacian);
```

Optional parameters can be set (see section *Parameters*), and then we can use the `MLMG` member function

```
Real solve (const Vector<MultiFab*>& a_sol,
            const Vector<MultiFab const*>& a_rhs,
            Real a_tol_rel, Real a_tol_abs);
```

to solve the problem given an initial guess and a right-hand side. Zero is a perfectly fine initial guess. The two `Reals` in the argument list are the targeted relative and absolute error tolerances. The relative error tolerance is hard-coded to be at least 10^{-16}. Given the linear system $Ax = b$, the solver will terminate when the max-norm of the residual $(b - Ax)$ is less than `std::max(a_tol_abs, a_tol_rel*max_norm)` where `max_norm` is the max-norm of the rhs, b, if the flag `always_use_bnorm` is set to True or if the rhs max-norm is greater than or equal to the max-norm error of the initial guess, otherwise `max_norm` is equal to the max-norm error of the initial guess. Set the absolute tolerance to zero if one does not have a good value for it. The return value of `solve` is the max-norm error.

After the solver returns successfully, if needed, we can call

```
void compResidual (const Vector<MultiFab*>& a_res,
                   const Vector<MultiFab*>& a_sol,
                   const Vector<MultiFab const*>& a_rhs);
```

to compute residual (i.e., $f - L(\phi)$) given the solution and the right-hand side. For cell-centered solvers, we can also call the following functions to compute gradient $\nabla\phi$ and fluxes $-\beta\nabla\phi$.

```
void getGradSolution (const Vector<Array<MultiFab*,AMREX_SPACEDIM> >& a_grad_sol);
void getFluxes       (const Vector<Array<MultiFab*,AMREX_SPACEDIM> >& a_fluxes);
```

10.2 Boundary Conditions

We now discuss how to set up boundary conditions for linear operators. In the following, physical domain boundaries refer to the boundaries of the physical domain, whereas coarse/fine boundaries refer to the boundaries between AMR levels. The following steps must be followed in the exact order.

1) For any type of solver, we first need to set physical domain boundary types via the `MLLinOp` member function

```
void setDomainBC (const Array<LinOpBCType,AMREX_SPACEDIM>& lobc,  // for lower ends
                  const Array<LinOpBCType,AMREX_SPACEDIM>& hibc); // for higher ends
```

The supported BC types at the physical domain boundaries are

- `LinOpBCType::Periodic` for periodic boundary.
- `LinOpBCType::Dirichlet` for Dirichlet boundary condition.
- `LinOpBCType::Neumann` for homogeneous Neumann boundary condition.
- `LinOpBCType::inhomogNeumann` for inhomogeneous Neumann boundary condition.
- `LinOpBCType::Robin` for Robin boundary conditions, $a\phi + b\frac{\partial \phi}{\partial n} = f$.
- `LinOpBCType::reflect_odd` for reflection with sign changed.

2) Cell-centered solvers only: if we want to do a linear solve where the boundary conditions on the coarsest AMR level of the solve come from a coarser level (e.g. the base AMR level of the solve is > 0 and does not cover the entire domain), we must explicitly provide the coarser data. Boundary conditions from a coarser level are always Dirichlet.

Note that this step, if needed, must be performed before the step below. The `MLLinOp` member function for this step is

```
void setCoarseFineBC (const MultiFab* crse, int crse_ratio);
```

Here `const MultiFab* crse` contains the Dirichlet boundary values at the coarse resolution, and `int crse_ratio` (e.g., 2) is the refinement ratio between the coarsest solver level and the AMR level below it. The MultiFab crse does not need to have ghost cells itself. If the coarse grid bc's for the solve are identically zero, `nullptr` can be passed instead of `crse`.

3) Cell-centered solvers only: before the solve one must always call the `MLLinOp` member function

```
virtual void setLevelBC (int amrlev, const MultiFab* levelbcdata,
                         const MultiFab* robinbc_a = nullptr,
                         const MultiFab* robinbc_b = nullptr,
                         const MultiFab* robinbc_f = nullptr) = 0;
```

If we want to supply an inhomogeneous Dirichlet or inhomogeneous Neumann boundary condition at the domain boundaries, we must supply those values in `MultiFab* levelbcdata`, which must have at least one ghost cell. Note that the argument `amrlev` is relative to the solve, not necessarily the full AMR hierarchy; amrlev = 0 refers to the coarsest level of the solve.

If the boundary condition is Dirichlet the ghost cells outside the domain boundary of `levelbcdata` must hold the value of the solution at the domain boundary; if the boundary condition is Neumann those ghost cells must hold the value of the gradient of the solution normal to the boundary (e.g. it would hold dphi/dx on both the low and high faces in the x-direction).

If the boundary conditions contain no inhomogeneous Dirichlet or Neumann boundaries, we can pass `nullptr` instead of a MultiFab.

We can use the solution array itself to hold these values; the values are copied to internal arrays and will not be overwritten when the solution array itself is being updated by the solver. Note, however, that this call does not provide an initial guess for the solve.

It should be emphasized that the data in `levelbcdata` for Dirichlet or Neumann boundaries are assumed to be exactly on the face of the physical domain; storing these values in the ghost cell of a cell-centered array is a convenience of implementation.

For Robin boundary conditions, the ghost cells in `MultiFab* robinbc_a`, `MultiFab* robinbc_b`, and `MultiFab* robinbc_f` store the numerical values in the condition, $a\phi + b\frac{\partial\phi}{\partial n} = f$.

10.3 Parameters

There are many parameters that can be set. Here we discuss some commonly used ones.

`MLLinOp::setVerbose(int)`, `MLMG::setVerbose(int)` and `MLMG:setBottomVerbose(int)` control the verbosity of the linear operator, multigrid solver and the bottom solver, respectively.

The multigrid solver is an iterative solver. The maximal number of iterations can be changed with `MLMG::setMaxIter(int)`. We can also do a fixed number of iterations with `MLMG::setFixedIter(int)`. By default, V-cycle is used. We can use `MLMG::setMaxFmgIter(int)` to control how many full multigrid cycles can be done before switching to V-cycle.

`LPInfo::setMaxCoarseningLevel(int)` can be used to control the maximal number of multigrid levels. We usually should not call this function. However, we sometimes build the solver to simply apply the operator (e.g., $L(\phi)$) without needing to solve the system. We can do something as follows to avoid the cost of building coarsened operators for the multigrid.

```
MLABecLaplacian mlabeclap({geom}, {grids}, {dmap}, LPInfo().setMaxCoarseningLevel(0));
// set up BC
// set up coefficients
MLMG mlmg(mlabeclap);
// out = L(in)
mlmg.apply(out, in);  // here both in and out are const Vector<MultiFab*>&
```

At the bottom of the multigrid cycles, we use a `bottom solver` which may be different than the relaxation used at the other levels. The default bottom solver is the biconjugate gradient stabilized method, but can easily be changed with the `MLMG` member method

```
void setBottomSolver (BottomSolver s);
```

Available choices are

- `MLMG::BottomSolver::bicgstab`: The default.
- `MLMG::BottomSolver::cg`: The conjugate gradient method. The matrix must be symmetric.
- `MLMG::BottomSolver::smoother`: Smoother such as Gauss-Seidel.
- `MLMG::BottomSolver::bicgcg`: Start with bicgstab. Switch to cg if bicgstab fails. The matrix must be symmetric.
- `MLMG::BottomSolver::cgbicg`: Start with cg. Switch to bicgstab if cg fails. The matrix must be symmetric.
- `MLMG::BottomSolver::hypre`: One of the solvers available through hypre; see the section below on External Solvers
- `MLMG::BottomSolver::petsc`: Currently for cell-centered only.
- `LPInfo::setAgglomeration(bool)` (by default true) can be used continue to coarsen the multigrid by copying what would have been the bottom solver to a new `MultiFab` with a new `BoxArray` with fewer, larger grids, to allow for additional coarsening.

- LPInfo::setConsolidation(bool) (by default true) can be used continue to transfer a multigrid problem to fewer MPI ranks. There are more setting sucsh as LPInfo::setConsolidationGridSize(int), LPInfo::setConsolidationRatio(int), and LPInfo::setConsolidationStrategy(int), to give control over how this process works.

10.4 Boundary Stencils for Cell-Centered Solvers

We have the option using the MLMG member method

```
void setMaxOrder (int maxorder);
```

to set the order of the cell-centered linear operator stencil at physical boundaries with Dirichlet boundary conditions and at coarse-fine boundaries. In both of these cases, the boundary value is not defined at the center of the ghost cell. The order determines the number of interior cells that are used in the extrapolation of the boundary value from the cell face to the center of the ghost cell, where the extrapolated value is then used in the regular stencil. For example, maxorder = 2 uses the boundary value and the first interior value to extrapolate to the ghost cell center; maxorder = 3 uses the boundary value and the first two interior values.

10.5 Curvilinear Coordinates

Some of the linear solvers support curvilinear coordinates including 1D spherical and 2d cylindrical (r, z). In those cases, the divergence operator has extra metric terms. If one does not want the solver to include the metric terms because they have been handled in other ways, one can turn them off with a setter funtion. For the cell-centered linear solvers *MLABecLaplacian* and *MLPoisson*, one can call setMetricTerm(bool) with false on the LPInfo object passed to the constructor of linear operators. For the node-based *MLNodeLaplacian*, one can call setRZCorrection (bool) with false on the *MLNodeLaplacian* object.

MLABecLaplacian and *MLPoisson* support both spherical and cylindrical coordinates, while *MLNodeLaplacian* supports only cylindrical at this time. Note that to use cylindrical coordinates with *MLNodeLaplacian*, the application code must scale sigma by the radial coordinate before calling setSigma().

10.6 Embedded Boundaries

AMReX supports multi-level solvers for use with embedded boundaries. These include 1) cell-centered solvers with homogeneous Neumann, homogeneous Dirichlet, or inhomogeneous Dirichlet boundary conditions on the EB faces, and 2) nodal solvers with homogeneous Neumann boundary conditions, or inflow velocity conditions on the EB faces.

To use a cell-centered solver with EB, one builds a linear operator MLEBABecLap with EBFArrayBoxFactory (instead of a MLABecLaplacian)

```
MLEBABecLap (const Vector<Geometry>& a_geom,
             const Vector<BoxArray>& a_grids,
             const Vector<DistributionMapping>& a_dmap,
             const LPInfo& a_info,
             const Vector<EBFArrayBoxFactory const*>& a_factory);
```

The usage of this EB-specific class is essentially the same as MLABecLaplacian.

The default boundary condition on EB faces is homogeneous Neumann.

To set homogeneous Dirichlet boundary conditions, call

```
ml_ebabeclap->setEBHomogDirichlet(lev, coeff);
```

where coeff can be a real number (i.e. the value is the same at every cell) or is the MultiFab holding the coefficient of the gradient at each cell with an EB face.

To set inhomogeneous Dirichlet boundary conditions, call

```
ml_ebabeclap->setEBDirichlet(lev, phi_on_eb, coeff);
```

where phi_on_eb is the MultiFab holding the Dirichlet values in every cut cell, and coeff again is a real number (i.e. the value is the same at every cell) or a MultiFab holding the coefficient of the gradient at each cell with an EB face.

Currently there are options to define the face-based coefficients on face centers vs face centroids, and to interpret the solution variable as being defined on cell centers vs cell centroids.

The default is for the solution variable to be defined at cell centers; to tell the solver to interpret the solution variable as living at cell centroids, you must set

```
ml_ebabeclap->setPhiOnCentroid();
```

The default is for the face-based coefficients to be defined at face centers; to tell the that the face-based coefficients should be interpreted as living at face centroids, modify the setBCoeffs command to be

```
ml_ebabeclap->setBCoeffs(lev, beta, MLMG::Location::FaceCentroid);
```

10.7 External Solvers

AMReX provides interfaces to the hypre preconditioners and solvers, including BoomerAMG, GMRES (all variants), PCG, and BICGStab as solvers, and BoomerAMG and Euclid as preconditioners. These can be called as as bottom solvers for both cell-centered and node-based problems.

If it is built with Hypre support, AMReX initializes Hypre by default in *amrex::Initialize*. If it is built with CUDA, AMReX will also set up Hypre to run on device by default. The user can choose to disable the Hypre initialization by AMReX with `ParmParse` parameter `amrex.init_hypre=[0|1]`.

By default the AMReX linear solver code always tries to geometrically coarsen the problem as much as possible. However, as we have mentioned, we can call `setMaxCoarseningLevel(0)` on the `LPInfo` object passed to the constructor of a linear operator to disable the coarsening completely. In that case the bottom solver is solving the residual correction form of the original problem. To build Hypre, follow the next steps:

```
1.- git clone https://github.com/hypre-space/hypre.git
2.- cd hypre/src
3.- ./configure
    (if you want to build hypre with long long int, do ./configure --enable-bigint )
4.- make install
5.- Create an environment variable with the HYPRE directory --
    HYPRE_DIR=/hypre_path/hypre/src/hypre
```

To use hypre, one must include `amrex/Src/Extern/HYPRE` in the build system. For examples of using hypre, we refer the reader to ABecLaplacian or NodeTensorLap.

The following parameter should be set to True if the problem to be solved has a singular matrix. In this case, the solution is only defined to within a constant. Setting this parameter to True replaces one row in the matrix sent to hypre from AMReX by a row that sets the value at one cell to 0.

- `hypre.adjust_singular_matrix`: Default is False.

The following parameters can be set in the inputs file to control the choice of preconditioner and smoother:

- `hypre.hypre_solver`: Default is BoomerAMG.
- `hypre.hypre_preconditioner`: Default is none; otherwise the type must be specified.
- `hypre.recompute_preconditioner`: Default true. Option to recompute the preconditioner.
- `hypre.write_matrix_files`: Default false. Option to write out matrix into text files.
- `hypre.overwrite_existing_matrix_files`: Default false. Option to over-write existing matrix files.

The following parameters can be set in the inputs file to control the BoomerAMG solver specifically:

- `hypre.bamg_verbose`: verbosity of BoomerAMG preconditioner. Default 0. See *HYPRE_BoomerAMGSetPrintLevel*
- `hypre.bamg_logging`: Default 0. See *HYPRE_BoomerAMGSetLogging*
- `hypre.bamg_coarsen_type`: Default 6. See *HYPRE_BoomerAMGSetCoarsenType*
- `hypre.bamg_cycle_type`: Default 1. See *HYPRE_BoomerAMGSetCycleType*
- `hypre.bamg_relax_type`: Default 6. See *HYPRE_BoomerAMGSetRelaxType*
- `hypre.bamg_relax_order`: Default 1. See *HYPRE_BoomerAMGSetRelaxOrder*
- `hypre.bamg_num_sweeps`: Default 2. See *HYPRE_BoomerAMGSetNumSweeps*
- `hypre.bamg_max_levels`: Default 20. See *HYPRE_BoomerAMGSetMaxLevels*
- `hypre.bamg_strong_threshold`: Default 0.25 for 2D, 0.57 for 3D. See *HYPRE_BoomerAMGSetStrongThreshold*
- `hypre.bamg_interp_type`: Default 0. See *HYPRE_BoomerAMGSetInterpType*

The user is referred to the hypre Hypre Reference Manual for full details on the usage of the parameters described briefly above.

AMReX can also use PETSc as a bottom solver for cell-centered problems. To build PETSc, follow the next steps:

```
1.- git clone https://github.com/petsc/petsc.git
2.- cd petsc
3.- ./configure --download-hypre=yes --prefix=build_dir
4.- Follow the steps given by petsc
5.- Create an environment variable with the PETSC directory --
    PETSC_DIR=/petsc_path/petsc/build_dir
```

To use PETSc, one must include `amrex/Src/Extern/PETSc` in the build system. For an example of using PETSc, we refer the reader to the tutorial, ABecLaplacian.

10.8 Tensor Solve

Application codes that solve the Navier-Stokes equations need to evaluate the viscous term; solving for this term implicitly requires a multi-component solve with cross terms. Because this is a commonly used motif, we provide a tensor solve for cell-centered velocity components.

Consider a velocity field $U = (u, v, w)$ with all components co-located on cell centers. The viscous term can be written in vector form as

$$\nabla \cdot (\eta \nabla U) + \nabla \cdot (\eta (\nabla U)^T) + \nabla \cdot ((\kappa - \frac{2}{3}\eta)(\nabla \cdot U))$$

and in 3-d Cartesian component form as

$$((\eta u_x)_x + (\eta u_y)_y + (\eta u_z)_z) + ((\eta u_x)_x + (\eta v_x)_y + (\eta w_x)_z) + ((\kappa - \frac{2}{3}\eta)(u_x + v_y + w_z))_x$$
$$((\eta v_x)_x + (\eta v_y)_y + (\eta v_z)_z) + ((\eta u_y)_x + (\eta v_y)_y + (\eta w_y)_z) + ((\kappa - \frac{2}{3}\eta)(u_x + v_y + w_z))_y$$
$$((\eta w_x)_x + (\eta w_y)_y + (\eta w_z)_z) + ((\eta u_z)_x + (\eta v_z)_y + (\eta w_z)_z) + ((\kappa - \frac{2}{3}\eta)(u_x + v_y + w_z))_z$$

Here η is the dynamic viscosity and κ is the bulk viscosity.

We evaluate the following terms from the above using the `MLABecLaplacian` and `MLEBABecLaplacian` operators;

$$((\frac{4}{3}\eta + \kappa)u_x)_x + (\eta u_y)_y + (\eta u_z)_z$$
$$(\eta v_x)_x + ((\frac{4}{3}\eta + \kappa)v_y)_y + (\eta v_z)_z$$
$$(\eta w_x)_x + (\eta w_y)_y + ((\frac{4}{3}\eta + \kappa)w_z)_z$$

the following cross-terms are evaluated separately using the `MLTensorOp` and `MLEBTensorOp` operators.

$$((\kappa - \frac{2}{3}\eta)(v_y + w_z))_x + (\eta v_x)_y + (\eta w_x)_z$$
$$(\eta u_y)_x + ((\kappa - \frac{2}{3}\eta)(u_x + w_z))_y + (\eta w_y)_z$$
$$(\eta u_z)_x + (\eta v_z)_y - ((\kappa - \frac{2}{3}\eta)(u_x + v_y))_z$$

The code below is an example of how to set up the solver to compute the viscous term *divtau* explicitly:

```
Box domain(geom[0].Domain());

// Set BCs for Poisson solver in bc_lo, bc_hi
...

//
// First define the operator "ebtensorop"
// Note we call LPInfo().setMaxCoarseningLevel(0) because we are only applying the␣
↪operator,
//      not doing an implicit solve
//
//       (alpha * a - beta * (del dot b grad)) sol
//
// LPInfo                        info;
MLEBTensorOp ebtensorop(geom, grids, dmap, LPInfo().setMaxCoarseningLevel(0),
                        amrex::GetVecOfConstPtrs(ebfactory));

// It is essential that we set MaxOrder of the solver to 2
// if we want to use the standard sol(i)-sol(i-1) approximation
// for the gradient at Dirichlet boundaries.
// The solver's default order is 3 and this uses three points for the
// gradient at a Dirichlet boundary.
ebtensorop.setMaxOrder(2);

// LinOpBCType Definitions are in amrex/Src/Boundary/AMReX_LO_BCTYPES.H
ebtensorop.setDomainBC ( {(LinOpBCType)bc_lo[0], (LinOpBCType)bc_lo[1], (LinOpBCType)bc_
↪lo[2]},
```

(continues on next page)

(continued from previous page)

```
                        {(LinOpBCType)bc_hi[0], (LinOpBCType)bc_hi[1], (LinOpBCType)bc_
↪hi[2]} );

// Return div (eta grad)) phi
ebtensorop.setScalars(0.0, -1.0);

amrex::Vector<amrex::Array<std::unique_ptr<amrex::MultiFab>, AMREX_SPACEDIM>> b;
b.resize(max_level + 1);

// Compute the coefficients
for (int lev = 0; lev < nlev; lev++)
{
    // We average eta onto faces
    for(int dir = 0; dir < AMREX_SPACEDIM; dir++)
    {
        BoxArray edge_ba = grids[lev];
        edge_ba.surroundingNodes(dir);
        b[lev][dir] = std::make_unique<MultiFab>(edge_ba, dmap[lev], 1, nghost, MFInfo(),
↪ *ebfactory[lev]);
    }

    average_cellcenter_to_face( GetArrOfPtrs(b[lev]), *etan[lev], geom[lev] );

    b[lev][0] -> FillBoundary(geom[lev].periodicity());
    b[lev][1] -> FillBoundary(geom[lev].periodicity());
    b[lev][2] -> FillBoundary(geom[lev].periodicity());

    ebtensorop.setShearViscosity  (lev, GetArrOfConstPtrs(b[lev]));
    ebtensorop.setEBShearViscosity(lev, (*eta[lev]));

    ebtensorop.setLevelBC ( lev, GetVecOfConstPtrs(vel)[lev] );
}

MLMG solver(ebtensorop);

solver.apply(GetVecOfPtrs(divtau), GetVecOfPtrs(vel));
```

10.9 Multi-Component Operators

This section discusses solving linear systems in which the solution variable ϕ has multiple components. An example (implemented in the `MultiComponent` tutorial) might be:

$$D(\phi)_i = \sum_{i=1}^{N} \alpha_{ij} \nabla^2 \phi_j$$

(Note: only operators of the form $D : \mathbb{R}^n \rightarrow \mathbb{R}^n$ are currently allowed.)

- To implement a multi-component *cell-based* operator, inherit from the `MLCellLinOp` class. Override the `getNComp` function to return the number of components (`N`)that the operator will use. The solution and rhs fabs must also have at least one ghost node. `Fapply`, `Fsmooth`, `Fflux` must be implemented such that the solution and rhs fabs all have `N` components.

- Implementing a multi-component *node-based* operator is slightly different. A MC nodal operator must specify that the reflux-free coarse/fine strategy is being used by the solver.

```
solver.setCFStrategy(MLMG::CFStrategy::ghostnodes);
```

  The reflux-free method circumvents the need to implement a special `reflux` at the coarse-fine boundary. This is accomplished by using ghost nodes. Each AMR level must have 2 layers of ghost nodes. The second (outermost) layer of nodes is treated as constant by the relaxation, essentially acting as a Dirichlet boundary. The first layer of nodes is evolved using the relaxation, in the same manner as the rest of the solution. When the residual is restricted onto the coarse level (in `reflux`) this allows the residual at the coarse-fine boundary to be interpolated using the first layer of ghost nodes. Fig. 10.1 illustrates the how the coarse-fine update takes place.

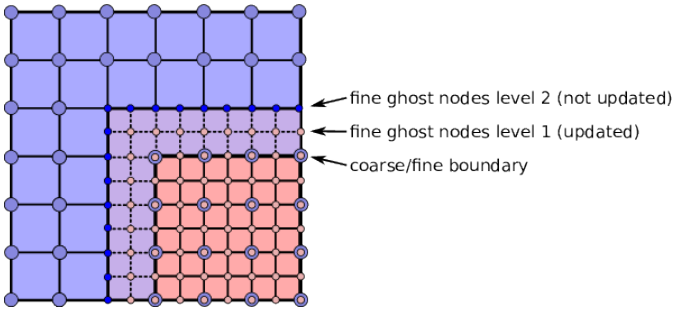


Fig. 10.1: Reflux-free coarse-fine boundary update. Level 2 ghost nodes (small dark blue) are interpolated from coarse boundary. Level 1 ghost nodes are updated during the relaxation along with all the other interior fine nodes. Coarse nodes (large blue) on the coarse/fine boundary are updated by restricting with interior nodes and the first level of ghost nodes. Coarse nodes underneath level 2 ghost nodes are not updated. The remaining coarse nodes are updates by restriction.

  The MC nodal operator can inherit from the `MCNodeLinOp` class. `Fapply`, `Fsmooth`, and `Fflux` must update level 1 ghost nodes that are inside the domain. *interpolation* and *restriction* can be implemented as usual. *reflux* is a straightforward restriction from fine to coarse, using level 1 ghost nodes for restriction as described above.

See `amrex-tutorials/ExampleCodes/LinearSolvers/MultiComponent` for a complete working example.

CHAPTER

ELEVEN

PARTICLES

In addition to the tools for working with mesh data described in previous chapters, AMReX also provides data structures and iterators for performing data-parallel particle simulations. Our approach is particularly suited to particles that interact with data defined on a (possibly adaptive) block-structured hierarchy of meshes. Example applications include Particle-in-Cell (PIC) simulations, Lagrangian tracers, or particles that exert drag forces onto a fluid, such as in multi-phase flow calculations. The overall goals of AMReX's particle tools are to allow users flexibility in specifying how the particle data is laid out in memory and to handle the parallel communication of particle data. In the following sections, we give an overview of AMReX's particle classes and how to use them.

11.1 The Particle

The particle classes can be used by including the header AMReX_Particles.H. The most basic particle data structure is the particle itself:

```
Particle<3, 2> p;
```

This is a templated data type, designed to allow flexibility in the number and type of components that the particles carry. The first template parameter is the number of extra `Real` variables this particle will have (either single or double precision[1]), while the second is the number of extra integer variables. It is important to note that this is the number of *extra* real and integer variables; a particle will always have at least `BL_SPACEDIM` real components that store the particle's position and 2 integer components that store the particle's `id` and `cpu` numbers.[2]

The particle struct is designed to store these variables in a way that minimizes padding, which in practice means that the `Real` components always come first, and the integer components second. Additionally, the required particle variables are stored before the optional ones, for both the real and the integer components. For example, say we want to define a particle type that stores a mass, three velocity components, and two extra integer flags. Our particle struct would be set up like:

```
Particle<4, 2> p;
```

and the order of the particle components in would be (assuming `BL_SPACEDIM` is 3): `x y z m vx vy vz id cpu flag1 flag2`.[3]

[1] Particles default to double precision for their real data. To use single precision, compile your code with `USE_SINGLE_PRECISION_PARTICLES=TRUE`.

[2] Note that `cpu` stores the number of the process the particle was *generated* on, not the one it's currently assigned to. This number is set on initialization and never changes, just like the particle `id`. In essence, the particles have two integer id numbers, and only the combination of the two is unique. This was done to facilitate the creation of particle initial conditions in parallel.

[3] Note that for the extra particle components, which component refers to which variable is an application-specific convention - the particles have 4 extra real comps, but which one is "mass" is up to the user. We suggest using an `enum` to keep these indices straight; please see `amrex-tutorials/ExampleCodes/Particles/ElectrostaticPIC/ElectrosticParticleContainer.H` for an example of this.

11.1.1 Setting Particle data

The `Particle` struct provides a number of methods for getting and setting a particle's data. For the required particle components, there are special, named methods. For the "extra" real and integer data, you can use the `rdata` and `idata` methods, respectively.

```
Particle<2, 2> p;

p.pos(0) = 1.0;
p.pos(1) = 2.0;
p.pos(2) = 3.0;
p.id() = 1;
p.cpu()  = 0;

// p.rdata(0) is the first extra real component, not the
// first real component overall
p.rdata(0) = 5.0;
p.rdata(1) = 5.0;

// and likewise for p.idata(0);
p.idata(0) = 17;
p.idata(1) = -64;
```

11.2 The ParticleContainer

One particle by itself is not very useful. To do real calculations, a collection of particles needs to be defined, and the location of the particles within the AMR hierarchy (and the corresponding MPI process) needs to be tracked as the particle positions change. To do this, we provide the `ParticleContainer` class:

```
ParticleContainer<3, 2, 4, 4> mypc;
```

11.2.1 Arrays-of-Structs and Structs-of-Arrays

Like the `Particle` class itself, the `ParticleContainer` class is templated. The first two template parameters have the same meaning as before: they define the number of each type of variables that the particles in this container will store. Particles added to the container are stored in the Array-of-Structs (AoS) style. In addition, there are two more optional template parameters that allow the user to specify additional particle variables that will be stored in Struct-of-Array (SoA) form. The difference between Array-of-Struct and Struct-of-Array data is in how the data is laid out in memory. For the AoS data, all the variables associated with particle 1 are next to each other in memory, followed by all the variables associated with particle 2, and so on. For variables stored in SoA style, all the particle data for a given component is next to each other in memory, and each component is stored in a separate array. For convenience, we (arbitrarily) refer to the components in the particle struct as particle *data*, and components stored in the Struct-of-Arrays as particle *attributes*. See the figure *below* for an illustration.

To see why the distinction between AoS and SoA data is important, consider the following extreme case. Say you have particles that carry 100 different components, but that most of the time, you only need to do calculations involving 3 of them (say, the particle positions) at once. In this case, storing all 100 particle variables in the particle struct is clearly inefficient, since most of the time you are reading 97 extra variables into cache that you will never use. By splitting up the particle variables into stuff that gets used all the time (stored in the AoS) and stuff that only gets used infrequently (stored in the SoA), you can in principle achieve much better cache reuse. Of course, the usage pattern

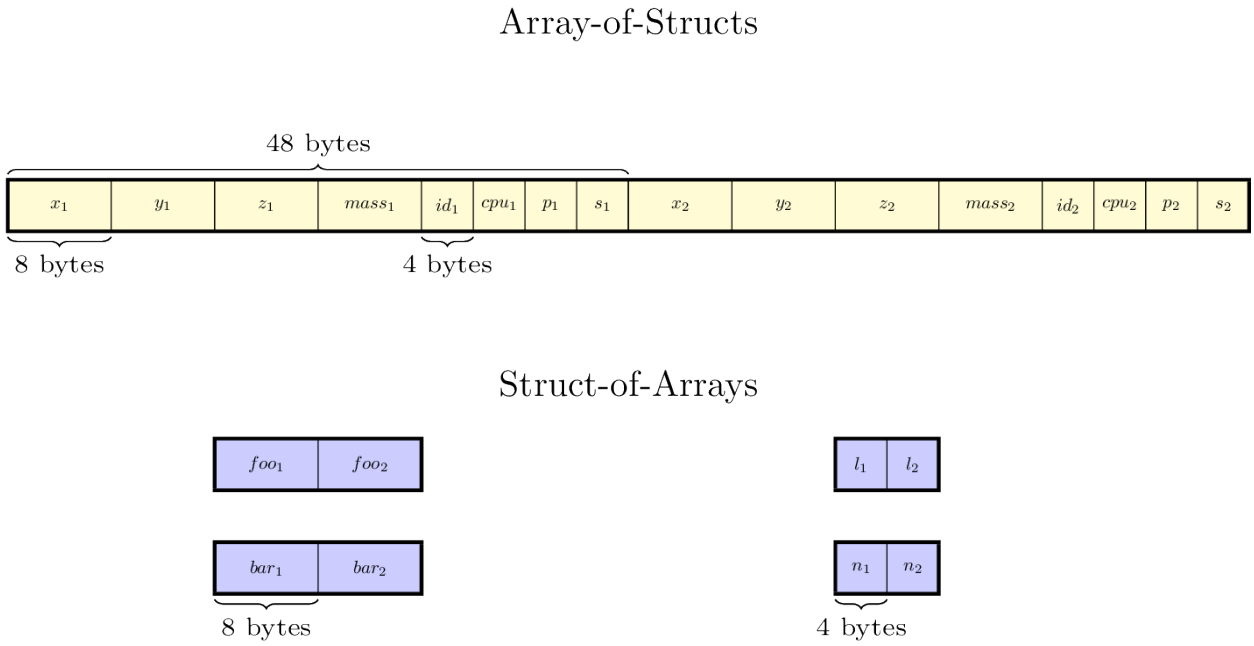


Fig. 11.1: An illustration of how the particle data for a single tile is arranged in memory. This particle container has been defined with `NStructReal = 1`, `NStructInt = 2`, `NArrayReal = 2`, and `NArrayInt = 2`. In this case, each tile in the particle container has five arrays: one with the particle struct data, two additional real arrays, and two additional integer arrays. In the tile shown, there are only 2 particles. We have labelled the extra real data member of the particle struct to be `mass`, while the extra integer members of the particle struct are labeled `p`, and `s`, for "phase" and "state". The variables in the real and integer arrays are labelled `foo`, `bar`, `l`, and `n`, respectively. We have assumed that the particles are double precision.

of your application likely won't be so clear-cut. Flexibility in how the particle data is stored also makes it easier to interface between AMReX and already-existing Fortran subroutines.

Note that while "extra" particle data can be stored in either the SoA or AoS style, the particle positions and id numbers are **always** stored in the particle structs. This is because these particle variables are special and used internally by AMReX to assign the particles to grids and to mark particles as valid or invalid, respectively.

11.2.2 Constructing ParticleContainers

A particle container is always associated with a particular set of AMR grids and a particular set of DistributionMaps that describes which MPI processes those grids live on. For example, if you only have one level, you can define a `ParticleContainer` to store particles on that level using the following constructor:

```
ParticleContainer (const Geometry            & geom,
                   const DistributionMapping & dmap,
                   const BoxArray            & ba);
```

Or, if you have multiple levels, you can use following constructor instead:

```
ParticleContainer (const Vector<Geometry>            & geom,
                   const Vector<DistributionMapping> & dmap,
                   const Vector<BoxArray>            & ba,
                   const Vector<int>                 & rr);
```

Note the set of grids used to define the `ParticleContainer` doesn't have to be the same set used to define the simulation's mesh data. However, it is often desirable to have the two hierarchies track each other. If you are using an `AmrCore` class in your simulation (see the Chapter on *AmrCore Source Code*), you can achieve this by using the `AmrParticleContainer` class. The constructor for this class takes a pointer to your AmrCore derived class, instead:

```
AmrTracerParticleContainer (AmrCore* amr_core);
```

In this case, the `Vector<BoxArray>` and `Vector<DistributionMap>` used by your `ParticleContainer` will be updated automatically to match those in your `AmrCore`.

11.2.3 The ParticleTile

The `ParticleContainer` stores the particle data in a manner prescribed by the set of AMR grids used to define it. Local particle data is always stored in a data structure called a `ParticleTile`, which contains a mixture of AoS and SoA components as described above. The tiling behavior of `ParticleTile` is determined by the parameter, `particle.do_tiling`:

- If `particles.do_tiling=0`, then there is always exactly one `ParticleTile` per grid. This is equivalent to setting a very large `particles.tile_size` in each direction.
- If `particles.do_tiling=1`, then each grid can have multiple `ParticleTile` objects associated with it based on the `particles.tile_size` parameter.

The AMR grid to which a particle is assigned, is determined by examining its position and binning it, using the domain left edge as an offset. By default, a particle is assigned to the finest level that contains its position, although this behavior can be tweaked if desired.

Note: `ParticleTile` data tiling with *MFIter* behaves differently than mesh data. With mesh data, the tiling is strictly logical –the data is laid out in memory the same way whether tiling is turned on or off. With particle data, however,

the particles are actually stored in different arrays when tiling is enabled. As with mesh data, the particle tile size can be tuned so that an entire tile's worth of particles will fit into a cache line at once.

11.2.4 Redistribute

Once the particles move, their data may no longer be in the right place in the container. They can be reassigned by calling the `Redistribute()` method of `ParticleContainer`. After calling this method, all the particles will be moved to their proper places in the container, and all invalid particles (particles with id set to `-1`) will be removed. All the MPI communication needed to do this happens automatically.

Application codes will likely want to create their own derived ParticleContainer class that specializes the template parameters and adds additional functionality, like setting the initial conditions, moving the particles, etc. See the *particle tutorials* for examples of this. .. particle tutorials: https://amrex-codes.github.io/amrex/tutorials_html/Particles_Tutorial.html

11.3 Initializing Particle Data

In the following code snippet, we demonstrate how to set particle initial conditions for both SoA and AoS data. We loop over all the tiles using `MFIter`, and add as many particles as we want to each one.

```
for (MFIter mfi = MakeMFIter(lev); mfi.isValid(); ++mfi) {

    // ``particles'' starts off empty
    auto& particles = GetParticles(lev)[std::make_pair(mfi.index(),
                                       mfi.LocalTileIndex())];

    ParticleType p;
    p.id()   = ParticleType::NextID();
    p.cpu()  = ParallelDescriptor::MyProc();
    p.pos(0) = ...
    etc...

    // AoS real data
    p.rdata(0) = ...
    p.rdata(1)  = ...

    // AoS int data
    p.idata(0) = ...
    p.idata(1) = ...

    // Particle real attributes (SoA)
    std::array<double, 2> real_attribs;
    real_attribs[0] = ...
    real_attribs[1] = ...

    // Particle int attributes (SoA)
    std::array<int, 2> int_attribs;
    int_attribs[0] = ...
    int_attribs[1]  = ...
```

(continues on next page)

(continued from previous page)

```
    particles.push_back(p);
    particles.push_back_real(real_attribs);
    particles.push_back_int(int_attribs);

    // ... add more particles if desired ...
  }
```

Often, it makes sense to have each process only generate particles that it owns, so that the particles are already in the right place in the container. In general, however, users may need to call `Redistribute()` after adding particles, if the processes generate particles they don't own (for example, if the particle positions are perturbed from the cell centers and thus end up outside their parent grid).

11.4 Adding particle components at runtime

In addition to the components specified as template parameters, you can also add additional `Real` and `int` components at runtime. These components will be stored in Struct-of-Array style. To add a runtime component, use the `AddRealComp` and `AddIntComp` methods of `ParticleContainer`, like so:

```
const bool communicate_this_comp = true;
for (int i = 0; i < num_runtime_real; ++i)
{
    AddRealComp(communicate_this_comp);
}
for (int i = 0; i < num_runtime_int; ++i)
{
    AddIntComp(communicate_this_comp);
}
```

Runtime-added components can be accessed like regular Struct-of-Array data. The new components will be added at the end of the compile-time defined ones.

When you are using runtime components, it is crucial that when you are adding particles to the container, you call the `DefineAndReturnParticleTile` method for each tile prior to adding any particles. This will make sure the space for the new components has been allocated. For example, in the above section on *initializing particle data*, we accessed the particle tile data using the `GetParticles` method. If we runtime components are used, `DefineAndReturnParticleTile` should be used instead:

```
for(MFIter mfi = MakeMFIter(lev); mfi.isValid(); ++mfi)
{
    // instead of this...
    // auto& particles = GetParticles(lev)[std::make_pair(mfi.index(),
    //                                    mfi.LocalTileIndex())];

    // we do this...
    auto& particle_tile = DefineAndReturnParticleTile(lev, mfi);

    // add particles to particle_tile as above...
}
```

11.5 Iterating over Particles

To iterate over the particles on a given level in your container, you can use the `ParIter` class, which comes in both const and non-const flavors. For example, to iterate over all the AoS data:

```
using MyParIter = ConstParIter<2*BL_SPACEDIM>;
for (MyParIter pti(pc, lev); pti.isValid(); ++pti) {
    const auto& particles = pti.GetArrayOfStructs();
    for (const auto& p : particles) {
        // do stuff with p...
    }
}
```

The outer loop will execute once every grid (or tile, if tiling is enabled) *that contains particles*; grids or tiles that don't have any particles will be skipped. You can also access the SoA data using the $ParIter$ as follows:

```
using MyParIter = ParIter<0, 0, 2, 2>;
for (MyParIter pti(pc, lev); pti.isValid(); ++pti) {
    auto& particle_attributes = pti.GetStructOfArrays();
    RealVector& real_comp0 = particle_attributes.GetRealData(0);
    IntVector&  int_comp1  = particle_attributes.GetIntData(1);
    for (int i = 0; i < pti.numParticles; ++i) {
        // do stuff with your SoA data...
    }
}
```

11.6 Passing particle data into Fortran routines

Because the AMReX particle struct is a Plain-Old-Data type, it is interoperable with Fortran when the `bind(C)` attribute is used. It is therefore possible to pass a grid or tile worth of particles into fortran routines for processing, instead of iterating over them in C++. You can also define a Fortran derived type that is equivalent to C struct used for the particles. For example:

```
use amrex_fort_module, only: amrex_particle_real
use iso_c_binding ,    only: c_int

type, bind(C)  :: particle_t
   real(amrex_particle_real) :: pos(3)
   real(amrex_particle_real) :: vel(3)
   real(amrex_particle_real) :: acc(3)
   integer(c_int)   :: id
   integer(c_int)   :: cpu
end type particle_t
```

is equivalent to a particle struct you get with `Particle<6, 0>`. Here, `amrex_particle_real` is either single or doubled precision, depending on whether `USE_SINGLE_PRECISION_PARTICLES` is `TRUE` or not. We recommend always using this type in Fortran routines that work on particle data to avoid hard-to-debug incompatibilities between floating point types.

11.7 Interacting with Mesh Data

It is common to want to have the mesh communicate information to the particles and vice versa. For example, in Particle-in-Cell calculations, the particles deposit their charges onto the mesh, and later, the electric fields computed on the mesh are interpolated back to the particles. Below, we show examples of both these sorts of operations.

```
Ex.FillBoundary(gm.periodicity());
Ey.FillBoundary(gm.periodicity());
Ez.FillBoundary(gm.periodicity());
for (MyParIter pti(MyPC, lev); pti.isValid(); ++pti) {
    const Box& box = pti.validbox();

    const auto& particles = pti.GetArrayOfStructs();
    int nstride = particles.dataShape().first;
    const long np  = pti.numParticles();

    const FArrayBox& exfab = Ex[pti];
    const FArrayBox& eyfab = Ey[pti];
    const FArrayBox& ezfab = Ex[pti];

    interpolate_cic(particles.data(), nstride, np,
                    exfab.dataPtr(), eyfab.dataPtr(), ezfab.dataPtr(),
                    box.loVect(), box.hiVect(), plo, dx, &ng);
    }
```

Here, `interpolate_cic` is a Fortran subroutine that actually performs the interpolation on a single box. `Ex`, `Ey`, and `Ez` are MultiFabs that contain the electric field data. These MultiFabs must be defined with the correct number of ghost cells to perform the desired type of interpolation, and we call `FillBoundary` prior to the Fortran call so that those ghost cells will be up-to-date.

In this example, we have assumed that the `ParticleContainer MyPC` has been defined on the same grids as the electric field MultiFabs, so that we use the `ParIter` to index into the MultiFabs to get the data associated with current tile. If this is not the case, then an additional copy will need to be performed. However, if the particles are distributed in an extremely uneven fashion, it is possible that the load balancing improvements associated with the two-grid approach are worth the cost of the extra copy.

The inverse operation, in which the particles communicate data *to* the mesh, is quite similar:

```
rho.setVal(0.0, ng);
for (MyParIter pti(*this, lev); pti.isValid(); ++pti) {
    const Box& box = pti.validbox();

    const auto& particles = pti.GetArrayOfStructs();
    int nstride = particles.dataShape().first;
    const long np  = pti.numParticles();

    FArrayBox& rhofab = (*rho[lev])[pti];

    deposit_cic(particles.data(), nstride, np, rhofab.dataPtr(),
                box.loVect(), box.hiVect(), plo, dx);
    }

rho.SumBoundary(gm.periodicity());
```

As before, we loop over all our particles, calling a Fortran routine that deposits them on to the appropriate `FArrayBox`

rhofab. The rhofab must have enough ghost cells to cover the support of all the particles associated with them. Note that we call SumBoundary instead of FillBoundary after performing the deposition, to add up the charge in the ghost cells surrounding each Fab into the corresponding valid cells.

For a complete example of an electrostatic PIC calculation that includes static mesh refinement, please see the *Electrostatic PIC tutorial*.

11.8 Short Range Forces

In a PIC calculation, the particles don't interact with each other directly; they only see each other through the mesh. An alternative use case is particles that exert short-range forces on each other. In this case, beyond some cut-off distance, the particles don't interact with each other and therefore don't need to be included in the force calculation. Our approach to these kind of particles is to fill "neighbor buffers" on each tile that contain copies of the particles on neighboring tiles that are within some number of cells N_g of the tile boundaries. See Fig. 11.2, below for an illustration. By choosing the number of ghost cells to match the interaction radius of the particles, you can capture all of the neighbors that can possibly influence the particles in the valid region of the tile. The forces on the particles on different tiles can then be computed independently of each other using a variety of methods.

For a ParticleContainer that does this neighbor finding, please see NeighborParticleContainer in amrex/Src/Particles/AMReX_NeighborParticleContainer.H. The NeighborParticleContainer has additional methods called fillNeighbors() and clearNeighbors() that fill the neighbors data structure with copies of the proper particles. A tutorial that uses these features is available at NeighborList. In this tutorial the function **void** MDParticleContainer:computeForces() computes the forces on a given tile via direct summation over the real and neighbor particles, as follows:

```
void MDParticleContainer::computeForces()
{
    BL_PROFILE("MDParticleContainer::computeForces");

    const int lev = 0;
    const Geometry& geom = Geom(lev);
    auto& plev  = GetParticles(lev);

    for(MFIter mfi = MakeMFIter(lev); mfi.isValid(); ++mfi)
    {
        int gid = mfi.index();
        int tid = mfi.LocalTileIndex();
        auto index = std::make_pair(gid, tid);

        auto& ptile = plev[index];
        auto& aos   = ptile.GetArrayOfStructs();
        const size_t np = aos.numParticles();

        auto nbor_data = m_neighbor_list[lev][index].data();
        ParticleType* pstruct = aos().dataPtr();

       // now we loop over the neighbor list and compute the forces
        AMREX_FOR_1D ( np, i,
        {
            ParticleType& p1 = pstruct[i];
            p1.rdata(PIdx::ax) = 0.0;
```

(continues on next page)

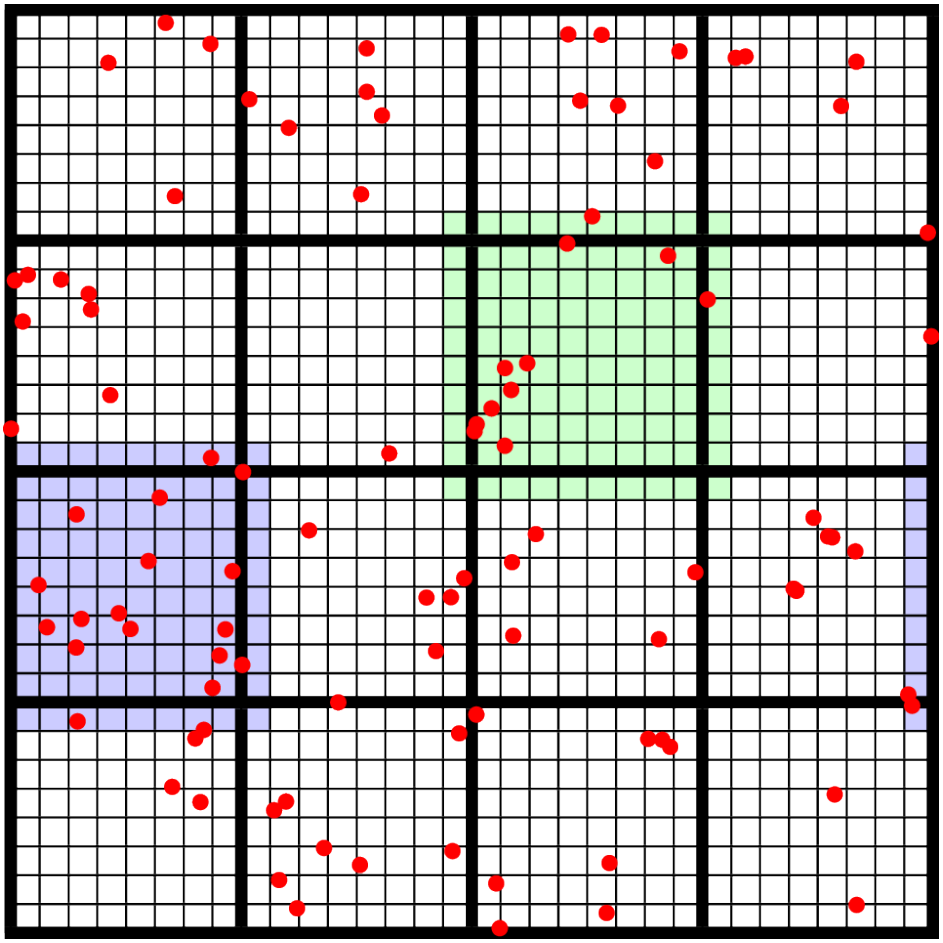


Fig. 11.2: : An illustration of filling neighbor particles for short-range force calculations. Here, we have a domain consisting of one 32×32 grid, broken up into 8×8 tiles. The number of ghost cells is taken to be 1. For the tile in green, particles on other tiles in the entire shaded region will copied and packed into the green tile's neighbor buffer. These particles can then be included in the force calculation. If the domain is periodic, particles in the grown region for the blue tile that lie on the other side of the domain will also be copied, and their positions will modified so that a naive distance calculation between valid particles and neighbors will be correct.

(continued from previous page)

```
            p1.rdata(PIdx::ay) = 0.0;
            p1.rdata(PIdx::az) = 0.0;

            for (const auto& p2 : nbor_data.getNeighbors(i))
            {
                Real dx = p1.pos(0) - p2.pos(0);
                Real dy = p1.pos(1) - p2.pos(1);
                Real dz = p1.pos(2) - p2.pos(2);

                Real r2 = dx*dx + dy*dy + dz*dz;
                r2 = amrex::max(r2, Params::min_r*Params::min_r);

                if (r2 > Params::cutoff*Params::cutoff) return;

                Real r = sqrt(r2);

                Real coef = (1.0 - Params::cutoff / r) / r2;
                p1.rdata(PIdx::ax) += coef * dx;
                p1.rdata(PIdx::ay) += coef * dy;
                p1.rdata(PIdx::az) += coef * dz;
            }
        });
    }
}
```

Doing a direct N^2 summation over the particles on a tile is avoided by binning the particles by cell and building a neighbor list. The data structure used to represent the neighbor lists is illustrated in Fig. 11.3.

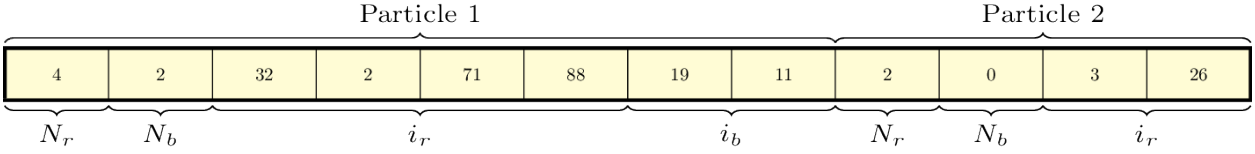


Fig. 11.3: : An illustration of the neighbor list data structure used by AMReX. The list for each tile is represented by an array of integers. The first number in the array is the number of real (i.e., not in the neighbor buffers) collision partners for the first particle on this tile, while the second is the number of collision partners from nearby tiles in the neighbor buffer. Based on the number of collision partners, the next several entries are the indices of the collision partners in the real and neighbor particle arrays, respectively. This pattern continues for all the particles on this tile.

This array can then be used to compute the forces on all the particles in one scan. Users can define their own `NeighborParticleContainer` subclasses that have their own collision criteria by overloading the virtual `check_pair` function.

11.9 Particle IO

AMReX provides routines for writing particle data to disk for analysis, visualization, and for checkpoint / restart. The most important methods are the `WritePlotFile`, `Checkpoint`, and `Restart` methods of `ParticleContainer`, which all use a parallel-aware binary file format for reading and writing particle data on a grid-by-grid basis. These methods are designed to complement the functions in AMReX_PlotFileUtil.H for performing mesh data IO. For example:

```
WriteMultiLevelPlotfile("plt00000", output_levs, GetVecOfConstPtrs(output),
                        varnames, geom, 0.0, level_steps, outputRR);
pc.Checkpoint("plt00000", "particle0");
```

will create a plot file called "plt00000" and write the mesh data in `output` to it, and then write the particle data in a subdirectory called "particle0". There is also the `WriteAsciiFile` method, which writes the particles in a human-readable text format. This is mainly useful for testing and debugging.

The binary file format is currently readable by `yt`. In additional, there is a Python conversion script in `amrex/Tools/Py_util/amrex_particles_to_vtp` that can convert both the ASCII and the binary particle files to a format readable by Paraview. See the chapter on *Visualization* for more information on visualizing AMReX datasets, including those with particles.

11.10 Inputs parameters

There are several runtime parameters users can set in their `inputs` files that control the behavior of the AMReX particle classes. These are summarized below. They should be preceded by "particles" in your inputs deck.

The first set of parameters concerns the tiling capability of the ParticleContainer. If you are seeing poor performance with OpenMP, the first thing to look at is whether there are enough tiles available for each thread to work on.

	Description	Type	Default
do_tiling	Whether to use tiling for particles. Should be on when using OpenMP, and off when running on GPUs.	Bool	False
tile_size	If tiling is on, the maximum tile_size to in each direction	Ints	1024000,8,8

The next set concerns runtime parameters that control the particle IO. Parallel file systems tend not to like it when too many MPI tasks touch the disk at once. Additionally, performance can degrade if all MPI tasks try writing to the same file, or if too many small files are created. In general, the "correct" values of these parameters will depend on the size of your problem (i.e., number of boxes, number of MPI tasks), as well as the system you are using. If you are experiencing problems with particle IO, you could try varying some / all of these parameters.

	Description	Type	Default
particles_nfiles	How many files to use when writing particle data to plt directories	Int	1024
nreaders	How many MPI tasks to use as readers when initializing particles from binary files.	Ints	64
nparts_per_read	How many particles each task should read from said files before calling Redistribute	Ints	100000
datadigits_read	This for backwards compatibility, don't use unless you need to read and old (pre mid 2017) AMReX dataset.	Int	5
use_prepost	This is an optimization for large particle datasets that groups MPI calls needed during the IO together. Try it seeing poor IO speeds on large problems.	Bool	False

The following runtime parameters affect the behavior of virtual particles in Nyx.

	Description	Type	Default
aggregation_type	How to create virtual particles from finer levels. The options are: "None" - don't do any aggregation. "Cell" - when creating virtuals, combine all particles that are in the same cell.	String	"None"
aggregation_buffer	If aggregation on, the number of cells around the coarse/fine boundary in which no aggregation should be performed.	Int	2

Finally, the *amrex.use_gpu_aware_mpi* switch can also affect the behavior of the particle communication routines when running on GPU platforms like Summit. We recommend leaving it off.

CHAPTER

TWELVE

FORTRAN INTERFACE

The core of AMReX is written in C++. For Fortran users who want to write all of their programs in Fortran, AMReX provides Fortran interfaces around most of functionalities except for the `AmrLevel` class (see the chapter on *Amr Source Code*) and particles (see the chapter on *Particles*). We should not confuse the Fortran interface in this chapter with the Fortran kernel functions called inside `MFIter` loops in codes (see the section on *Fortran and C++ Kernels*). For the latter, Fortran is used in some sense as a domain-specific language with native multi-dimensional arrays, whereas here Fortran is used to drive the whole application code. In order to better understand AMReX, Fortran interface users should read the rest of the documentation except for the Chapters on *Amr Source Code* & *Particles*.

12.1 Getting Started

We have discussed AMReX's build systems in the chapter on *Building AMReX*. To build with GNU Make, we need to include the Fortran interface source tree into the make system. The source codes for the Fortran interface are in `amrex/Src/F_Interfaces` and there are several sub-directories. The "Base" directory includes sources for the basic functionality, the "AmrCore" directory wraps around the `AmrCore` class (see the chapter on *AmrCore Source Code*), and the "Octree" directory adds support for octree type of AMR grids. Each directory has a "Make.package" file that can be included in make files (see HelloWorld_F and Advection_F in the tutorials for examples). The libamrex approach includes the Fortran interface by default.

A simple example can be found at `amrex-tutorials/Basic/HelloWorld_F/`. The source code is shown below in its entirety.

```
program main
  use amrex_base_module
  implicit none
  call amrex_init()
  if (amrex_parallel_ioprocessor()) then
     print *, "Hello world!"
  end if
  call amrex_finalize()
end program main
```

To access the AMReX Fortran interfaces, we can use these three modules, `amrex_base_module` for the basics functionalities (Section 2 *The Basics*), `amrex_amrcore_module` for AMR support (Section 3 *Amr Core Infrastructure*) and `amrex_octree_module` for octree style AMR (Section 4 *Octree*).

12.2 The Basics

Module amrex_base_module is a collection of various Fortran modules providing interfaces to most of the basics of AMReX C++ library (see the chapter on *Basics*). These modules shown in this section can be used without being explicitly included because they are included by amrex_base_module.

The spatial dimension is an integer parameter amrex_spacedim. We can also use the AMREX_SPACEDIM macro in preprocessed Fortran codes (e.g., .F90 files) just like in the C++ codes. Unlike in C++, the convention for AMReX Fortran interface is that coordinate direction index starts at 1.

There is an integer parameter amrex_real, a Fortran kind parameter for **real**. Fortran **real**(amrex_real) corresponds to amrex::Real in C++, which is either double or single precision depending the setting of precision.

The module amrex_parallel_module (amrex/Src/F_Interfaces/Base/AMReX_parallel_mod.F90) includes wrappers to the ParallelDescriptor namespace, which is in turn a wrapper to the parallel communication library used by AMReX (e.g. MPI).

The module amrex_parmparse_module (amrex/Src/Base/AMReX_parmparse_mod.F90) provides interface to ParmParse (see the section on *ParmParse*). Here are some examples.

```
type(amrex_parmparse) :: pp
integer :: n_cell, max_grid_size
call amrex_parmparse_build(pp)
call pp%get("n_cell", n_cell)
max_grid_size = 32 ! default size
call pp%query("max_grid_size", max_grid_size)
call amrex_parmpase_destroy(pp) ! optional if compiler supports finalization
```

Finalization is a Fortran 2003 feature that some compilers may not support. For those compilers, we must explicitly destroy the objects, otherwise there will be memory leaks. This applies to many other derived types.

amrex_box is a derived type in amrex_box_module amrex/Src/F_Interfaces/Base/AMReX_box_mod.F90. It has three members, lo (lower corner), hi (upper corner) and nodal (logical flag for index type).

amrex_geometry is a wrapper for the Geometry class containing information for the physical domain. Below is an example of building it.

```
integer :: n_cell
type(amrex_box) :: domain
type(amrex_geometry) : geom
! n_cell = ...
! Define a single box covering the domain
domain = amrex_box((/0,0,0/), (/n_cell-1, n_cell-1, n_cell-1/))
! This defines a amrex_geometry object.
call amrex_geometry_build(geom, domain)
!
! ...
!
call amrex_geometry_destroy(geom)
```

amrex_boxarray (amrex/Src/F_Interfaces/Base/AMReX_boxarray_mod.F90) is a wrapper for the BoxArray class, and amrex_distromap (amrex/Src/F_Interfaces/Base/AMReX_distromap_mod.F90) is a wrapper for the DistributionMapping class. Here is an example of building a BoxArray and a DistributionMapping.

```
integer :: n_cell
type(amrex_box) :: domain
```

(continues on next page)

(continued from previous page)

```
type(amrex_boxarray) : ba
type(amrex_distromap) :: dm
! n_cell = ...
! Define a single box covering the domain
domain = amrex_box((/0,0,0/), (/n_cell-1, n_cell-1, n_cell-1/))
! Initialize the boxarray "ba" from the single box "bx"
call amrex_boxarray_build(ba, domain)
! Break up boxarray "ba" into chunks no larger than "max_grid_size"
call ba%maxSize(max_grid_size)
! Build a DistributionMapping for the boxarray
call amrex_distromap_build(dm, ba)
!
! ...
!
call amrex_distromap_distromap(dm)
call amrex_boxarray_destroy(ba)
```

Given `amrex_boxarray` and `amrex_distromap`, we can build `amrex_multifab`, a wrapper for the `MultiFab` class, as follows.

```
integer :: ncomp, nghost
type(amrex_boxarray) : ba
type(amrex_distromap) :: dm
type(amrex_multifab) :: mf, ndmf
! Build amrex_boxarray and amrex_distromap
! ncomp = ...
! nghost = ...
! ...
! Build amrex_multifab with ncomp component and nghost ghost cells
call amrex_multifab_build(mf, ba, dm, ncomp, nghost)
! Build a nodal multifab
call amrex_multifab_build(ndmf,ba,dm,ncomp,nghost,(/.true.,.true.,.true./))
!
! ...
!
call amrex_multifab_destroy(mf)
call amrex_multifab_destroy(ndmf)
```

There are many type-bound procedures for `amrex_multifab`. For example

```
ncomp   ! Return the number of components
nghost  ! Return the number of ghost cells
setval  ! Set the data to the given value
copy    ! Copy data from given amrex_multifab to this amrex_multifab
```

Note that the copy function here only works on copying data from another `amrex_multifab` built with the same `amrex_distromap`, like the `MultiFab::Copy` function in C++. `amrex_multifab` also has two parallel communication procedures, `fill_boundary` and `parallel_copy`. Their and interface and usage are very similar to functions `FillBoundary` and `ParallelCopy` for `MultiFab` in C++.

```
type(amrex_geometry) :: geom
type(amrex_multifab) :: mf, mfsrc
! ...
```

(continues on next page)

(continued from previous page)

```
call mf%fill_boundary(geom)       ! Fill all components
call mf%fill_boundary(geom, 1, 3) ! Fill 3 components starting with component 1

call mf%parallel_copy(mfsrc, geom) ! Parallel copy from another multifab
```

It should be emphasized that the component index for amrex_multifab starts with 1 following Fortran convention. This is different from the C++ part of AMReX.

AMReX provides a Fortran interface to MFIter for iterating over the data in amrex_multifab. The Fortran type for this is amrex_mfiter. Here is an example of using amrex_mfiter to loop over amrex_multifab with tiling and launch a kernel function.

```
integer :: plo(4), phi(4)
type(amrex_box) :: bx
real(amrex_real), contiguous, dimension(:,:,:,:), pointer :: po, pn
type(amrex_multifab) :: old_phi, new_phi
type(amrex_mfiter) :: mfi
! Define old_phi and new_phi ...
! In this example they are built with the same boxarray and distromap.
! And they have the same number of ghost cells and 1 component.
call amrex_mfiter_build(mfi, old_phi, tiling=.true.)
do while (mfi%next())
  bx = mfi%tilebox()
  po => old_phi%dataptr(mfi)
  pn => new_phi%dataptr(mfi)
  plo = lbound(po)
  phi = ubound(po)
  call update_phi(bx%lo, bx&hi, po, pn, plo, phi)
end do
call amrex_mfiter_destroy(mfi)
```

Here procedure update_phi is

```
subroutine update_phi (lo, hi, pold, pnew, plo, phi)
 integer, intent(in) :: lo(3), hi(3), plo(3), phi(3)
  real(amrex_real),intent(in   ) pold(plo(1):phi(1),plo(2):phi(2),plo(3):phi(3))
  real(amrex_real),intent(inout) pnew(plo(1):phi(1),plo(2):phi(2),plo(3):phi(3))
  ! ...
end subroutine update_phi
```

Note that amrex_multifab's procedure dataptr takes amrex_mfiter and returns a 4-dimensional Fortran pointer. For performance, we should declare the pointer as contiguous. In C++, the similar operation returns a reference to FArrayBox. However, FArrayBox and Fortran pointer have a similar capability of containing array bound information. We can call lbound and ubound on the pointer to return its lower and upper bounds. The first three dimensions of the bounds are spatial and the fourth is for the number of component.

Many of the derived Fortran types in (e.g., amrex_multifab, amrex_boxarray, amrex_distromap, amrex_mfiter, and amrex_geometry) contain a type(c_ptr) that points a C++ object. They also contain a logical type indicating whether or not this object owns the underlying object (i.e., responsible for deleting the object). Due to the semantics of Fortran, one should not return these types with functions. Instead we should pass them as arguments to procedures (preferably with intent specified). These five types all have assignment(=) operator that performs a shallow copy. After the assignment, the original objects still owns the data and the copy is just an alias. For example,

```
type(amrex_multifab) :: mf1, mf2
call amrex_multifab_build(mf1, ...)
call amrex_multifab_build(mf2, ...)
! At this point, both mf1 and mf2 are data owners
mf2 = mf1   ! This will destroy the original data in mf2.
            ! Then mf2 becomes a shallow copy of mf1.
            ! mf1 is still the owner of the data.
call amrex_multifab_destroy(mf1)
! mf2 no longer contains a valid pointer because mf1 has been destroyed.
call amrex_multifab_destroy(mf2)  ! But we still need to destroy it.
```

If we need to transfer the ownership, `amrex_multifab`, `amrex_boxarray` and `amrex_distromap` provide type-bound `move` procedure. We can use it as follows

```
type(amrex_multifab) :: mf1, mf2
call amrex_multifab_build(mf1, ...)
call mf2%move(mf1)   ! mf2 is now the data owner and mf1 is not.
call amrex_multifab_destroy(mf1)
call amrex_multifab_destroy(mf2)
```

`amrex_multifab` also has a type-bound `swap` procedure for exchanging the data.

AMReX also provides `amrex_plotfile_module` for writing plotfiles. The interface is similar to the C++ versions.

12.3 Amr Core Infrastructure

The module `amrex_amr_module` provides interfaces to AMR core infrastructure. With AMR, the main program might look like below,

```
program main
  use amrex_amr_module
  implicit none
  call amrex_init()
  call amrex_amrcore_init()
  call my_amr_init()       ! user's own code, not part of AMReX
  ! ...
  call my_amr_finalize()   ! user's own code, not part of AMReX
  call amrex_amrcore_finalize()
  call amrex_finalize()
end program main
```

Here we need to call `amrex_amrcore_init` and `amrex_amrcore_finalize`. And usually we need to call application code specific procedures to provide some "hooks" needed by AMReX. In C++, this is achieved by using virtual functions. In Fortran, we need to call

```
subroutine amrex_init_virtual_functions (mk_lev_scrtch, mk_lev_crse, &
                                         mk_lev_re, clr_lev, err_est)

  ! Make a new level from scratch using provided boxarray and distromap
  ! Only used during initialization.
  procedure(amrex_make_level_proc)  :: mk_lev_scrtch
  ! Make a new level using provided boxarray and distromap, and fill
```

(continues on next page)

(continued from previous page)

```
  ! with interpolated coarse level data.
  procedure(amrex_make_level_proc)  :: mk_lev_crse
  ! Remake an existing level using provided boxarray and distromap,
  ! and fill with existing fine and coarse data.
  procedure(amrex_make_level_proc)  :: mk_lev_re
  ! Delete level data
  procedure(amrex_clear_level_proc) :: clr_lev
  ! Tag cells for refinement
  procedure(amrex_error_est_proc)   :: err_est
end subroutine amrex_init_virtual_functions
```

We need to provide five functions and these functions have three types of interfaces:

```
subroutine amrex_make_level_proc (lev, time, ba, dm) bind(c)
  import
  implicit none
  integer, intent(in), value :: lev
  real(amrex_real), intent(in), value :: time
  type(c_ptr), intent(in), value :: ba, dm
end subroutine amrex_make_level_proc

subroutine amrex_clear_level_proc (lev) bind(c)
  import
  implicit none
  integer, intent(in) , value :: lev
end subroutine amrex_clear_level_proc

subroutine amrex_error_est_proc (lev, tags, time, tagval, clearval) bind(c)
  import
  implicit none
  integer, intent(in), value :: lev
  type(c_ptr), intent(in), value :: tags
  real(amrex_real), intent(in), value :: time
  character(c_char), intent(in), value :: tagval, clearval
end subroutine amrex_error_est_proc
```

`amrex-tutorials/ExampleCodes/FortranInterface/Advection_F/Source/my_amr_mod.F90` shows an example of the setup process. The user provided `procedure(amrex_error_est_proc)` has a tags argument that is of type `c_ptr` and its value is a pointer to a `TagBoxArray` object. We need to convert this into a Fortran `amrex_tagboxarray` object.

```
type(amrex_tagboxarray) :: tag
tag = tags
```

The module `amrex_fillpatch_module` provides interface to C++ functions `FillPatchSinglelevel` and `FillPatchTwoLevels`. To use it, the application code needs to provide procedures for interpolation and filling physical boundaries. See `amrex-tutorials/ExampleCodes/FortranInterface/Advection_F/Source/fillpatch_mod.F90` for an example.

Module `amrex_fluxregister_module` provides interface to `FluxRegister` (see the section on *Using FluxRegisters*). Its usage is demonstrated in the tutorial at Advection_F.

12.4 Octree

In AMReX, the union of fine level grids is properly contained within the union of coarse level grids. There are no required direct parent-child connections between levels. Therefore, grids in AMReX in general cannot be represented by trees. Nevertheless, octree type grids are supported via Fortran interface, because grids are more general than octree grids. A tutorial example using amrex_octree_module (`amrex/Src/F_Interfaces/Octree/AMReX_octree_mod.f90`) is available at `amrex-tutorials/ExampleCodes/FortranInterface/Advection_F/Advection_octree_F/`. Procedures `amrex_octree_init` and `amrex_octree_finalize` must be called as follows,

```
program main
  use amrex_amrcore_module
  use amrex_octree_module
  implicit none
  call amrex_init()
  call amrex_octree_int()  ! This should be called before amrex_amrcore_init.
  call amrex_amrcore_init()
  call my_amr_init()       ! user's own code, not part of AMReX
  ! ...
  call my_amr_finalize()   ! user's own code, not part of AMReX
  call amrex_amrcore_finalize()
  call amrex_octree_finalize()
  call amrex_finalize()
end program main
```

By default, the grid size is 8^3, and this can be changed via `ParmParse` parameter `amr.max_grid_size`. The module `amrex_octree_module` provides `amrex_octree_iter` that can be used to iterate over leaves of octree. For example,

```
type(amrex_octree_iter) :: oti
type(multifab) :: phi_new(*)   ! one multifab for each level
integer :: ilev, igrd
type(amrex_box) :: bx
real(amrex_real), contiguous, pointer, dimension(:,:,:,:) :: pout
call amrex_octree_iter_build(oti)
do while(oti%next())
   ilev = oti%level()
   igrd = oti%grid_index()
   bx   = oti%box()
   pout => phi_new(ilev)%dataptr(igrd)
   ! ...
end do
call amrex_octree_iter_destroy(oti)
```

CHAPTER

THIRTEEN

EMBEDDED BOUNDARIES

For computations with complex geometries, AMReX provides data structures and algorithms to employ an embedded boundary (EB) approach to PDE discretizations. In this approach, the underlying computational mesh is uniform and block-structured, but the boundary of the irregular-shaped computational domain conceptually cuts through this mesh. Each cell in the mesh becomes labeled as regular, cut or covered, and the finite-volume based discretization methods traditionally used in AMReX applications can be modified to incorporate these cell shapes. See Fig. 13.1 for an illustration.

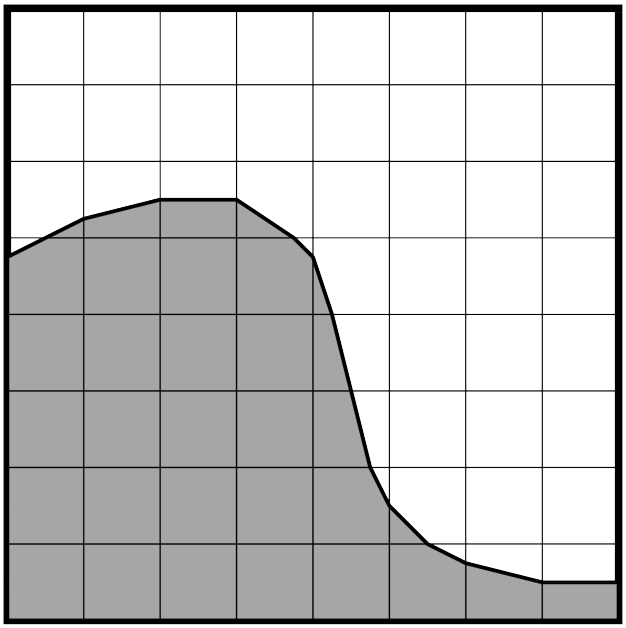


Fig. 13.1: : In the embedded boundary approach to discretizing PDEs, the (uniform) rectangular mesh is cut by the irregular shape of the computational domain. The cells in the mesh are label as regular, cut or covered.

Note that in a completely general implementation of the EB approach, there would be no restrictions on the shape or complexity of the EB surface. With this generality comes the possibility that the process of "cutting" the cells results in a single (i, j, k) cell being broken into multiple cell fragments. The current release of AMReX does not support multi-valued cells, thus there is a practical restriction on the complexity of domains (and numerical algorithms) supported.

AMReX's relatively simple grid generation technique allows computational meshes for rather complex geometries to be generated quickly and robustly. However, the technique can produce arbitrarily small cut cells in the domain. In practice such small cells can have significant impact on the robustness and stability of traditional finite volume methods. The redistribution section in AMReX-Hydro's documentation overviews the finite volume discretization in an embedded boundary cell and a class of approaches to deal with this "small cell" problem in a robust and efficient way.

This chapter discusses the EB tools, data structures and algorithms currently supported by AMReX to enable the construction of discretizations of conservation law systems. The discussion will focus on general requirements associated with building fluxes and taking divergences of them to advance such systems. We also give examples of how to initialize the geometry data structures and access them to build the numerical difference operators. Finally we present EB support of linear solvers.

13.1 Initializing the Geometric Database

In AMReX geometric information is stored in a distributed database class that must be initialized at the start of the calculation. The procedure for this goes as follows:

- Define an implicit function of position which describes the surface of the embedded object. Specifically, the function class must have a public member function that takes a position and returns a negative value if that position is inside the fluid, a positive value in the body, and identically zero at the embedded boundary.

```
Real operator() (const Array<Real,AMREX_SPACEDIM>& p) const;
```

- Make a `EB2::GeometryShop` object using the implicit function.
- Build an `EB2::IndexSpace` with the `EB2::GeometryShop` object and a `Geometry` object that contains the information about the domain and the mesh.

Here is a simple example of initialize the database for an embedded sphere.

```
Real radius = 0.5;
Array<Real,AMREX_SPACEDIM> center{0., 0., 0.}; //Center of the sphere
bool inside = false;  // Is the fluid inside the sphere?
EB2::SphereIF sphere(radius, center, inside);

auto shop = EB2::makeShop(sphere);

Geometry geom(...);
EB2::Build(shop, geom, 0, 0);
```

Alternatively, the EB information can be initialized from an STL file specified by a `ParmParse` parameter `eb2.stl_file`. The initialization is done by calling

```
EB2::Build (const Geometry& geom,
            int required_coarsening_level,
            int max_coarsening_level,
            int ngrow = 4,
            bool build_coarse_level_by_coarsening = true);
```

Additionally one can use `eb2.stl_scale`, `eb2.stl_center` and `eb2.stl_reverse_normal` to scale, translate and reverse the object, respectively.

13.1.1 Implicit Function

In `amrex/Src/EB/`, there are a number of predefined implicit function classes for basic shapes. One can use these directly or as template for their own classes.

- `AllRegularIF`: No embedded boundaries at all.
- `BoxIF`: Box.
- `CylinderIF`: Cylinder.
- `EllipsoidIF`: Ellipsoid.
- `PlaneIF`: Half-space plane.
- `SphereIF`: Sphere.

AMReX also provides a number of transformation operations to apply to an object.

- `makeComplement`: Complement of an object. E.g. a sphere with fluid on outside becomes a sphere with fluid inside.
- `makeIntersection`: Intersection of two or more objects.
- `makeUnion`: Union of two or more objects.
- `Translate`: Translates an object.
- `scale`: Scales an object.
- `rotate`: Rotates an object.
- `lathe`: Creates a surface of revolution by rotating a 2D object around an axis.

Here are some examples of using these functions.

```
EB2::SphereIF sphere1(...);
EB2::SphereIF sphere2(...);
EB2::BoxIF box(...);
EB2::CylinderIF cylinder(...);
EB2::PlaneIF plane(...);

// union of two spheres
auto twospheres = EB2::makeUnion(sphere1, sphere2);

// intersection of a rotated box, a plane and the union of two spheres
auto box_plane = EB2::makeIntersection(amrex::rotate(box,...),
                                       plane,
                                       twospheres);

// scale a cylinder by a factor of 2 in x and y directions, and 3 in z-direction.
auto scylinder = EB2::scale(cylinder, {2., 2., 3.});
```

13.1.2 EB2::GeometryShop

Given an implicit function object, say f, we can make a GeometryShop object with

```
auto shop = EB2::makeShop(f);
```

13.1.3 EB2::IndexSpace

We build EB2::IndexSpace with a template function

```
template <typename G>
void EB2::Build (const G& gshop, const Geometry& geom,
                 int required_coarsening_level,
                 int max_coarsening_level,
                 int ngrow = 4);
```

Here the template parameter is a EB2::GeometryShop. Geometry (see section *RealBox and Geometry*) describes the rectangular problem domain and the mesh on the finest AMR level. Coarse level EB data is generated from coarsening the original fine data. The int required_coarsening_level parameter specifies the number of coarsening levels required. This is usually set to $N-1$, where N is the total number of AMR levels. The int max_coarsening_levels parameter specifies the number of coarsening levels AMReX should try to have. This is usually set to a big number, say 20 if multigrid solvers are used. This essentially tells the build to coarsen as much as it can. If there are no multigrid solvers, the parameter should be set to the same as required_coarsening_level. It should be noted that coarsening could create multi-valued cells even if the fine level does not have any multi-valued cells. This occurs when the embedded boundary cuts a cell in such a way that there is fluid on multiple sides of the boundary within that cell. Because multi-valued cells are not supported, it will cause a runtime error if the required coarsening level generates multi-valued cells. The optional int ngrow parameter specifies the number of ghost cells outside the domain on required levels. For levels coarser than the required level, no EB data are generated for ghost cells outside the domain.

The newly built EB2::IndexSpace is pushed on to a stack. Static function EB2::IndexSpace::top() returns a const & to the new EB2::IndexSpace object. We usually only need to build one EB2::IndexSpace object. However, if your application needs multiple EB2::IndexSpace objects, you can save the pointers for later use. For simplicity, we assume there is only one *EB2::IndexSpace* object for the rest of this chapter.

13.2 EBFArrayBoxFactory

After the EB database is initialized, the next thing we build is EBFArrayBoxFactory. This object provides access to the EB database in the format of basic AMReX objects such as BaseFab, FArrayBox, FabArray, and MultiFab. We can construct it with

```
EBFArrayBoxFactory (const Geometry& a_geom,
                    const BoxArray& a_ba,
                    const DistributionMapping& a_dm,
                    const Vector<int>& a_ngrow,
                    EBSupport a_support);
```

or

```
std::unique_ptr<EBFArrayBoxFactory>
makeEBFabFactory (const Geometry& a_geom,
                  const BoxArray& a_ba,
```

(continues on next page)

(continued from previous page)

```
                const DistributionMapping& a_dm,
                const Vector<int>& a_ngrow,
                EBSupport a_support);
```

Argument `Vector<int> const& a_ngrow` specifies the number of ghost cells we need for EB data at various `EBSupport` levels, and argument `EBSupport a_support` specifies the level of support needed.

- `EBSupport:basic`: basic flags for cell types
- `EBSupport:volume`: basic plus volume fraction and centroid
- `EBSupport:full`: volume plus area fraction, boundary centroid and face centroid

`EBFArrayBoxFactory` is derived from `FabFactory<FArrayBox>`. `MultiFab` constructors have an optional argument `const FabFactory<FArrayBox>&`. We can use `EBFArrayBoxFactory` to build `MultiFab`s that carry EB data. Member function of `FabArray`

```
const FabFactory<FAB>& Factory () const;
```

can then be used to return a reference to the `EBFArrayBoxFactory` used for building the `MultiFab`. Using `dynamic_cast`, we can test whether a `MultiFab` is built with an `EBFArrayBoxFactory`.

```
auto factory = dynamic_cast<EBFArrayBoxFactory const*>(&(mf.Factory()));
if (factory) {
    // this is EBFArrayBoxFactory
} else {
    // regular FabFactory<FArrayBox>
}
```

13.3 Embedded Boundary Data

Through member functions of `EBFArrayBoxFactory`, we have access to the following data:

```
// see section on EBCellFlagFab
const FabArray<EBCellFlagFab>& getMultiEBCellFlagFab () const;

// volume fraction
const MultiFab& getVolFrac () const;

// volume centroid
const MultiCutFab& getCentroid () const;

// embedded boundary centroid
const MultiCutFab& getBndryCent () const;

// area fractions
Array<const MultiCutFab*,AMREX_SPACEDIM> getAreaFrac () const;

// face centroid
Array<const MultiCutFab*,AMREX_SPACEDIM> getFaceCent () const;
```

- **Volume fraction** is in a single-component `MultiFab`. Data are in the range of $[0, 1]$ with zero representing covered cells and one for regular cells.

- **Volume centroid** (also called cell centroid) is in a `MultiCutFab` with `AMREX_SPACEDIM` components. Each component of the data is in the range of $[-0.5, 0.5]$, based on each cell's local coordinates with respect to the regular cell's center.
- **Boundary centroid** is also in a `MultiCutFab` with `AMREX_SPACEDIM` components. Each component of the data is in the range of $[-0.5, 0.5]$, based on each cell's local coordinates with respect to the regular cell's center.
- **Face centroid** is in a `MultiCutFab` with `AMREX_SPACEDIM` components. Each component of the data is in the range of $[-0.5, 0.5]$, based on each cell's local coordinates with respect to the embedded boundary.
- **Area fractions** are returned in an `Array` of `MultiCutFab` pointers. For each direction, area fraction is for the face of that direction. Data are in the range of $[0, 1]$ with zero representing a covered face and one an un-cut face.
- **Face centroids** are returned in an `Array` of `MultiCutFab` pointers. There are two components for each direction and the ordering is always the same as the original ordering of the coordinates. For example, for y face, the component 0 is for x coordinate and 1 for z. The coordinates are in each face's local frame normalized to the range of $[-0.5, 0.5]$.

13.4 Embedded Boundary Data Structures

A `MultiCutFab` is very similar to a `MultiFab`. Its data can be accessed with subscript operator

```
const CutFab& operator[] (const MFIter& mfi) const;
```

Here `CutFab` is derived from `FArrayBox` and can be passed to Fortran just like `FArrayBox`. The difference between `MultiCutFab` and `MultiFab` is that to save memory `MultiCutFab` only has data on boxes that contain cut cells. It is an error to call `operator[]` if that box does not have cut cells. Thus the call must be in a `if` test block (see section *EBCellFlagFab*).

13.4.1 EBCellFlagFab

`EBCellFlagFab` contains information on cell types. We can use it to determine if a box contains cut cells.

```
auto const& flags = factory->getMultiEBCellFlagFab();
MultiCutFab const& centroid = factory->getCentroid();

for (MFIter mfi ...) {
    const Box& bx = mfi.tilebox();
    FabType t = flags[mfi].getType(bx);
    if (FabType::regular == t) {
        // This box is regular
    } else if (FabType::covered == t) {
        // This box is covered
    } else if (FabType::singlevalued == t) {
        // This box has cut cells
        // Getting cutfab is safe
        const auto& centroid_fab = centroid[mfi];
    }
}
```

`EBCellFlagFab` is derived from `BaseFab`. Its data are stored in an array of 32-bit integers, and can be used in C++ or passed to Fortran just like an `IArrayBox` (section *BaseFab, FArrayBox, IArrayBox, and Array4*). AMReX provides a Fortran module called `amrex_ebcellflag_module`. This module contains procedures for testing cell types and getting neighbor information. For example

```
use amrex_ebcellflag_module, only : is_regular_cell, is_single_valued_cell, is_covered_
↪cell

integer, intent(in) :: flags(...)

integer :: i,j,k

do k = ...
    do j = ...
        do i = ...
            if (is_covered_cell(flags(i,j,k))) then
                ! this is a completely covered cells
            else if (is_regular_cell(flags(i,j,k))) then
                ! this is a regular cell
            else if (is_single_valued_cell(flags(i,j,k))) then
                ! this is a cut cell
            end if
        end do
    end do
end do
```

13.5 Linear Solvers

Linear solvers for the canonical form (equation (10.1)) have been discussed in chapter *Linear Solvers*.

AMReX supports multi-level 1) cell-centered solvers with homogeneous Neumann, homogeneous Dirichlet, or inhomogeneous Dirichlet boundary conditions on the EB faces, and 2) nodal solvers with homogeneous Neumann boundary conditions, or inflow velocity conditions on the EB faces.

To use a cell-centered solver with EB, one builds a linear operator `MLEBABecLap` with `EBFArrayBoxFactory` (instead of a `MLABecLaplacian`)

```
MLEBABecLap (const Vector<Geometry>& a_geom,
             const Vector<BoxArray>& a_grids,
             const Vector<DistributionMapping>& a_dmap,
             const LPInfo& a_info,
             const Vector<EBFArrayBoxFactory const*>& a_factory);
```

The usage of this EB-specific class is essentially the same as `MLABecLaplacian`.

The default boundary condition on EB faces is homogeneous Neumann.

To set homogeneous Dirichlet boundary conditions, call

```
ml_ebabeclap->setEBHomogDirichlet(lev, coeff);
```

where coeff can be a real number (i.e. the value is the same at every cell) or is the MultiFab holding the coefficient of the gradient at each cell with an EB face.

To set inhomogeneous Dirichlet boundary conditions, call

```
ml_ebabeclap->setEBDirichlet(lev, phi_on_eb, coeff);
```

where phi_on_eb is the MultiFab holding the Dirichlet values in every cut cell, and coeff again is a real number (i.e. the value is the same at every cell) or a MultiFab holding the coefficient of the gradient at each cell with an EB face.

Currently there are options to define the face-based coefficients on face centers vs face centroids, and to interpret the solution variable as being defined on cell centers vs cell centroids.

The default is for the solution variable to be defined at cell centers; to tell the solver to interpret the solution variable as living at cell centroids, you must set

```
ml_ebabeclap->setPhiOnCentroid();
```

The default is for the face-based coefficients to be defined at face centers; to tell the that the face-based coefficients should be interpreted as living at face centroids, modify the setBCoeffs command to be

```
ml_ebabeclap->setBCoeffs(lev, beta, MLMG::Location::FaceCentroid);
```

13.6 Tutorials

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

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.

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

FOURTEEN

TIME INTEGRATION

AMReX provides a basic explicit time integrator capable of Forward Euler or both predefined and custom Runge-Kutta schemes designed to advance data on a particular AMR level by a timestep. This integrator is designed to be flexible, requiring the user to supply a right-hand side function taking a `MultiFab` of state data and filling a `MultiFab` of the corresponding right hand side data. The user simply needs to supply a C++ lambda function to implement whatever right hand side operations they need.

14.1 A Simple Time Integrator Setup

This is best shown with some sample code that sets up a time integrator and asks it to step forwards by some interval `dt`. The user needs to supply at minimum, the right-hand side function using the `TimeIntegrator::set_rhs()` function. By using the `TimeIntegrator::set_post_update()` function, a user can also supply a post update function which is called on state data immediately before evaluating the right-hand side. This post update function is a good opportunity to fill boundary conditions for Runge-Kutta stage solution data so that ghost cells are filled when the right hand side function is called on that solution data.

```
#include <AMReX_TimeIntegrator.H>

MultiFab Sborder; // MultiFab containing old-time state data and ghost cells
MultiFab Snew;    // MultiFab where we want new-time state data
Geometry geom;    // The domain (or level) geometry

// [Fill Sborder here]

// Create a time integrator that will work with
// MultiFabs with the same BoxArray, DistributionMapping,
// and number of components as the state_data MultiFab.
TimeIntegrator<MultiFab> integrator(Sborder);

// Create a RHS source function we will integrate
auto source_fun = [&](MultiFab& rhs, const MultiFab& state, const Real time){
    // User function to calculate the rhs MultiFab given the state MultiFab
    fill_rhs(rhs, state, time);
};

// Create a function to call after updating a state
auto post_update_fun = [&](MultiFab& S_data, const Real time) {
    // Call user function to update state MultiFab, e.g. fill BCs
    post_update(S_data, time, geom);
};
```

(continues on next page)

(continued from previous page)

```
// Attach the right hand side and post-update functions
// to the integrator
integrator.set_rhs(source_fun);
integrator.set_post_update(post_update_fun);

// integrate forward one step from `time` by `dt` to fill S_new
integrator.advance(Sborder, S_new, time, dt);
```

14.2 Using SUNDIALS

The AMReX Time Integration interface also supports a SUNDIALS backend that wraps both the explicit Runge-Kutta (ERK) and multirate (MRI) integration schemes in the SUNDIALS ARKODE package. To use either of them, the user needs to compile AMReX with *USE_SUNDIALS=TRUE* and use SUNDIALS v. 6.0 or later.

There are only minor changes to the code above required to use the SUNDIALS interface. The first change is that the integration datatype is now a *Vector<MultiFab>* type instead of simply *MultiFab*. The reason for introducing a *Vector<MultiFab>* in this case, is to permit integrating state data with different spatial centering (e.g. cell centered, face centered, node centered) concurrently. Shown here is sample code equivalent to the code above, suitable for the SUNDIALS explicit Runge-Kutta integrator:

```
#include <AMReX_TimeIntegrator.H>

Vector<MultiFab> Sborder; // MultiFab(s) containing old-time state data and ghost cells
Vector<MultiFab> Snew;    // MultiFab(s) where we want new-time state data
Geometry geom;    // The domain (or level) geometry

// [Fill Sborder here]

// Create a time integrator that will work with
// MultiFabs with the same BoxArray, DistributionMapping,
// and number of components as the state_data MultiFab.
TimeIntegrator<Vector<MultiFab> > integrator(Sborder);

// Create a RHS source function we will integrate
auto source_fun = [&](Vector<MultiFab>& rhs, const Vector<MultiFab>& state, const Real␣
↪time){
    // User function to calculate the rhs MultiFab given the state MultiFab
    fill_rhs(rhs, state, time);
};

// Create a function to call after updating a state
auto post_update_fun = [&](Vector<MultiFab>& S_data, const Real time) {
    // Call user function to update state MultiFab, e.g. fill BCs
    post_update(S_data, time, geom);
};

// Attach the right hand side and post-update functions
// to the integrator
integrator.set_rhs(source_fun);
```

(continues on next page)

(continued from previous page)

```
integrator.set_post_update(post_update_fun);

// integrate forward one step from `time` by `dt` to fill S_new
integrator.advance(Sborder, S_new, time, dt);
```

Afterwards, to select the ERK integrator, one needs only to add the following two input parameters at runtime:

```
integration.type = SUNDIALS
integration.sundials.strategy = ERK
```

If instead one wishes to use the SUNDIALS multirate integrator, then the user will need to use the following runtime inputs parameters:

```
integration.type = SUNDIALS
integration.sundials.strategy = MRI
```

In addition, to set up the multirate problem, the user needs to supply a fast timescale right-hand-side function in addition to the usual right hand side function (which is interpreted as the slow timescale right-hand side). The user will also need to supply the ratio of the slow timestep size to the fast timestep size, which is an integer corresponding to the number of fast timesteps the integrator will take per every slow timestep. An example code snippet would look as follows:

```
#include <AMReX_TimeIntegrator.H>

Vector<MultiFab> Sborder; // Vector of MultiFab(s) containing old-time state data and␣
↪ghost cells
Vector<MultiFab> Snew;    // Vector of MultiFab(s) where we want new-time state data
Geometry geom;    // The domain (or level) geometry

// [Fill Sborder here]

// Create a time integrator that will work with
// MultiFabs with the same BoxArray, DistributionMapping,
// and number of components as the state_data MultiFab.
TimeIntegrator<Vector<MultiFab> > integrator(Sborder);

// Create a slow timescale RHS function we will integrate
auto rhs_fun = [&](Vector<MultiFab>& rhs, const Vector<MultiFab>& state, const Real time)
↪{
    // User function to calculate the rhs MultiFab given the state MultiFab(s)
    fill_rhs(rhs, state, time);
};

// Create a fast timescale RHS function to integrate
auto rhs_fun_fast = [&](Vector<MultiFab>& rhs,
                        const Vector<MultiFab>& stage_data,
                        const Vector<MultiFab>& state, const Real time) {
     // User function to calculate the fast-timescale rhs MultiFab given
     // the state MultiFab and stage_data which holds the previously
     // accessed slow-timescale stage state data.
     fill_fast_rhs(rhs, stage_data, state, time);
};

// The post update function is called after updating state data or
```

(continues on next page)

(continued from previous page)

```
// immediately before using state data to calculate a fast or slow right hand side.
// (it is a good place to e.g. fill boundary conditions)
auto post_update_fun = [&](Vector<MultiFab>& S_data, const Real time) {
    // Call user function to update state MultiFab(s), e.g. fill BCs
    post_update(S_data, time, geom);
};

// Attach the slow and fast right hand side functions to integrator
integrator.set_rhs(rhs_fun);
integrator.set_fast_rhs(rhs_fun_fast);

// This sets the ratio of slow timestep size to fast timestep size as an integer,
// or equivalently, the number of fast timesteps per slow timestep.
integrator.set_slow_fast_timestep_ratio(2);

// Attach the post update function to the integrator
integrator.set_post_update(post_update_fun);

// integrate forward one step from `time` by `dt` to fill S_new
integrator.advance(Sborder, S_new, time, dt);
```

14.3 Picking A Time Integration Method

The user can customize which integration method they wish to use with a set of runtime parameters that allow choosing between a simple Forward Euler method or a generic explicit Runge-Kutta method. If Runge-Kutta is selected, then the user can choose which of a set of predefined Butcher Tables to use, or can choose to use a custom table and supply it manually.

When AMReX is compiled with SUNDIALS v.6 or later, the user also has an option to use the SUNDIALS ARKODE integrator as a backend for the AMReX Time Integrator class. The features of this interface evolve with the needs of our codes, so they may not yet support all SUNDIALS configurations available. If you find you need SUNDIALS options we have not implemented, please let us know.

The full set of integrator options are detailed as follows:

```
# INTEGRATION

## *** Selecting the integrator backend ***
## integration.type can take on the following string or int values:
## (without the quotation marks)
## "ForwardEuler" or "0" = Native Forward Euler Integrator
## "RungeKutta" or "1"   = Native Explicit Runge Kutta
## "SUNDIALS" or "2"     = SUNDIALS ARKODE Integrator
## for example:
integration.type = RungeKutta

## *** Parameters Needed For Native Explicit Runge-Kutta ***
#
## integration.rk.type can take the following values:
### 0 = User-specified Butcher Tableau
### 1 = Forward Euler
```

(continues on next page)

(continued from previous page)

```
### 2 = Trapezoid Method
### 3 = SSPRK3 Method
### 4 = RK4 Method
integration.rk.type = 3

## If using a user-specified Butcher Tableau, then
## set nodes, weights, and table entries here:
#
## The Butcher Tableau is read as a flattened,
## lower triangular matrix (but including the diagonal)
## in row major format.
integration.rk.weights = 1
integration.rk.nodes = 0
integration.rk.tableau = 0.0

## *** Parameters Needed For SUNDIALS ARKODE Integrator ***
## integration.sundials.strategy specifies which ARKODE strategy to use.
## The available options are (without the quoatations):
## "ERK" = Explicit Runge Kutta
## "MRI" = Multirate Integrator
## "MRITEST" = Tests the Multirate Integrator by setting a zero-valued fast RHS function
## for example:
integration.sundials.strategy = ERK

## *** Parameters Specific to SUNDIALS ERK Strategy ***
## (Requires integration.type=SUNDIALS and integration.sundials.strategy=ERK)
## integration.sundials.erk.method specifies which explicit Runge Kutta method
## for SUNDIALS to use. The following options are supported:
## "SSPRK3" = 3rd order strong stability preserving RK (default)
## "Trapezoid" = 2nd order trapezoidal rule
## "ForwardEuler" = 1st order forward euler
## for example:
integration.sundials.erk.method = SSPRK3

## *** Parameters Specific to SUNDIALS MRI Strategy ***
## (Requires integration.type=SUNDIALS and integration.sundials.strategy=MRI)
## integration.sundials.mri.implicit_inner specifies whether or not to use an implicit␣
↪inner solve
## integration.sundials.mri.outer_method specifies which outer (slow) method to use
## integration.sundials.mri.inner_method specifies which inner (fast) method to use
## The following options are supported for both the inner and outer methods:
## "KnothWolke3" = 3rd order Knoth-Wolke method (default for outer method)
## "Trapezoid" = 2nd order trapezoidal rule
## "ForwardEuler" = 1st order forward euler (default for inner method)
## for example:
integration.sundials.mri.implicit_inner = false
integration.sundials.mri.outer_method = KnothWolke3
integration.sundials.mri.inner_method = Trapezoid
```

CHAPTER

FIFTEEN

GPU

In this chapter, we will present the GPU support in AMReX. AMReX targets NVIDIA, AMD and Intel GPUs using their native vendor language and therefore requires CUDA, HIP/ROCm and DPC++/SYCL, for NVIDIA, AMD and Intel GPUs, respectively. Users can also use OpenMP and/or OpenACC in their applications.

AMReX supports NVIDIA GPUs with compute capability >= 6 and CUDA >= 10. While HIP and DPC++ compilers are in development in preparation for Frontier and Aurora, AMReX only supports the latest publicly released versions of those compilers on the Iris and Tulip testbeds.

For complete details of CUDA, HIP, DPC++, OpenMP and OpenACC languages, see their respective documentations.

Be aware, this documentation is currently focused on CUDA. HIP and DPC++ documentation is forthcoming.

A number of tutorials can be found at Tutorials/GPU.

15.1 Overview of AMReX GPU Strategy

AMReX's GPU strategy focuses on providing performant GPU support with minimal changes and maximum flexibility. This allows application teams to get running on GPUs quickly while allowing long term performance tuning and programming model selection. AMReX uses the native programming language for GPUs: CUDA for NVIDIA, HIP for AMD and DPC++ for Intel. This will be designated with `CUDA/HIP/DPC++` throughout the documentation. However, application teams can also use OpenACC or OpenMP in their individual codes.

At this time, AMReX does not support cross-native language compilation (HIP for non-AMD systems and DPC++ for non Intel systems). It may work with a given version, but AMReX does not track or guarantee such functionality.

When running AMReX on a CPU system, the parallelization strategy is a combination of MPI and OpenMP using tiling, as detailed in *MFIter with Tiling*. However, tiling is ineffective on GPUs due to the overhead associated with kernel launching. Instead, efficient use of the GPU's resources is the primary concern. Improving resource efficiency allows a larger percentage of GPU threads to work simultaneously, increasing effective parallelism and decreasing the time to solution.

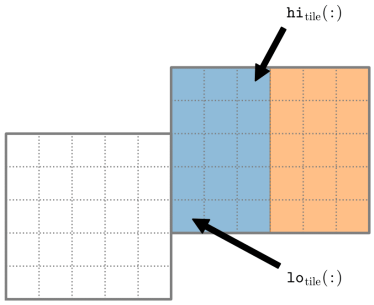
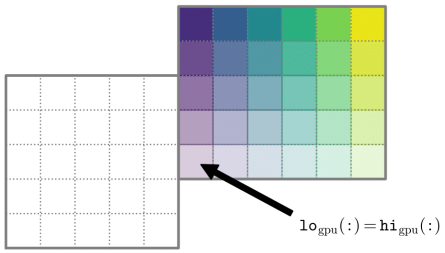
When running on CPUs, AMReX uses an `MPI+X` strategy where the `X` threads are used to perform parallelization techniques, like tiling. The most common `X` is `OpenMP`. On GPUs, AMReX requires `CUDA/HIP/DPC++` and can be further combined with other parallel GPU languages, including `OpenACC` and `OpenMP`, to control the offloading of subroutines to the GPU. This `MPI+CUDA+X` GPU strategy has been developed to give users the maximum flexibility to find the best combination of portability, readability and performance for their applications.

Presented here is an overview of important features of AMReX's GPU strategy. Additional information that is required for creating GPU applications is detailed throughout the rest of this chapter:

- Each MPI rank offloads its work to a single GPU. `(MPI ranks == Number of GPUs)`

- Calculations that can be offloaded efficiently to GPUs use GPU threads to parallelize over a valid box at a time. This is done by launching over a large number GPU threads that only work on a few cells each. This work distribution is illustrated in Table 15.1.

Table 15.1: Comparison of OpenMP and GPU work distribution. Pictures provided by Mike Zingale and the CASTRO team.

	
OpenMP tiled box. OpenMP threads break down the valid box into two large boxes (blue and orange). The lo and hi of one tiled box are marked.	GPU threaded box. Each GPU thread works on a few cells of the valid box. This example uses one cell per thread, each thread using a box with lo = hi.

- C++ macros and GPU extended lambdas are used to provide performance portability while making the code as understandable as possible to science-focused code teams.
- AMReX utilizes GPU managed memory to automatically handle memory movement for mesh and particle data. Simple data structures, such as `IntVect`s can be passed by value and complex data structures, such as `FArrayBox`es, have specialized AMReX classes to handle the data movement for the user. Tests have shown CUDA managed memory to be efficient and reliable, especially when applications remove any unnecessary data accesses.
- Application teams should strive to keep mesh and particle data structures on the GPU for as long as possible, minimizing movement back to the CPU. This strategy lends itself to AMReX applications readily; the mesh and particle data can stay on the GPU for most subroutines except for of redistribution, communication and I/O operations.
- AMReX's GPU strategy is focused on launching GPU kernels inside AMReX's `MFIter` and `ParIter` loops. By performing GPU work within `MFIter` and `ParIter` loops, GPU work is isolated to independent data sets on well-established AMReX data objects, providing consistency and safety that also matches AMReX's coding methodology. Similar tools are also available for launching work outside of AMReX loops.
- AMReX further parallelizes GPU applications by utilizing streams. Streams guarantee execution order of kernels within the same stream, while allowing different streams to run simultaneously. AMReX places each iteration of `MFIter` loops on separate streams, allowing each independent iteration to be run simultaneously and sequentially, while maximizing GPU usage.

  The AMReX implementation of streams is illustrated in Fig. 15.1. The CPU runs the first iteration of the MFIter loop (blue), which contains three GPU kernels. The kernels begin immediately in GPU Stream 1 and run in the same order they were added. The second (red) and third (green) iterations are similarly launched in Streams 2 and 3. The fourth (orange) and fifth (purple) iterations require more GPU resources than remain, so they have to wait until resources are freed before beginning. Meanwhile, after all the loop iterations are launched, the CPU reaches a synchronize in the MFIter's destructor and waits for all GPU launches to complete before continuing.
- The Fortran interface of AMReX does not currently have GPU support. AMReX recommends porting Fortran code to C++ when coding for GPUs.

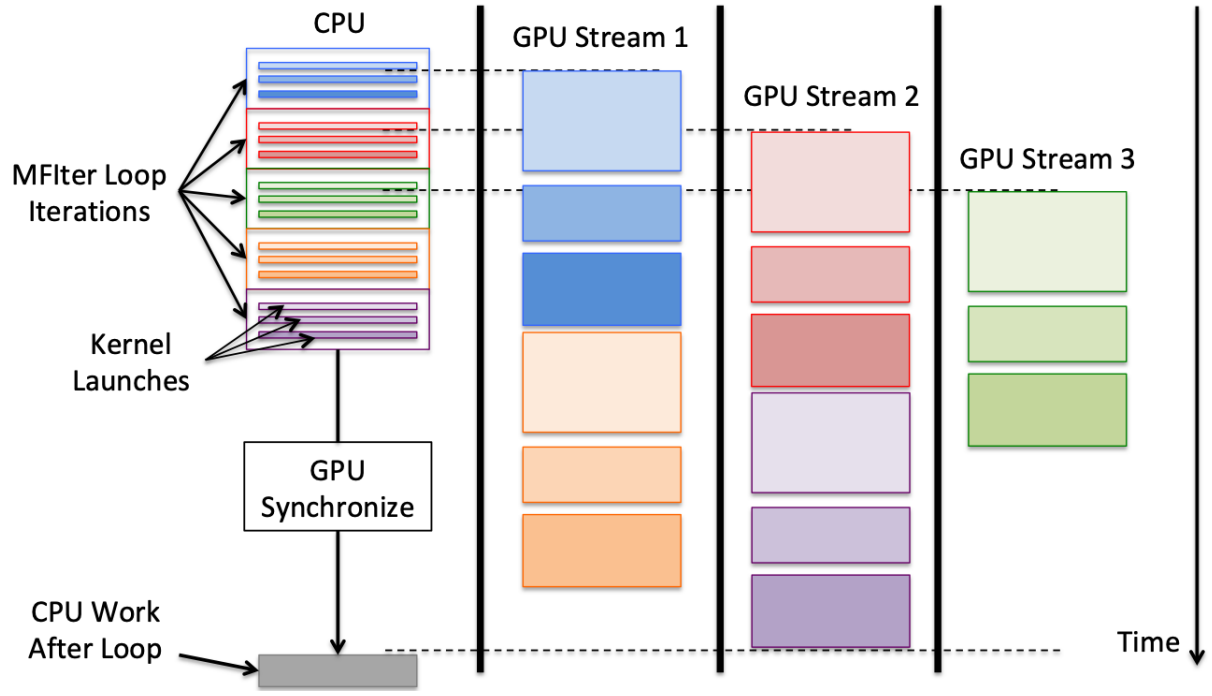


Fig. 15.1: Timeline illustration of GPU streams. Illustrates the case of an MFIter loop of five iterations with three GPU kernels each being ran with three GPU streams.

15.2 Building GPU Support

15.2.1 Building with GNU Make

To build AMReX with GPU support, add `USE_CUDA=TRUE`, `USE_HIP=TRUE` or `USE_DPCPP=TRUE` to the `GNUmakefile` or as a command line argument.

AMReX does not require OpenACC, but application codes can use them if they are supported by the compiler. For OpenACC support, add `USE_ACC=TRUE`. PGI, Cray and GNU compilers support OpenACC. Thus, for OpenACC, you must use `COMP=pgi`, `COMP=cray` or `COMP=gnu`.

Currently, only IBM is supported with OpenMP offloading. To use OpenMP offloading, make with `USE_OMP_OFFLOAD=TRUE`.

Compiling AMReX with CUDA requires compiling the code through NVIDIA's CUDA compiler driver in addition to the standard compiler. This driver is called `nvcc` and it requires a host compiler to work through. The default host compiler for NVCC is GCC even if `COMP` is set to a different compiler. One can change this by setting `NVCC_HOST_COMP`. For example, `COMP=pgi` alone will compile C/C++ codes with NVCC/GCC and Fortran codes with PGI, and link with PGI. Using `COMP=pgi` and `NVCC_HOST_COMP=pgi` will compile C/C++ codes with PGI and NVCC/PGI.

You can use `amrex-tutorials/ExampleCodes/Basic/HelloWorld_C/` to test your programming environment. For example, building with:

```
make COMP=gnu USE_CUDA=TRUE
```

should produce an executable named `main3d.gnu.DEBUG.CUDA.ex`. You can run it and that will generate results like:

```
$ ./main3d.gnu.DEBUG.CUDA.ex
Initializing CUDA...
CUDA initialized with 1 GPU
AMReX (19.06-404-g0455b168b69c-dirty) initialized
Hello world from AMReX version 19.06-404-g0455b168b69c-dirty
Total GPU global memory (MB): 6069
Free  GPU global memory (MB): 5896
[The         Arena] space (MB): 4552
[The Managed Arena] space (MB): 8
[The  Pinned Arena] space (MB): 8
AMReX (19.06-404-g0455b168b69c-dirty) finalized
```

15.2.2 Building with CMake

To build AMReX with GPU support in CMake, add `-DAMReX_GPU_BACKEND=CUDA|HIP|SYCL` to the `cmake` invocation, for CUDA, HIP and SYCL, respectively. By default, AMReX uses 256 threads per GPU block/group in most situations. This can be changed with `-DAMReX_GPU_MAX_THREADS=N`, where `N` is 128 for example.

Enabling CUDA support

To build AMReX with CUDA support in CMake, add `-DAMReX_GPU_BACKEND=CUDA` to the `cmake` invocation. For a full list of CUDA-specific configuration options, check the *table* below.

Table 15.2: AMReX CUDA-specific build options

Variable Name	Description	Default	Possible values
AMReX_CUDA_ARCH	CUDA target architecture	Auto	User-defined
AMReX_CUDA_FASTMATH	Enable CUDA fastmath library	YES	YES, NO
AMReX_CUDA_BACKTRACE	Host function symbol names (e.g. cuda-memcheck)	Auto	YES, NO
AMReX_CUDA_COMPILATION_TIMER	CSV table with time for each compilation phase	NO	YES, NO
AMReX_CUDA_DEBUG	Device debug information (optimizations: off)	YES: Debug	YES, NO
AMReX_CUDA_ERROR_CAPTURE_THIS	Error if a CUDA lambda captures a class' this	NO	YES, NO
AMReX_CUDA_ERROR_CROSS _EXECUTION_SPACE_CALL	Error if a host function is called from a host device function	NO	YES, NO
AMReX_CUDA_KEEP_FILES	Keep intermediately files (folder: nvcc_tmp)	NO	YES, NO
AMReX_CUDA_LTO	Enable CUDA link-time-optimization	NO	YES, NO
AMReX_CUDA_MAXREGCOUNT	Limits the number of CUDA registers available	255	User-defined
AMReX_CUDA_PTX_VERBOSE	Verbose code generation statistics in ptxas	NO	YES, NO
AMReX_CUDA_SHOW_CODELINES	Source information in PTX (optimizations: on)	Auto	YES, NO
AMReX_CUDA_SHOW_LINENUMBERS	Line-number information (optimizations: on)	Auto	YES, NO
AMReX_CUDA_WARN_CAPTURE_THIS	Warn if a CUDA lambda captures a class' this	YES	YES, NO

The target architecture to build for can be specified via the configuration option `-DAMReX_CUDA_ARCH=<target-architecture>`, where `<target-architecture>` can be either the name of the NVIDIA GPU generation, i.e. `Turing`, `Volta`, `Ampere`, `...` , or its compute capability, i.e. `10.0`, `9.0`, `...` . For example, on Cori GPUs you can specify the architecture as follows:

```
cmake [options] -DAMReX_GPU_BACKEND=CUDA -DAMReX_CUDA_ARCH=Volta /path/to/amrex/source
```

If no architecture is specified, CMake will default to the architecture defined in the *environment variable* `AMREX_CUDA_ARCH` (note: all caps). If the latter is not defined, CMake will try to determine which GPU architecture is supported by the system. If more than one is found, CMake will build for all of them. If autodetection fails, a list of "common" architectures is assumed. Multiple CUDA architectures can also be set manually as semicolon-separated list, e.g. `-DAMReX_CUDA_ARCH=7.0;8.0`. Building for multiple CUDA architectures will generally result in a larger library and longer build times.

Note that AMReX supports NVIDIA GPU architectures with compute capability 6.0 or higher and CUDA Toolkit version 9.0 or higher.

In order to import the CUDA-enabled AMReX library into your CMake project, you need to include the following code into the appropriate CMakeLists.txt file:

```
# Find CUDA-enabled AMReX installation
find_package(AMReX REQUIRED CUDA)
```

If instead of using an external installation of AMReX you prefer to include AMReX as a subproject in your CMake

setup, we strongly encourage you to use the AMReX_SetupCUDA module as shown below if the CMake version is less than 3.20:

```
# Enable CUDA in your CMake project
enable_language(CUDA)

# Include the AMReX-provided CUDA setup module -- OBSOLETE with CMake >= 3.20
if(CMAKE_VERSION VERSION_LESS 3.20)
    include(AMReX_SetupCUDA)
endif()

# Include AMReX source directory ONLY AFTER the two steps above
add_subdirectory(/path/to/amrex/source/dir)
```

To ensure consistency between CUDA-enabled AMReX and any CMake target that links against it, we provide the helper function setup_target_for_cuda_compilation():

```
# Set all sources for my_target
target_sources(my_target source1 source2 source3 ...)

# Setup my_target to be compiled with CUDA and be linked against CUDA-enabled AMReX
# MUST be done AFTER all sources have been assigned to my_target
setup_target_for_cuda_compilation(my_target)

# Link against amrex
target_link_libraries(my_target PUBLIC AMReX::amrex)
```

Enabling HIP Support

To build AMReX with HIP support in CMake, add -DAMReX_GPU_BACKEND=HIP -DAMReX_AMD_ARCH=<target-arch> -DCMAKE_CXX_COMPILER=<your-hip-compiler> to the cmake invocation. If you don't need Fortran features (AMReX_FORTRAN=OFF), it is recommended to use AMD's clang++ as the HIP compiler. (Please see these issues for reference in rocm/HIP <= 4.2.0 [1] [2].)

In AMReX CMake, the HIP compiler is treated as a special C++ compiler and therefore the standard CMake variables used to customize the compilation process for C++, for example CMAKE_CXX_FLAGS, can be used for HIP as well.

Since CMake does not support autodetection of HIP compilers/target architectures yet, CMAKE_CXX_COMPILER must be set to a valid HIP compiler, i.e. clang++ or hipcc, and AMReX_AMD_ARCH to the target architecture you are building for. Thus **AMReX_AMD_ARCH and CMAKE_CXX_COMPILER are required user-inputs when AMReX_GPU_BACKEND=HIP**. We again read also an *environment variable*: AMREX_AMD_ARCH (note: all caps) and the C++ compiler can be hinted as always, e.g. with export CXX=$(which clang++). Below is an example configuration for HIP on Tulip:

```
cmake -S . -B build -DAMReX_GPU_BACKEND=HIP -DCMAKE_CXX_COMPILER=$(which clang++) -
↪DAMReX_AMD_ARCH="gfx906;gfx908"  # [other options]
cmake --build build -j 6
```

Enabling SYCL Support

To build AMReX with SYCL support in CMake, add `-DAMReX_GPU_BACKEND=SYCL -DCMAKE_CXX_COMPILER=<your-sycl-compiler>` to the `cmake` invocation. For a full list of SYCL-specific configuration options, check the *table* below.

In AMReX CMake, the SYCL compiler is treated as a special C++ compiler and therefore the standard CMake variables used to customize the compilation process for C++, for example `CMAKE_CXX_FLAGS`, can be used for DPCPP as well.

Since CMake does not support autodetection of SYCL compilers yet, `CMAKE_CXX_COMPILER` must be set to a valid SYCL compiler. i.e. `icpx`. Thus **CMAKE_CXX_COMPILER is a required user-input when AMReX_GPU_BACKEND=SYCL**. At this time, **the only supported SYCL compiler is icpx**. Below is an example configuration for SYCL:

```
cmake -DAMReX_GPU_BACKEND=SYCL -DCMAKE_CXX_COMPILER=$(which icpx)  [other options] /path/
↪to/amrex/source
```

Table 15.3: AMReX SYCL-specific build options

Variable Name	Description	Default	Possible values
AMReX_DPCPP_AOT	Enable DPCPP ahead-of-time compilation	NO	YES, NO
AMREX_INTEL_ARCH	Specify target if AOT is enabled	•	Gen9, etc.
AMReX_DPCPP_SPLIT_KERNEL	Enable DPCPP kernel splitting	YES	YES, NO
AMReX_DPCPP_ONEDPL	Enable DPCPP's oneDPL algorithms	NO	YES, NO

15.3 Gpu Namespace and Macros

Most GPU related classes and functions are in `namespace Gpu`, which is inside `namespace amrex`. For example, the GPU configuration class `Device` can be referenced to at `amrex::Gpu::Device`.

For portability, AMReX defines some macros for CUDA function qualifiers and they should be preferred to allow execution with `USE_CUDA=FALSE`. These include:

```
#define AMREX_GPU_HOST        __host__
#define AMREX_GPU_DEVICE      __device__
#define AMREX_GPU_GLOBAL      __global__
#define AMREX_GPU_HOST_DEVICE __host__ __device__
```

Note that when AMReX is not built with `CUDA/HIP/DPC++`, these macros expand to empty space.

When AMReX is compiled with `USE_CUDA=TRUE`, the preprocessor macros `AMREX_USE_CUDA` and `AMREX_USE_GPU` are defined for conditional programming. When AMReX is compiled with `USE_ACC=TRUE`, `AMREX_USE_ACC` is defined. When AMReX is compiled with `USE_OMP_OFFLOAD=TRUE`, `AMREX_USE_OMP_OFFLOAD` is defined.

In addition to AMReX's preprocessor macros, CUDA provides the `__CUDA_ARCH__` macro which is only defined when in device code. `__CUDA_ARCH__` should be used when a `__host__ __device__` function requires separate code for the CPU and GPU implementations.

15.4 Memory Allocation

To provide portability and improve memory allocation performance, AMReX provides a number of memory pools. When compiled without GPU support, all Arenas use standard new and delete operators. With GPU support, the Arenas each allocate with a specific type of GPU memory:

Table 15.4: Memory Arenas

Arena	Memory Type
The_Arena()	managed or device memory
The_Device_Arena()	device memory, could be an alias to The_Arena()
The_Managed_Arena()	managed memory, could be an alias to The_Arena()
The_Pinned_Arena()	pinned memory

The Arena object returned by these calls provides access to two functions:

```
void* alloc (std::size_t sz);
void free (void* p);
```

The_Arena() is used for memory allocation of data in BaseFab. By default, it allocates managed memory. This can be changed with a boolean runtime parameter amrex.the_arena_is_managed. Therefore the data in a MultiFab is placed in managed memory by default and is accessible from both CPU host and GPU device. This allows application codes to develop their GPU capability gradually. The behavior of The_Managed_Arena() likewise depends on the amrex.the_arena_is_managed parameter. If amrex.the_arena_is_managed=0, The_Managed_Arena() is a separate pool of managed memory. If amrex.the_arena_is_managed=1, The_Managed_Arena() is simply aliased to The_Arena() to reduce memory fragmentation.

In amrex::Initialize, a large amount of GPU device memory is allocated and is kept in The_Arena(). The default is 3/4 of the total device memory, and it can be changed with a ParmParse parameter, amrex.the_arena_init_size, in the unit of bytes. The default initial size for other arenas is 8388608 (i.e., 8 MB). For The_Managed_Arena() and The_Device_Arena(), it can be changed with amrex.the_managed_arena_init_size and amrex.the_device_arena_init_size, respectively, if they are not an alias to The_Arena(). For The_Pinned_Arena(), it can be changed with amrex.the_pinned_arena_init_size. The user can also specify a release threshold for these arenas. If the memory usage in an arena is below the threshold, the arena will keep the memory for later reuse, otherwise it will try to release memory back to the system if it is not being used. By default, the release threshold for The_Arena() is set to be a huge number that prevents the memory being released automatically, and it can be changed with a parameter, amrex.the_arena_release_threshold. For The_Pinned_Arena(), the default release threshold is the size of the total device memory, and the runtime parameter is amrex.the_pinned_arena_release_threshold. If it is a separate arena, the behavior of The_Device_Area() or The_Managed_Arena() can be changed with amrex.the_device_arena_release_threshold or amrex.the_managed_arena_release_threshold. Note that the units for all the parameter discussed above are bytes. All these areans also have a member function freeUnused() that can be used to manually release unused memory back to the system.

If you want to print out the current memory usage of the Arenas, you can call amrex::Arena::PrintUsage(). When AMReX is built with SUNDIALS turned on, amrex::sundials::The_SUNMemory_Helper() can be provided to SUNDIALS data structures so that they use the appropriate Arena object when allocating memory. For example, it can be provided to the SUNDIALS CUDA vector:

```
N_Vector x = N_VNewWithMemHelp_Cuda(size, use_managed_memory, *The_SUNMemory_Helper());
```

15.5 GPU Safe Classes and Functions

AMReX GPU work takes place inside of MFIter and particle loops. Therefore, there are two ways classes and functions have been modified to interact with the GPU:

1. A number of functions used within these loops are labelled using `AMREX_GPU_HOST_DEVICE` and can be called on the device. This includes member functions, such as `IntVect::type()`, as well as non-member functions, such as `amrex::min` and `amrex::max`. In specialized cases, classes are labeled such that the object can be constructed, destructed and its functions can be implemented on the device, including `IntVect`.

2. Functions that contain MFIter or particle loops have been rewritten to contain device launches. For example, the `FillBoundary` function cannot be called from device code, but calling it from CPU will launch GPU kernels if AMReX is compiled with GPU support.

Necessary and convenient AMReX functions and objects have been given a device version and/or device access.

In this section, we discuss some examples of AMReX device classes and functions that are important for programming GPUs.

15.5.1 GpuArray, Array1D, Array2D, and Array3D

`GpuArray`, `Array1D`, `Array2D`, and `Array3D` are trivial types that work on both host and device. They can be used whenever a fixed size array needs to be passed to the GPU or created on GPU. A variety of functions in AMReX return `GpuArray` and they can be lambda-captured to GPU code. For example, `GeometryData::CellSizeArray()`, `GeometryData::InvCellSizeArray()` and `Box::length3d()` all return `GpuArrays`.

15.5.2 AsyncArray

Where the `GpuArray` is a statically-sized array designed to be passed by value onto the device, `AsyncArray` is a dynamically-sized array container designed to work between the CPU and GPU. `AsyncArray` stores a CPU pointer and a GPU pointer and coordinates the movement of an array of objects between the two. It can take initial values from the host and move them to the device. It can copy the data from device back to host. It can also be used as scratch space on device.

The call to delete the memory is added to the GPU stream as a callback function in the destructor of `AsyncArray`. This guarantees the memory allocated in `AsyncArray` continues to exist after the `AsyncArray` object is deleted when going out of scope until after all GPU kernels in the stream are completed without forcing the code to synchronize. The resulting `AsyncArray` class is "async-safe", meaning it can be safely used in asynchronous code regions that contain both CPU work and GPU launches, including `MFIter` loops.

`AsyncArray` is also portable. When built without `USE_CUDA`, the object only stores and handles the CPU version of the data.

An example using `AsyncArray` is given below,

```
Real h_s = 0.0;
AsyncArray<Real> aa_s(&h_s, 1);  // Build AsyncArray of size 1
Real* d_s = aa_s.data();         // Get associated device pointer

for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    Vector<Real> h_v = a_cpu_function();
    AsyncArray<Real> aa_v1(h_v.data(), h_v.size());
    Real* d_v1 = aa_v1.data();  // A device copy of the data
```

(continues on next page)

(continued from previous page)

```
    std::size_t n = ...;
    AsyncArray<Real> aa_v2(n);  // Allocate temporary space on device
    Real* d_v2 = aa_v2.data();  // A device pointer to uninitialized data

    ... // gpu kernels using the data pointed by d_v1 and atomically
        // updating the data pointed by d_s.
        // d_v2 can be used as scratch space and for pass data
        // between kernels.

    // If needed, we can copy the data back to host using
    // AsyncArray::copyToHost(host_pointer, number_of_elements);

    // At the end of each loop the compiler inserts a call to the
    // destructor of aa_v* on cpu.  Objects aa_v* are deleted, but
    // their associated memory pointed by d_v* is not deleted
    // immediately until the gpu kernels in this loop finish.
}

aa_s.copyToHost(&h_s, 1); // Copy the value back to host
```

15.5.3 Gpu Vectors

AMReX also provides a number of dynamic vectors for use with GPU kernels. These are configured to use the different AMReX memory Arenas, as summarized below. By using the memory Arenas, we can avoid expensive allocations and deallocations when (for example) resizing vectors.

Table 15.5: Memory Arenas Associated with each Gpu Vector

Vector	Arena
DeviceVector	The_Arena()
HostVector	The_Pinned_Arena()
ManagedVector	The_Managed_Arena()

These classes behave identically to an `amrex::Vector`, (see *Vector, Array, GpuArray, Array1D, Array2D, and Array3D*), except that they can only hold "plain-old-data" objects (e.g. Reals, integers, amrex Particles, etc…). If you want a resizable vector that doesn't use a memory Arena, simply use `amrex::Vector`.

Note that, even if the data in the vector is managed and available on GPUs, the member functions of e.g. `Gpu::ManagedVector` are not. To use the data on the GPU, it is necessary to pass the underlying data pointer in to the GPU kernels. The managed data pointer can be accessed using the `data()` member function.

Be aware: resizing of dynamically allocated memory on the GPU is unsupported. All resizing of the vector should be done on the CPU, in a manner that avoids race conditions with concurrent GPU kernels.

Also note: `Gpu::ManagedVector` is not async-safe. It cannot be safely constructed inside of an MFIter loop with GPU kernels and great care should be used when accessing `Gpu::ManagedVector` data on GPUs to avoid race conditions.

15.5.4 MultiFab Reductions

AMReX provides functions for performing standard reduction operations on `MultiFabs`, including `MultiFab::sum` and `MultiFab::max`. When `USE_CUDA=TRUE`, these functions automatically implement the corresponding reductions on GPUs in an efficient manner.

Function template `ParReduce` can be used to implement user-defined reduction functions over `MultiFabs`. For example, the following function computes the sum of total kinetic energy using the data in a `MultiFab` storing the mass and momentum density.

```
Real compute_ek (MultiFab const& mf)
{
    auto const& ma = mf.const_arrays();
    return ParReduce(TypeList<ReduceOpSum>{}, TypeList<Real>{},
                     mf, IntVect(0), // zero ghost cells
            [=] AMREX_GPU_DEVICE (int box_no, int i, int j, int k)
                noexcept -> GpuTuple<Real>
            {
                Array4<Real const> const& a = ma[box_no];
                Real rho = a(i,j,k,0);
                Real rhovx = a(i,j,k,1);
                Real rhovy = a(i,j,k,2);
                Real rhovz = a(i,j,k,3);
                Real ek = (rhovx*rhovx+rhovy*rhovy+rhovz*rhovz)/(2.*rho);
                return { ek };
            });
}
```

As another example, the following function computes the max- and 1-norm of a `MultiFab` in the masked region specified by an `iMultiFab`.

```
GpuTuple<Real,Real> compute_norms (MultiFab const& mf,
                                   iMulitiFab const& mask)
{
    auto const& data_ma = mf.const_arrays();
    auto const& mask_ma = mask.const_arrays();
    return ParReduce(TypeList<ReduceOpMax,ReduceOpSum>{},
                     TypeList<Real,Real>{},
                     mf, IntVect(0), // zero ghost cells
            [=] AMREX_GPU_DEVICE (int box_no, int i, int j, int k)
                noexcept -> GpuTuple<Real,Real>
            {
                if (mask_ma[box_no](i,j,k)) {
                    Real a = amrex::Math::abs(data_ma[box_no](i,j,k));
                    return { a, a };
                } else {
                    return { 0., 0. };
                }
            });
}
```

It should be noted that the reduction result of `ParReduce` is local and it is the user's responsibility if MPI communication is needed.

15.5.5 Box, IntVect and IndexType

In AMReX, `Box`, `IntVect` and `IndexType` are classes for representing indices. These classes and most of their member functions, including constructors and destructors, have both host and device versions. They can be used freely in device code.

15.5.6 Geometry

AMReX's `Geometry` class is not a GPU safe class. However, we often need to use geometric information such as cell size and physical coordinates in GPU kernels. We can use the following member functions and pass the returned values to GPU kernels:

```
GpuArray<Real,AMREX_SPACEDIM> ProbLoArray () const noexcept;
GpuArray<Real,AMREX_SPACEDIM> ProbHiArray () const noexcept;
GpuArray<int,AMREX_SPACEDIM> isPeriodicArray () const noexcept;
GpuArray<Real,AMREX_SPACEDIM> CellSizeArray () const noexcept;
GpuArray<Real,AMREX_SPACEDIM> InvCellSizeArray () const noexcept;
```

Alternatively, we can copy the data into a GPU safe class that can be passed by value to GPU kernels. This class is called `GeometryData`, which is created by calling `Geometry::data()`. The accessor functions of `GeometryData` are identical to `Geometry`.

15.5.7 BaseFab, FArrayBox, IArrayBox

`BaseFab<T>`, `IArrayBox` and `FArrayBox` have some GPU support. They cannot be constructed in device code unless they are constructed as an alias to `Array4`. Many of their member functions can be used in device code as long as they have been constructed in device memory. Some of the device member functions include `array`, `dataPtr`, `box`, `nComp`, and `setVal`.

All `BaseFab<T>` objects in `FabArray<FAB>` are allocated in CPU memory, including `IArrayBox` and `FArrayBox`, which are derived from `BaseFab`, although the array data contained are allocated in managed memory. We cannot pass a `BaseFab` object by value because they do not have copy constructor. However, we can make an `Array4` using member function `BaseFab::array()`, and pass it by value to GPU kernels. In GPU device code, we can use `Array4` or, if necessary, we can make an alias `BaseFab` from an `Array4`. For example,

```
AMREX_GPU_HOST_DEVICE void g (FArrayBox& fab) { ... }

AMREX_GPU_HOST_DEVICE void f (Box const& bx, Array4<Real> const& a)
{
  FArrayBox fab(a,bx.ixType());
  g(fab);
}
```

15.5.8 Elixir

We often have temporary FArrayBoxes in MFIter loops. These objects go out of scope at the end of each iteration. Because of the asynchronous nature of GPU kernel execution, their destructors might get called before their data are used on GPU. Elixir can be used to extend the life of the data. For example,

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi) {
  const Box& bx = mfi.tilebox();
  FArrayBox tmp_fab(bx, numcomps);
  Elixir tmp_eli = tmp_fab.elixir();
  Array4<Real> const& tmp_arr = tmp_fab.array();

  // GPU kernels using the temporary
}
```

Without Elixir, the code above will likely cause memory errors because the temporary FArrayBox is deleted on cpu before the gpu kernels use its memory. With Elixir, the ownership of the memory is transferred to Elixir that is guaranteed to be async-safe.

15.5.9 Async Arena

CUDA 11.2 has introduced a new feature, stream-ordered CUDA memory allocator. This feature enables AMReX to solve the temporary memory allocation and deallocation issue discussed above using a memory pool. Instead of using Elixir, we can write code like below,

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi) {
  const Box& bx = mfi.tilebox();
  FArrayBox tmp_fab(bx, numcomps, The_Async_Arena());
  Array4<Real> const& tmp_arr = tmp_fab.array();

  // GPU kernels using the temporary
}
```

This is now the recommended way because it's usually more efficient than Elixir. Note that the code above works for CUDA older than 11.2, HIP and DPC++ as well, and it's equivalent to using Elixir in these cases. By default, the release threshold for the memory pool is unlimited. One can adjust it with ParmParse parameter, amrex.the_async_arena_release_threshold.

15.6 Kernel Launch

In this section, how to offload work to the GPU will be demonstrated. AMReX supports offloading work with CUDA, OpenACC, or OpenMP.

When using CUDA, AMReX provides users with portable C++ function calls or C++ macros that launch a user-defined lambda function. When compiled without CUDA, the lambda function is ran on the CPU. When compiled with CUDA, the launch function prepares and launches the lambda function on the GPU. The preparation includes calculating the appropriate number of blocks and threads, selecting the CUDA stream and defining the appropriate work chunk for each CUDA thread.

When using OpenACC or OpenMP offloading pragmas, the users add the appropriate pragmas to their work loops and functions to offload to the GPU. These work in conjunction with AMReX's internal CUDA-based memory management, described earlier, to ensure the required data is available on the GPU when the offloaded function is executed.

The available launch schema are presented here in three categories: launching nested loops over Boxes or 1D arrays, launching generic work and launching using OpenACC or OpenMP pragmas. The latest versions of the examples used in this section of the documentation can be found in the AMReX source code in the Launch tutorials. Users should also refer to Chapter *Basics* as needed for information about basic AMReX classes.

AMReX also recommends writing primary floating point operation kernels in C++ using AMReX's `Array4` object syntax. It provides a multi-dimensional array syntax, similar in appearance to Fortran, while maintaining performance. The details can be found in *Array4* and *C++ Kernel*.

15.6.1 Launching C++ nested loops

The most common AMReX work construct is a set of nested loops over the cells in a box. AMReX provides C++ functions and macro equivalents to port nested loops efficiently onto the GPU. There are 3 different nested loop GPU launches: a 4D launch for work over a box and a number of components, a 3D launch for work over a box and a 1D launch for work over a number of arbitrary elements. Each of these launches provides a performance portable set of nested loops for both CPU and GPU applications.

These loop launches should only be used when each iteration of the nested loop is independent of other iterations. Therefore, these launches have been marked with `AMREX_PRAGMA_SIMD` when using the CPU and they should only be used for `simd`-capable nested loops. Calculations that cannot vectorize should be rewritten wherever possible to allow efficient utilization of GPU hardware.

However, it is important for applications to use these launches whenever appropriate because they contain optimizations for both CPU and GPU variations of nested loops. For example, on the GPU the spatial coordinate loops are reduced to a single loop and the component loop is moved to these inner most loop. AMReX's launch functions apply the appropriate optimizations for `USE_CUDA=TRUE` and `USE_CUDA=FALSE` in a compact and readable format.

AMReX also provides a variation of the launch function that is implemented as a C++ macro. It behaves identically to the function, but hides the lambda function from the user. There are some subtle differences between the two implementations, that will be discussed. It is up to the user to select which version they would like to use. For simplicity, the function variation will be discussed throughout the rest of this documentation, however all code snippets will also include the macro variation for reference.

A 4D example of the launch function, `amrex::ParallelFor`, is given here:

```
int ncomp = mf.nComp();
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    Array4<Real> const& fab = mf.array(mfi);

    amrex::ParallelFor(bx, ncomp,
    [=] AMREX_GPU_DEVICE (int i, int j, int k, int n)
    {
        fab(i,j,k,n) += 1.;
    });

    /* MACRO VARIATION:
    /
    /   AMREX_PARALLEL_FOR_4D ( bx, ncomp, i, j, k, n,
    /   {
    /       fab(i,j,k,n) += 1.;
    /   });
    */
}
```

This code works whether it is compiled for GPUs or CPUs. `TilingIfNotGPU()` returns `false` in the GPU case to turn off tiling and maximize the amount of work given to the GPU in each launch. When tiling is off, `tilebox()` returns the `validbox()`. The `BaseFab::array()` function returns a lightweight `Array4` object that defines access to the underlying `FArrayBox` data. The `Array4`s is then captured by the C++ lambda functions defined in the launch function.

`amrex::ParallelFor()` expands into different variations of a quadruply-nested **for** loop depending dimensionality and whether it is being implemented on CPU or GPU. The best way to understand this function is to take a look at the 4D `amrex::ParallelFor` that is implemented when `USE_CUDA=FALSE`. A simplified version is reproduced here:

```
void ParallelFor (Box const& box, int ncomp, /* LAMBDA FUNCTION */)
{
    const Dim3 lo = amrex::lbound(box);
    const Dim3 hi = amrex::ubound(box);

    for (int n = 0; n < ncomp; ++n) {
        for (int z = lo.z; z <= hi.z; ++z) {
        for (int y = lo.y; y <= hi.y; ++y) {
        AMREX_PRAGMA_SIMD
        for (int x = lo.x; x <= hi.x; ++x) {
            /* LAUNCH LAMBDA FUNCTION (x,y,z,n) */
        }}}
    }
}
```

`amrex::ParallelFor` takes a `Box` and a number of components, which define the bounds of the quadruply-nested **for** loop, and a lambda function to run on each iteration of the nested loop. The lambda function takes the loop iterators as parameters, allowing the current cell to be indexed in the lambda. In addition to the loop indices, the lambda function captures any necessary objects defined in the local scope.

CUDA lambda functions can only capture by value, as the information must be able to be copied onto the device. In this example, the lambda function captures a `Array4` object, `fab`, that defines how to access the `FArrayBox`. The macro uses `fab` to increment the value of each cell within the `Box bx`. If `USE_CUDA=TRUE`, this incrementation is performed on the GPU, with GPU optimized loops.

This 4D launch can also be used to work over any sequential set of components, by passing the number of consecutive components and adding the iterator to the starting component: `fab(i,j,k,n_start+n)`.

The 3D variation of the loop launch does not include a component loop and has the syntax shown here:

```
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    Array4<Real> const& fab = mf.array(mfi);
    amrex::ParallelFor(bx,
    [=] AMREX_GPU_DEVICE (int i, int j, int k)
    {
        fab(i,j,k) += 1.;
    });

    /* MACRO VARIATION:
    /
    /   AMREX_PARALLEL_FOR_3D ( bx, i, j, k,
    /   {
    /       fab(i,j,k) += 1.;
    /   });
```

(continues on next page)

(continued from previous page)

```
    */
}
```

Finally, a 1D version is available for looping over a number of elements, such as particles. An example of a 1D function launch is given here:

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox& fab = mf[mfi];
    Real* AMREX_RESTRICT p = fab.dataPtr();
    const long nitems = fab.box().numPts() * fab.nComp();

    amrex::ParallelFor(nitems,
    [=] AMREX_GPU_DEVICE (long idx)
    {
        p[idx] += 1.;
    });

    /* MACRO VARIATION:
    /
    /   AMREX_PARALLEL_FOR_1D ( nitems, idx,
    /   {
    /       p[idx] += 1.;
    /   });
    */
}
```

Instead of passing an `Array4`, `FArrayBox::dataPtr()` is called to obtain a CUDA managed pointer to the `FArrayBox` data. This is an alternative way to access the `FArrayBox` data on the GPU. Instead of passing a `Box` to define the loop bounds, a `long` or `int` number of elements is passed to bound the single `for` loop. This construct can be used to work on any contiguous set of memory by passing the number of elements to work on and indexing the pointer to the starting element: `p[idx + 15]`.

15.6.2 GPU block size

By default, `ParallelFor` launches `AMREX_GPU_MAX_THREADS` threads per GPU block, where `AMREX_GPU_MAX_THREADS` is a compile-time constant with a default value of 256. The users can also explcitly specify the number of threads per block by `ParallelFor<MY_BLOCK_SIZE>(...)`, where `MY_BLOCK_SIZE` is a multiple of the warp size (e.g., 128). This allows the users to do performance tuning for individual kernels.

15.6.3 Launching general kernels

To launch more general work on the GPU, AMReX provides a standard launch function: `amrex::launch`. Instead of creating nested loops, this function prepares the device launch based on a `Box`, launches with an appropriate sized GPU kernel and constructs a thread `Box` that defines the work for each thread. On the CPU, the thread `Box` is set equal to the total launch `Box`, so tiling works as expected. On the GPU, the thread `Box` usually contains a single cell to allow all GPU threads to be utilized effectively.

An example of a generic function launch is shown here:

```
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    Array4<Real> const& arr = mf.array(mfi);

    amrex::launch(bx,
    [=] AMREX_GPU_DEVICE (Box const& tbx)
    {
        pluseone_array4(tbx, arr);
        FArrayBox fab(arr, tbx.ixType());
        plusone_fab(tbx, fab); // this version takes FArrayBox
    });

    /* MACRO VARIATION
    /
    /   AMREX_LAUNCH_DEVICE_LAMBDA ( bx, tbx,
    /   {
    /       plusone_array4(tbx, arr);
    /       plusone_fab(tbx, FArrayBox(arr,tbx.ixType()));
    /   });
    */
}
```

It also shows how to make a `FArrayBox` from `Array4` when needed. Note that `FarrayBox`es cannot be passed to GPU kernels directly. `TilingIfNotGPU()` returns `false` in the GPU case to turn off tiling and maximize the amount of work given to the GPU in each launch, which substantially improves performance. When tiling is off, `tilebox()` returns the `validbox()` of the `FArrayBox` for that iteration.

15.6.4 Offloading work using OpenACC or OpenMP pragmas

When using OpenACC or OpenMP with AMReX, the GPU offloading work is done with pragmas placed on the nested loops. This leaves the `MFIter` loop largely unchanged. An example GPU pragma based `MFIter` loop that calls a Fortran function is given here:

```
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    FArrayBox& fab = mf[mfi];
    plusone_acc(BL_TO_FORTRAN_BOX(tbx),
                BL_TO_FORTRAN_ANYD(fab));
}
```

The function `plusone_acc` is a CPU host function. The `FArrayBox` reference from `operator[]` is a reference to a `FArrayBox` in host memory with data that has been placed in managed CUDA memory. `BL_TO_FORTRAN_BOX` and `BL_TO_FORTRAN_ANYD` behave identically to implementations used on the CPU. These macros return the individual components of the AMReX C++ objects to allow passing to the Fortran function.

The corresponding OpenACC labelled loop in `plusone_acc` is:

```
!dat = pointer to fab's managed data

!$acc kernels deviceptr(dat)
do       k = lo(3), hi(3)
```

(continues on next page)

(continued from previous page)

```
   do    j = lo(2), hi(2)
      do i = lo(1), hi(1)
         dat(i,j,k) = dat(i,j,k) + 1.0_amrex_real
      end do
   end do
end do
!$acc end kernels
```

Since the data pointer passed to `plusone_acc` points to unified memory, OpenACC can be told the data is available on the device using the `deviceptr` construct. For further details about OpenACC programming, consult the OpenACC user's guide.

The OpenMP implementation of this loop is similar, only requiring changing the pragmas utilized to obtain the proper offloading. The OpenMP labelled version of this loop is:

```
!dat = pointer to fab's managed data

!$omp target teams distribute parallel do collapse(3) schedule(static,1) is_device_
↪ptr(dat)
do       k = lo(3), hi(3)
   do    j = lo(2), hi(2)
      do i = lo(1), hi(1)
         dat(i,j,k) = dat(i,j,k) + 1.0_amrex_real
      end do
   end do
end do
```

In this case, `is_device_ptr` is used to indicate that `dat` is available in device memory. For further details about programming with OpenMP for GPU offloading, consult the OpenMP user's guide.

15.6.5 Kernel launch details

CUDA kernel calls are asynchronous and they return before the kernel is finished on the GPU. So the `MFIter` loop finishes iterating on the CPU and is ready to move on to the next work before the actual work completes on the GPU. To guarantee consistency, there is an implicit device synchronization (a GPU barrier) in the destructor of `MFIter`. This ensures that all GPU work inside of an `MFIter` loop will complete before code outside of the loop is executed. Any CUDA kernel launches made outside of an `MFIter` loop must ensure appropriate device synchronization occurs. This can be done by calling `Gpu::streamSynchronize()`.

CUDA supports multiple streams and kernels. Kernels launched in the same stream are executed sequentially, but different streams of kernel launches may be run in parallel. For each iteration of `MFIter`, AMReX uses a different CUDA stream (up to 16 streams in total). This allows each iteration of an `MFIter` loop to run independently, but in the expected sequence, and maximize the use of GPU parallelism. However, AMReX uses the default CUDA stream outside of `MFIter` loops.

Launching kernels with AMReX's launch macros or functions implement a C++ lambda function. Lambdas functions used with CUDA have some restrictions the user must understand. First, the function enclosing the extended lambda must not have private or protected access within its parent class, otherwise the code will not compile. This can be fixed by changing the access of the enclosing function to public.

Another pitfall that must be considered: if the lambda function accesses a member of the enclosing class, the lambda function actually captures `this` pointer by value and accesses variables and functions via `this->`. If the object is not accessible on GPU, the code will not work as intended. For example,

```
class MyClass {
public:
    Box bx;
    int m;                          // Unmanaged integer created on the host.
    void f () {
        amrex::launch(bx,
        [=] AMREX_GPU_DEVICE (Box const& tbx)
        {
            printf("m = %d\n", m);   // Failed attempt to use m on the GPU.
        });
    }
};
```

The function `f` in the code above will not work unless the `MyClass` object is in unified memory. If it is undesirable to put the object into unified memory, a local copy of the information can be created for the lambda to capture. For example:

```
class MyClass {
public:
    Box bx;
    int m;
    void f () {
        int local_m = m;                  // Local temporary copy of m.
        amrex::launch(bx,
        [=] AMREX_GPU_DEVICE (Box const& tbx)
        {
            printf("m = %d\n", local_m);  // Lambda captures local_m by value.
        });
    }
};
```

C++ macros have some important limitations. For example, commas outside of a set of parentheses are interpreted by the macro, leading to errors such as:

```
AMREX_PARALLEL_FOR_3D (bx, tbx,
{
    Real a, b;   <---- Error. Macro reads "{ Real a" as a parameter
                                          and "b; }" as
                                          another.
    Real a;      <---- OK
    Real b;
});
```

One should also avoid using **`continue`** and **`return`** inside the macros because it is not an actual **`for`** loop. Users that choose to implement the macro launches should be aware of the limitations of C++ preprocessing macros to ensure GPU offloading is done properly.

Finally, AMReX's most common CPU threading strategy for GPU/CPU systems is to utilize OpenMP threads to maintain multi-threaded parallelism on work chosen to run on the host. This means OpenMP pragmas should be maintained where CPU work is performed and usually turned off where work is offloaded onto the GPU. OpenMP pragmas can be turned off using the conditional pragma and `Gpu::notInLaunchRegion()`, as shown below:

```
#ifdef AMREX_USE_OMP
#pragma omp parallel if (Gpu::notInLaunchRegion())
```

(continues on next page)

(continued from previous page)

```
#endif
```

It is generally expected that simply using OpenMP threads to launch GPU work quicker will show little improvement or even perform worse. So, this conditional statement should be added to MFIter loops that contain GPU work, unless users specifically test the performance or are designing more complex workflows that require OpenMP.

15.7 Stream and Synchronization

As mentioned in Section *Overview of AMReX GPU Strategy*, AMReX uses a number of GPU streams that are either CUDA streams or HIP streams or SYCL queues. Many GPU functions (e.g., `ParallelFor` and `Gpu::copyAsync`) are asynchronous with respect to the host. To facilitate synchronization that is sometimes necessary, AMReX provides `Gpu::streamSynchronize()` and `Gpu::streamSynchronizeAll()` to synchronize the current stream and all AMReX streams, respectively. For performance reasons, one should try to minimize the number of synchronization calls. For example,

```
// The synchronous version is NOT recommended
Gpu::copy(Gpu::deviceToHost, ....);
Gpu::copy(Gpu::deviceToHost, ....);
Gpu::copy(Gpu::deviceToHost, ....);

// NOT recommended because of unnecessary synchronization
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::streamSynchronize();
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::streamSynchronize();
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::streamSynchronize();

// recommended
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::copyAsync(Gpu::deviceToHost, ....);
Gpu::streamSynchronize();
```

In addition to stream synchronization, there is also `Gpu::synchronize()` that will perform a device wide synchronization. However, a device wide synchronization is usually too excessive and it might interfere with other libraries (e.g., MPI).

15.8 An Example of Migrating to GPU

The nature of GPU programming poses difficulties for a number of common AMReX patterns, such as the one below:

```
// Given MultiFab uin and uout
#ifdef AMREX_USE_OMP
#pragma omp parallel
#endif
{
  FArrayBox q;
  for (MFIter mfi(uin,true); mfi.isValid(); ++mfi)
```

(continues on next page)

(continued from previous page)

```
  {
    const Box& tbx = mfi.tilebox();
    const Box& gbx = amrex::grow(tbx,1);
    q.resize(gbx);

    // Do some work with uin[mfi] as input and q as output.
    // The output region is gbx;
    f1(gbx, q, uin[mfi]);

    // Then do more work with q as input and uout[mfi] as output.
    // The output region is tbx.
    f2(tbx, uout[mfi], q);
  }
}
```

There are several issues in migrating this code to GPUs that need to be addressed. First, functions `f1` and `f2` have different work regions (`tbx` and `gbx`, respectively) and there are data dependencies between the two (`q`). This makes it difficult to put them into a single GPU kernel, so two separate kernels will be launched, one for each function.

As we have discussed, AMReX uses multiple CUDA streams for launching kernels. Because `q` is used inside `MFIter` loops, multiple GPU kernels on different streams are accessing its data. This creates a race condition. One way to fix this is to move `FArrayBox q` inside the loop to make it local to each loop and use `Elixir` to make it async-safe (see Section *Elixir*). This strategy works well for GPU. However it is not optimal for OpenMP CPU threads when CUDA is not used, because of the memory allocation inside OpenMP parallel region. It turns out it is actually unnecessary to make `FArrayBox q` local to each iteration when `Elixir` is used to extend the life of its floating point data. The code below shows an example of how to rewrite the example in a performance portable way.

```
// Given MultiFab uin and uout
#ifdef AMREX_USE_OMP
#pragma omp parallel if (Gpu::notInLaunchRegion())
#endif
{
  FArrayBox q;
  for (MFIter mfi(uin,TilingIfNotGPU()); mfi.isValid(); ++mfi)
  {
    const Box& tbx = mfi.tilebox();
    const Box& gbx = amrex::grow(tbx,1);
    q.resize(gbx);
    Elixir eli = q.elixir();
    Array4<Real> const& qarr = q.array();

    Array4<Real const> const& uinarr = uin.const_array(mfi);
    Array4<Real> const& uoutarr = uout.array(mfi);

    amrex::launch(gbx,
    [=] AMREX_GPU_DEVICE (Box const& b)
    {
      f1(b, qarr, uinarr);
    });

    amrex::launch(tbx,
    [=] AMREX_GPU_DEVICE (Box const& b)
    {
```

(continues on next page)

(continued from previous page)

```
      f2(b, uoutarr, qarr);
    });
  }
}
```

15.9 Assertions and Error Checking

To help debugging, we often use `amrex::Assert` and `amrex::Abort`. These functions are GPU safe and can be used in GPU kernels. However, implementing these functions requires additional GPU registers, which will reduce overall performance. Therefore, it is preferred to implement such calls in debug mode only by wrapping the calls using `#ifdef AMREX_DEBUG`.

In CPU code, `AMREX_GPU_ERROR_CHECK()` can be called to check the health of previous GPU launches. This call looks up the return message from the most recently completed GPU launch and aborts if it was not successful. Many kernel launch macros as well as the `MFIter` destructor include a call to `AMREX_GPU_ERROR_CHECK()`. This prevents additional launches from being called if a previous launch caused an error and ensures all GPU launches within an `MFIter` loop completed successfully before continuing work.

However, due to asynchronicity, determining the source of the error can be difficult. Even if GPU kernels launched earlier in the code result in a CUDA error, the error may not be output at a nearby call to `AMREX_GPU_ERROR_CHECK()` by the CPU. When tracking down a CUDA launch error, `Gpu::synchronize()`, `Gpu::streamSynchronize()`, or `Gpu::streamSynchronizeAll()` can be used to synchronize the device, the current GPU stream, or all GPU streams, respectively, and track down the specific launch that causes the error.

15.10 Particle Support

As with `MultiFab`, particle data stored in AMReX `ParticleContainer` classes are stored in unified memory when AMReX is compiled with `USE_CUDA=TRUE`. This means that the `dataPtr` associated with particles is managed and can be passed into GPU kernels. These kernels can be launched with a variety of approaches, including Cuda C / Fortran and OpenACC. An example Fortran particle subroutine offloaded via OpenACC might look like the following:

```
subroutine push_position_boris(np, structs, uxp, uyp, uzp, gaminv, dt)

use em_particle_module, only : particle_t
use amrex_fort_module, only : amrex_real
implicit none

integer,          intent(in), value  :: np
type(particle_t), intent(inout)      :: structs(np)
real(amrex_real), intent(in)         :: uxp(np), uyp(np), uzp(np), gaminv(np)
real(amrex_real), intent(in), value  :: dt

integer                              :: ip

!$acc parallel deviceptr(structs, uxp, uyp, uzp, gaminv)
!$acc loop gang vector
do ip = 1, np
    structs(ip)%pos(1) = structs(ip)%pos(1) + uxp(ip)*gaminv(ip)*dt
    structs(ip)%pos(2) = structs(ip)%pos(2) + uyp(ip)*gaminv(ip)*dt
```

(continues on next page)

(continued from previous page)

```
    structs(ip)%pos(3) = structs(ip)%pos(3) + uzp(ip)*gaminv(ip)*dt
end do
!$acc end loop
!$acc end parallel

end subroutine push_position_boris
```

Note the use of the `!$acc parallel deviceptr` clause to specify which data has been placed in managed memory. This instructs OpenACC to treat those variables as if they already live on the device, bypassing the usual copies. For complete examples of a particle code that has been ported to GPUs using Cuda, OpenACC, and OpenMP, please see the tutorial Electromagnetic PIC.

GPU-aware implementations of many common particle operations are provided with AMReX, including neighbor list construction and traversal, particle-mesh deposition and interpolation, parallel reductions of particle data, and a set of transformation and filtering operations that are useful when operating on sets of particles. For examples of these features in use, please see `Tests/Particles/`.

Finally, the parallel communication of particle data has been ported and optimized for performance on GPU platforms. This includes `Redistribute()`, which moves particles back to the proper grids after their positions have changed, as well as `fillNeighbors()` and `updateNeighbors()`, which are used to exchange halo particles. As with `MultiFab` data, these have been designed to minimize host / device traffic as much as possible, and can take advantage of the Cuda-aware MPI implementations available on platforms such as ORNL's Summit.

15.11 Profiling with GPUs

When profiling for GPUs, AMReX recommends `nvprof`, NVIDIA's visual profiler. `nvprof` returns data on how long each kernel launch lasted on the GPU, the number of threads and registers used, the occupancy of the GPU and recommendations for improving the code. For more information on how to use `nvprof`, see NVIDIA's User's Guide as well as the help web pages of your favorite supercomputing facility that uses NVIDIA GPUs.

AMReX's internal profilers currently cannot hook into profiling information on the GPU and an efficient way to time and retrieve that information is being explored. In the meantime, AMReX's timers can be used to report some generic timers that are useful in categorizing an application.

Due to the asynchronous launching of GPU kernels, any AMReX timers inside of asynchronous regions or inside GPU kernels will not measure useful information. However, since the `MFIter` synchronizes when being destroyed, any timer wrapped around an `MFIter` loop will yield a consistent timing of the entire set of GPU launches contained within. For example:

```
BL_PROFILE_VAR("A_NAME", blp);     // Profiling start
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    // gpu works
}
BL_PROFILE_STOP(blp);              // Profiling stop
```

For now, this is the best way to profile GPU codes using `TinyProfiler`. If you require further profiling detail, use `nvprof`.

15.12 Performance Tips

Here are some helpful performance tips to keep in mind when working with AMReX for GPUs:

- To obtain the best performance when using CUDA kernel launches, all device functions called within the launch region should be inlined. Inlined functions use substantially fewer registers, freeing up GPU resources to perform other tasks. This increases parallel performance and greatly reduces runtime. Functions are written inline by putting their definitions in the `.H` file and using the `AMREX_FORCE_INLINE` AMReX macro. Examples can be found in in the Launch tutorial. For example:

```
AMREX_GPU_DEVICE
AMREX_FORCE_INLINE
void plusone_cudacpp (amrex::Box const& bx, amrex::FArrayBox& fab)
{
    ...
}
```

- Pay attention to what GPUs your job scheduler is assigning to each MPI rank. In most cases you'll achieve the best performance when a single MPI rank is assigned to each GPU, and has boxes large enough to saturate that GPU's compute capacity. While there are some cases where multiple MPI ranks per GPU can make sense (typically this would be when you have some portion of your code that is not GPU accelerated and want to have many MPI ranks to make that part faster), this is probably the minority of cases. For example, on OLCF Summit you would want to ensure that your resource sets contain one MPI rank and GPU each, using *jsrun -n N -a 1 -c 7 -g 1*, where *N* is the total number of MPI ranks/GPUs you want to use. (See the OLCF [job step viewer](https://jobstepviewer.olcf.ornl.gov/) for more information.)

  Conversely, if you choose to have multiple GPUs visible to each MPI rank, AMReX will attempt to do the best job it can assigning MPI ranks to GPUs by doing round robin assignment. This may be suboptimal because this assignment scheme would not be aware of locality benefits that come from having an MPI rank be on the same socket as the GPU it is managing. If you know the hardware layout of the system you're running on, specifically the number of GPUs per socket (*M*) and number of GPUs per node (*N*), you can set the preprocessor defines *-DAMREX_GPUS_PER_SOCKET=M* and *-DAMREX_GPUS_PER_NODE=N*, which are exposed in the GNU Make system through the variables *GPUS_PER_SOCKET* and *GPUS_PER_NODE* respectively (see an example in *Tools/GNUMake/sites/Make.olcf*). Then AMReX can ensure that each MPI rank selects a GPU on the same socket as that rank (assuming your MPI implementation supports MPI 3.)

15.13 Inputs Parameters

The following inputs parameters control the behavior of amrex when running on GPUs. They should be prefaced by "amrex" in your `inputs` file.

	Description	Type	Default
use_gpu_aware_mpi	Whether to use GPU memory for communication buffers during MPI calls. If true, the buffers will use device memory. If false, they will use pinned memory. In practice, we find it is usually not worth it to use GPU aware MPI.	Bool	False
abort_on_out_of_gpu_memory	If the size of free memory on the GPU is less than the size of a requested allocation, AMReX will call AMReX::Abort() with an error describing how much free memory there is and what was requested.	Bool	False
the_arena_is_managed	Whether `The_Arena()` allocates managed memory.	Bool	True

CHAPTER

SIXTEEN

VISUALIZATION

There are several visualization tools that can be used for AMReX plotfiles. The standard tool used within the AMReX-community is Amrvis, a package developed and supported by CCSE that is designed specifically for highly efficient visualization of block-structured hierarchical AMR data. Plotfiles can also be viewed using the VisIt, ParaView, and yt packages. Particle data can be viewed using ParaView.

16.1 Amrvis

Our favorite visualization tool is Amrvis. We heartily encourage you to build the `amrvis1d`, `amrvis2d`, and `amrvis3d` executables, and use them to visualize your data. A useful feature is `View/Dataset`, which allows you to view data in a nested spreadsheet that reflects the AMR hierarchy – this can be handy for debugging. Other display options include: the ability to select the number of levels of data to show, whether to display grid boxes, and to specify the color palette. Below are instructions and tips for using Amrvis. Additional information is contained in the document `Amrvis/Docs/Amrvis.tex` (which can built into a `pdf` using `pdflatex`).

1. **Download and Build**:

   Amrvis is available for download from the `AMReX-Codes/Amrvis` GitHub repository. To download use,

   ```
   git clone https://github.com/AMReX-Codes/Amrvis
   ```

   To build, `cd` into `Amrvis/`, and edit `GNUmakefile` by setting the variable `COMP` to your compiler suite.

   Type `make DIM=1`, `make DIM=2`, or `make DIM=3` to build. The result is an executable that looks like `amrvis2d.<ver>.ex`.

   3D Data Visualization with Volpack

   If you want to build Amrvis with `DIM=3` for display of 3-dimensional data, you must first download and build `volpack`. This can be done by cloning the repository or via package manager. To install by cloning the repository:

   ```
   git clone https://ccse.lbl.gov/pub/Downloads/volpack.git
   ```

   After downloading, `cd` into `volpack/` and type `make`.

   To install via package manager, it is necessary to install the package, `libvolpack1-dev`. This package is available for Debian Linux and can be installed with the command:

```
sudo apt install libvolpack1-dev
```

Note: Amrvis requires the OSF/Motif libraries and headers. If you don't have these you will need to install the development version of motif through your package manager. `lesstif` gives some functionality and will allow you to build the Amrvis executable, but Amrvis may exhibit subtle anomalies.

On most Linux distributions, the motif library is provided by the `openmotif` package, and its header files (like `Xm.h`) are provided by `openmotif-devel`. If those packages are not installed, then use the OS-specific package management tool to install them.

Note: These instructions assume that the install directories for Amrvis and volpack share the same parent directory. To install volpack in a different location specify the location of volpack in Amrvis's `GNUmakefile` by changing the variable `VOLPACKDIR` to the desired location.

After building you may want to create an alias for convenience. To do this type,

```
alias amrvis2d /tmp/Amrvis/amrvis2d.<ver>.ex
```

2. **Configure**:

   The settings for Amrvis are saved in the configuration file `.amrvis.defaults` in your home directory. A default version of this file is available in the parent directory of the Amrvis repo. Run the command `cp Amrvis/amrvis.defaults ~/.amrvis.defaults` to copy it to your home directory. A color pallete is also available in the Amrvis directory as a file named `Palette`. To configure Amrvis to use this pallete you can open the `.amrvis.defaults` file in your home directory and edit the line containing `palette` to point to the location of this file. For example,

   ```
   palette     ~/Amrvis/Palette
   ```

   Other lines in `.amrvis.defaults` control options such as the initial field to display, the number format, window size, etc. If there are multiple instances of the same option, the last option takes precedence.

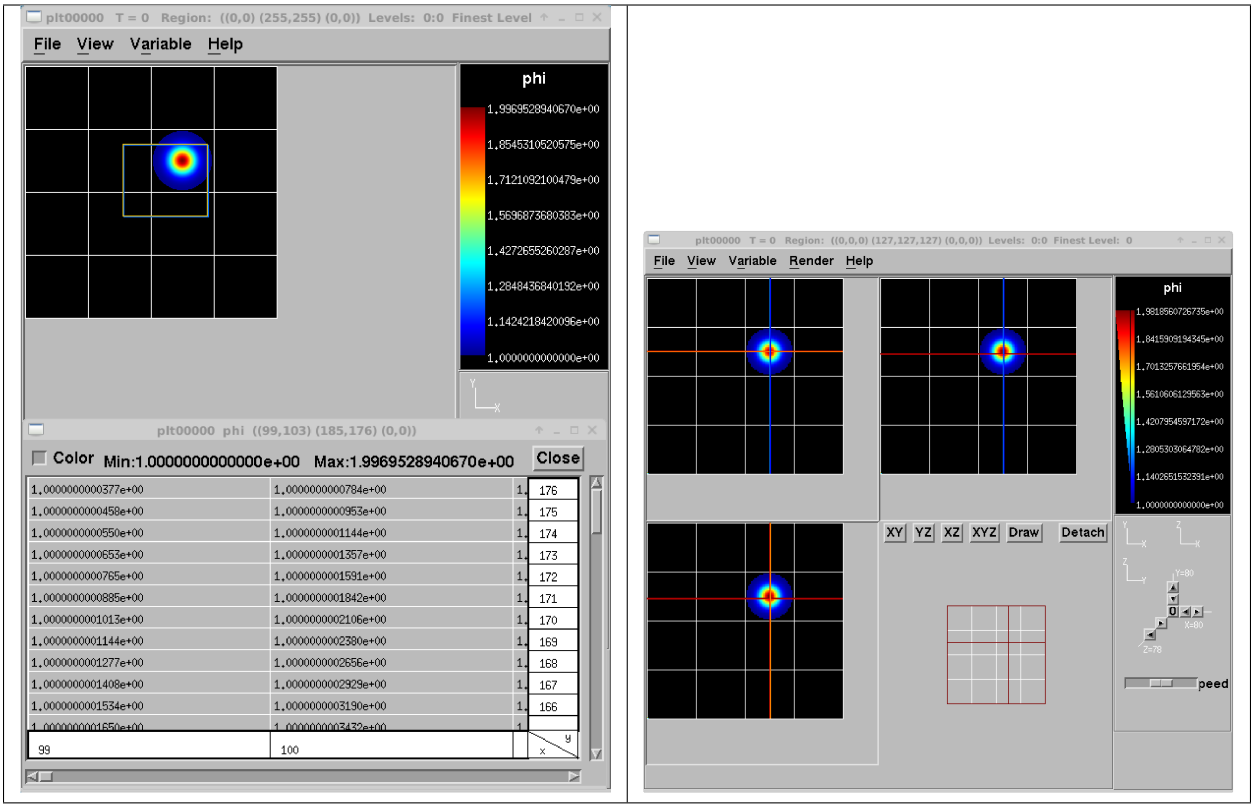
3. **Run**:

By default, the plotfiles are directories that have the form pltXXXXX, where XXXXX is a number corresponding to the timestep that the file was created. Use `amrvis2d <filename>` or `amrvis3d <filename>` to see a single plotfile, or for 2D data sets, `amrvis2d -a plt*`, which will animate the sequence of plotfiles. FArrayBoxes and MultiFabs can also be viewed with the `-fab` and `-mf` options. When opening MultiFabs, use the name of the MultiFab's header file `amrvis2d -mf MyMultiFab_H`.

You can use the "Variable" menu to change the variable. You can left-click drag a box around a region and click "View" → "Dataset" in order to look at the actual numerical values (see Table 16.1). Or you can simply left click on a point to obtain the numerical value. You can also export the pictures in several different formats under `File/Export`. In 2D you can right or center click to get line-out plots. In 3D you can right or center click to change the planes, and hold shift+(right or center) click to get line-out plots.

We have created a number of routines to convert AMReX plotfile data to other formats (such as Matlab), but in order to properly interpret the hierarchical AMR data, each tends to require its own idiosyncrasies. If you would like to display the data in another format, please leave a message on AMReX's GitHub Discussions page.

Table 16.1: . 2D and 3D images generated using Amrvis.



16.1.1 Building Amrvis on macOS

As previously outlined at the end of section *Building with GNU Make*, it is recommended to build using the homebrew package manager to install gcc. Furthermore, you will also need x11 and openmotif. These can be installed using homebrew also:

1. `brew cask install xquartz`
2. `brew install openmotif`

Note that when the `GNUmakefile` detects a macOS install, it assumes that dependencies are installed in the locations that Homebrew uses. Namely the `/usr/local/` tree for regular dependencies and the `/opt/` tree for X11.

16.2 VisIt

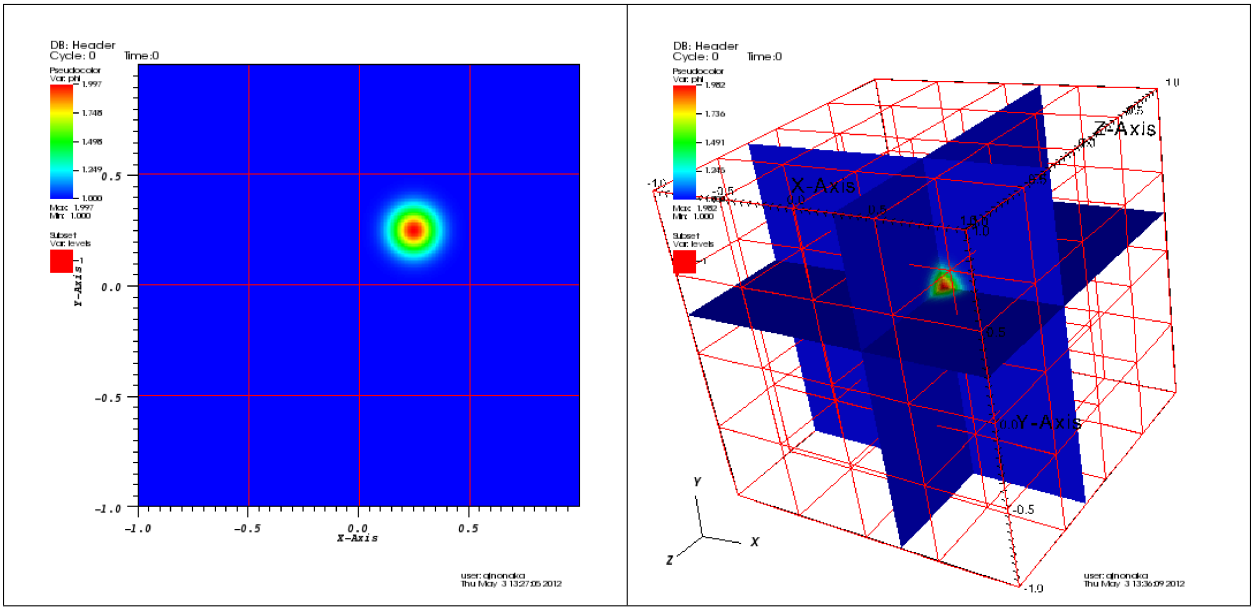
AMReX data can also be visualized by VisIt, an open source visualization and analysis software. To follow along with this example, first build and run the first heat equation tutorial code (see the section on *Example: Heat Equation Solver*).

Next, download and install VisIt from https://wci.llnl.gov/simulation/computer-codes/visit. To open a single plotfile, run VisIt, then select "File" → "Open file …", then select the Header file associated with the plotfile of interest (e.g., `plt00000/Header`). Assuming you ran the simulation in 2D, here are instructions for making a simple plot:

- To view the data, select "Add" → "Pseudocolor" → "phi", and then select "Draw".
- To view the grid structure (not particularly interesting yet, but when we add AMR it will be), select "Add" → "Subset" → "levels". Then double-click the text "Subset - levels", enable the "Wireframe" option, select "Apply", select "Dismiss", and then select "Draw".
- To 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 left side of Table 16.2.

Table 16.2: : 2D (left) and 3D (right) images generated using VisIt.



In 3D, you must apply the "Operators" → "Slicing" → "ThreeSlice", with the "ThreeSlice operator attribute" set to `x=0.25`, `y=0.25`, and `z=0.25`. You can left-click and drag over the image to rotate the image to generate something similar to right side of Table 16.2.

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. This can most easily be done using the command:

```
~/amrex-tutorials/ExampleCodes/Basic/HeatEquation_EX1_C> ls -1 plt*/Header | tee movie.
↪visit
plt00000/Header
plt01000/Header
plt02000/Header
plt03000/Header
plt04000/Header
plt05000/Header
plt06000/Header
plt07000/Header
plt08000/Header
plt09000/Header
plt10000/Header
```

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 on-screen instructions.

Warning: The Visit reader determines the value of `Cycle` from the name of the plotfile (directory), specifically from the integer that follows the string "plt" in the plotfile name. So if you call it `plt00100`, `myplt00100` or `this_is_my_plt00100` then it will correctly recognize and print `Cycle:  100`. If you call it `plt00100_old` it will also correctly recognize and print `Cycle:  100`.

However, if you do not have `plt` followed immediately by the number, e.g. you name it `pltx00100`, then VisIt will not be able to correctly recognize and print the value for `Cycle`. (It will still read and display the data itself.)

16.2.1 VisIt HDF5 Format

The plotfiles generated with the HDF5 format can be visualized by VisIt as well. To open a single plotfile, run VisIt, then select "File" → "Open file …", then select the HDF5 plotfile of interest (e.g.,``plt00000.h5``), and select "Chombo" in the "Open file as type" dropdown menu. VisIt can also recognize the time steps automatically based on the numbers in the HDF5 plotfile names in a directory.

16.3 ParaView

The open source visualization package ParaView v5.7 and later can be used to view 2D and 3D plotfiles, as well as particles data. Download the package at https://www.paraview.org/.

To open a plotfile (for example, you could run the `HeatEquation_EX1_C` in 3D):

1. Run ParaView v5.7, then select "File" → "Open".
2. Navigate to your run directory, and select the fluid or particle plotfile. Note that you can either open single/multiple plotfile(s) at once by selecting them one by one or select an ensemble of file, labelled as `plt..` and indicated as a Group in the "Type" column of the file explorer (see Fig. 16.2). In the later case, Paraview

will load the plotfiles as a time series. ParaView will ask you about the file type – choose "AMReX/BoxLib Grid Reader" or "AMReX/BoxLib Particles Reader".

3. Under the "Cell Arrays" field, select a variable (e.g., "phi") and click "Apply". Note that the default number of refinement levels loaded and visualized is 1. Change to the required number of AMR level before clicking "Apply".
4. Under "Representation" select "Surface".
5. Under "Coloring" select the variable you chose above.
6. To add planes, near the top left you will see a cube icon with a green plane slicing through it. If you hover your mouse over it, it will say "Slice". Click that button.
7. You can play with the Plane Parameters to define a plane of data to view, as shown in Fig. 16.1.

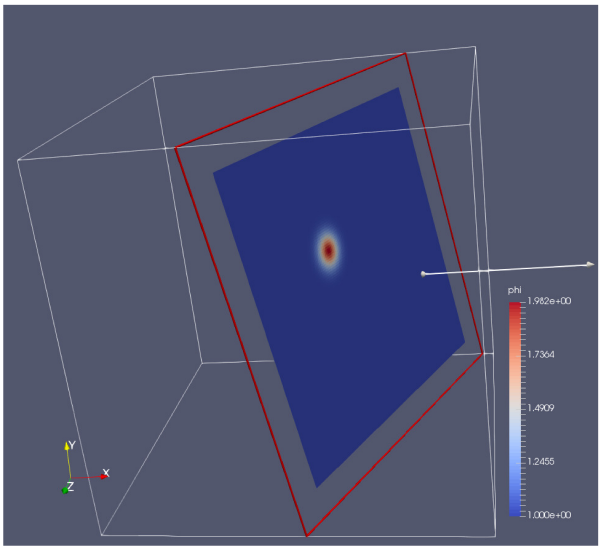


Fig. 16.1: : Plotfile image generated with ParaView

16.3.1 Building an Iso-surface

Note that Paraview is not able to generate iso-surfaces from cell centered data. To build an iso-surface (or iso-line in 2D):

1. Perform a cell to node interpolation: "Filters" → "Alphabetical" → "Cell Data to Point Data".
2. Use the "Contour" icon (next to the calculator) to select the data from which to build the contour ("Contour by"), enters the iso-surfaces values and click "Apply".

16.3.2 Visualizing Particle Data

To visualize particle data within plofile directories (for example, you could run the NeighborList example in Tutorials/Particles):

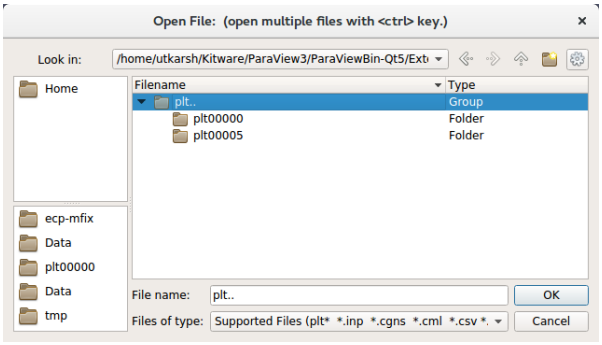


Fig. 16.2: : File dialog in ParaView showing a group of plotfile directories selected

1. Run ParaView v5.7, and select then "File" → "Open". You will see a combined "plt.." group. Click on "+" to expand the group, if you want inspect the files in the group. You can select an individual plotfile directory or select a group of directories to read them a time series, as shown in Fig. 16.2, and click OK. ParaView will ask you about the file type – choose "AMReX/BoxLib Particles Reader".
2. The "Properties" panel in ParaView allows you to specify the "Particle Type", which defaults to "particles". Using the "Properties" panel, you can also choose which point arrays to read.
3. Click "Apply" and under "Representation" select "Point Gaussian".
4. Change the Gaussian Radius if you like. You can scroll through the frames with the VCR-like controls at the top, as shown in Fig. 16.3.

Following these instructions, you can open fluid and/or particles plotfiles and visualize them together on the same Panel View.

Once you have loaded an AMReX plotfile time series (fluid and/or particles), you can generate a movie following these instructions:

1. "File" → "Save Animation…".
2. Enter a file name, select ".avi" as the Type of File and click "OK".
3. Adjust the resolution, compression and framerate, and click "OK"

16.3.3 Plot a Vector Field

Paraview can be used to plot a vector field from AMR plotfile data. In this example we will assume a single vector has been stored as three separate variables, `V_x`, `V_y` and `V_z`. The steps below outline a basic construction:

1. Open a plotfile or plotfile group, using `File` → `Open`. A pop-up will appear, select "AMReX/Boxlib Grid Reader".
2. Select the plotfile or group in the Pipeline Browser. The Cell Array Status window of the Properties should populate with the values `V_x`, `V_y` and `V_z`. Select these values and click apply.
3. Select the Cell Centers filter from `Filters` → `Alphabetical` → `Cell Centers` and apply.

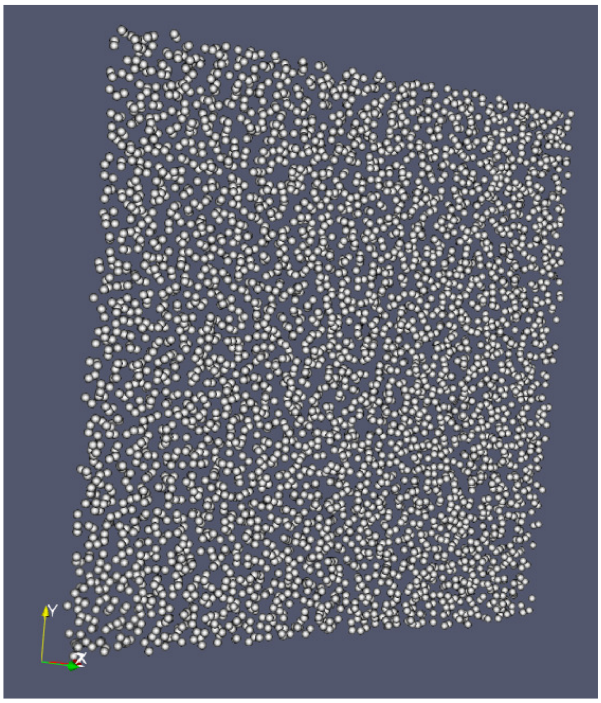


Fig. 16.3: : Particle image generated with ParaView

4. Next we'll define a vector variable using the Calculator filter. Select `Filters` → `Alphabetical` → `Calculator`. Under the Properties heading, set the Attribute Type to Point Data. The Result Array Name is the name of the vector value we will create. In the line below that we define a new vector value with the equation: `V_x*iHat + V_y*jHat + V_z*kHat` Note that, the values `V_x`, `V_y` and `V_z`, should be selectable from the dropdown Scalars menu. Apply the filter.

5. To plot the arrows, select the Glyph filter, `Filters` → `Alphabetical` → `Glyph`. Under the heading, Glyph Source, select `Arrow`. Under Orientation, select the name of the vector value created in the last step. The default name is `Result`. Apply the filter to display the vector field.

   One may want to adjust the appearance of the vector field by scaling each vector by its magnitude. To do this, look under the Scale heading, select the vector value as the Scale Array and select Scale by Magnitude.

   To adjust the number and location of vectors displayed, one may alter the settings under the Masking heading.

16.3.4 ParaView HDF5 Format

The plotfiles generated with the HDF5 format can be visualized by ParaView. To open a single plotfile, run VisIt, select "File" → "Open", then select the HDF5 plotfile (e.g.,``plt00000.h5``). You can select an individual plotfile or select a group of files to read as time series, then click OK. ParaView will ask you about the file type – choose "VisItChomboReader".

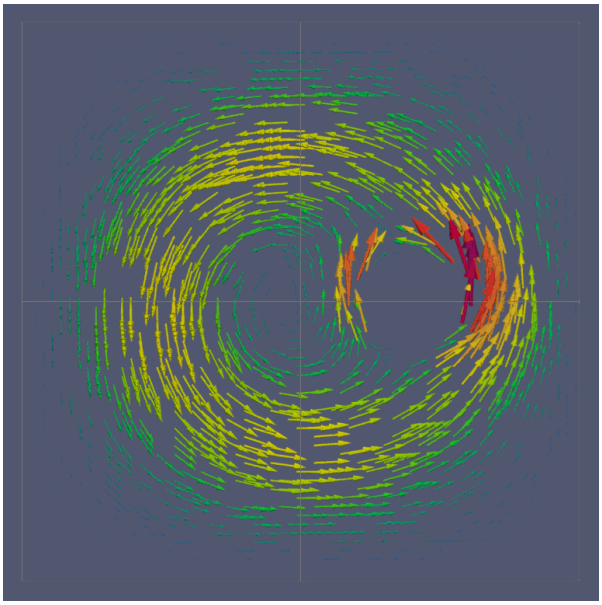


Fig. 16.4: Vector Field generated with ParaView

16.4 yt

yt, an open source Python package available at http://yt-project.org/, can be used for analyzing and visualizing mesh and particle data generated by AMReX codes. Some of the AMReX developers are also yt project members. Below we describe how to use on both a local workstation, as well as at the NERSC HPC facility for high-throughput visualization of large data sets.

Note - AMReX datasets require yt version 3.4 or greater.

16.4.1 Using on a local workstation

Running yt on a local system generally provides good interactivity, but limited performance. Consequently, this configuration is best when doing exploratory visualization (e.g., experimenting with camera angles, lighting, and color schemes) of small data sets.

To use yt on an AMReX plot file, first start a Jupyter notebook or an IPython kernel, and import the `yt` module:

```
In [1]: import yt

In [2]: print(yt.__version__)
3.4-dev
```

Next, load a plot file; in this example we use a plot file from the Nyx cosmology application:

```
In [3]: ds = yt.load("plt00401")
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: current_time              = 0.
↪00605694344696544
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_dimensions         = [128␣
↪128 128]
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_left_edge          = [ 0.  0.
↪  0.]
```

(continues on next page)

(continued from previous page)

```
yt : [INFO     ] 2017-05-23 10:03:56,183 Parameters: domain_right_edge          = [ 14.
↪24501  14.24501  14.24501]

In [4]: ds.field_list
Out[4]:
[('DM', 'particle_mass'),
 ('DM', 'particle_position_x'),
 ('DM', 'particle_position_y'),
 ('DM', 'particle_position_z'),
 ('DM', 'particle_velocity_x'),
 ('DM', 'particle_velocity_y'),
 ('DM', 'particle_velocity_z'),
 ('all', 'particle_mass'),
 ('all', 'particle_position_x'),
 ('all', 'particle_position_y'),
 ('all', 'particle_position_z'),
 ('all', 'particle_velocity_x'),
 ('all', 'particle_velocity_y'),
 ('all', 'particle_velocity_z'),
 ('boxlib', 'density'),
 ('boxlib', 'particle_mass_density')]
```

From here one can make slice plots, 3-D volume renderings, etc. An example of the slice plot feature is shown below:

```
In [9]: slc = yt.SlicePlot(ds, "z", "density")
yt : [INFO     ] 2017-05-23 10:08:25,358 xlim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,358 ylim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,359 xlim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,359 ylim = 0.000000 14.245010

In [10]: slc.show()

In [11]: slc.save()
yt : [INFO     ] 2017-05-23 10:08:34,021 Saving plot plt00401_Slice_z_density.png
Out[11]: ['plt00401_Slice_z_density.png']
```

The resulting image is Fig. 16.5. One can also make volume renderings with ; an example is show below:

```
In [12]: sc = yt.create_scene(ds, field="density", lens_type="perspective")

In [13]: source = sc[0]

In [14]: source.tfh.set_bounds((1e8, 1e15))

In [15]: source.tfh.set_log(True)

In [16]: source.tfh.grey_opacity = True

In [17]: sc.show()
<Scene Object>:
Sources:
    source_00: <Volume Source>:YTRegion (plt00401): , center=[  1.09888770e+25   1.
↪09888770e+25   1.09888770e+25] cm, left_edge=[ 0.  0.  0.] cm, right_edge=[  2.
↪19777540e+25   2.19777540e+25   2.19777540e+25] cm transfer_function:None
```

(continues on next page)

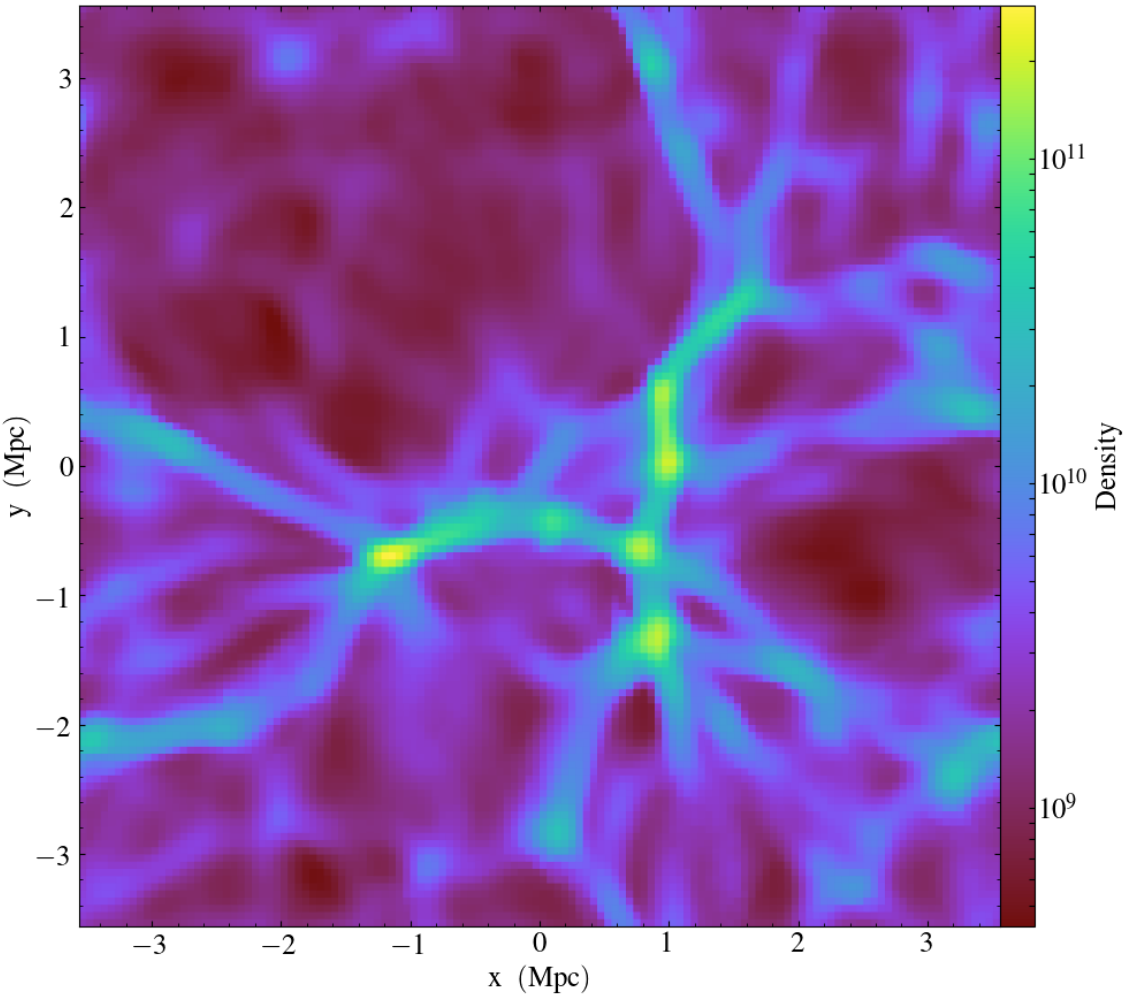


Fig. 16.5: : Slice plot of 128^3 Nyx simulation using yt.

(continued from previous page)

```
Camera:
    <Camera Object>:
    position:[ 14.24501  14.24501  14.24501] code_length
    focus:[ 7.122505  7.122505  7.122505] code_length
    north_vector:[ 0.81649658 -0.40824829 -0.40824829]
    width:[ 21.367515  21.367515  21.367515] code_length
    light:None
    resolution:(512, 512)
Lens: <Lens Object>:
    lens_type:perspective
    viewpoint:[ 0.95423473  0.95423473  0.95423473] code_length

In [19]: sc.save()
yt : [INFO     ] 2017-05-23 10:15:07,825 Rendering scene (Can take a while).
yt : [INFO     ] 2017-05-23 10:15:07,825 Creating volume
yt : [INFO     ] 2017-05-23 10:15:07,996 Creating transfer function
yt : [INFO     ] 2017-05-23 10:15:07,997 Calculating data bounds. This may take a while.
Set the TransferFunctionHelper.bounds to avoid this.
yt : [INFO     ] 2017-05-23 10:15:16,471 Saving render plt00401_Render_density.png
```

The output of this is Fig. 16.6.

16.4.2 Using yt at NERSC (*under development*)

Because yt is Python-based, it is portable and can be used in many software environments. Here we focus on yt's capabilities at NERSC, which provides resources for performing both interactive and batch queue-based visualization and analysis of AMReX data. Coupled with yt's MPI and OpenMP parallelization capabilities, this can enable high-throughput visualization and analysis workflows.

Interactive yt with Jupyter notebooks

Unlike VisIt (see the section on *VisIt*), yt has no client-server interface. Such an interface is often crucial when one has large data sets generated on a remote system, but wishes to visualize the data on a local workstation. Both copying the data between the two systems, as well as visualizing the data itself on a workstation, can be prohibitively slow.

Fortunately, NERSC has implemented several resources which allow one to interact with yt remotely, emulating a client-server model. In particular, NERSC now hosts Jupyter notebooks which run IPython kernels on the Cori system; this provides users access to the `$HOME`, `/project`, and `$SCRATCH` file systems from a web browser-based Jupyter notebook. ***Please note that Jupyter hosting at NERSC is still under development, and the environment may change without notice.***

NERSC also provides Anaconda Python, which allows users to create their own customizable Python environments. It is recommended to install yt in such an environment. One can do so with the following example:

```
user@cori10:~> module load python/3.5-anaconda
user@cori10:~> conda create -p $HOME/yt-conda numpy
user@cori10:~> source activate $HOME/yt-conda
(/global/homes/u/user/yt-conda/) user@cori10:~> pip install yt
```

More information about Anaconda Python at NERSC is here: http://www.nersc.gov/users/data-analytics/data-analytics/python/anaconda-python/.

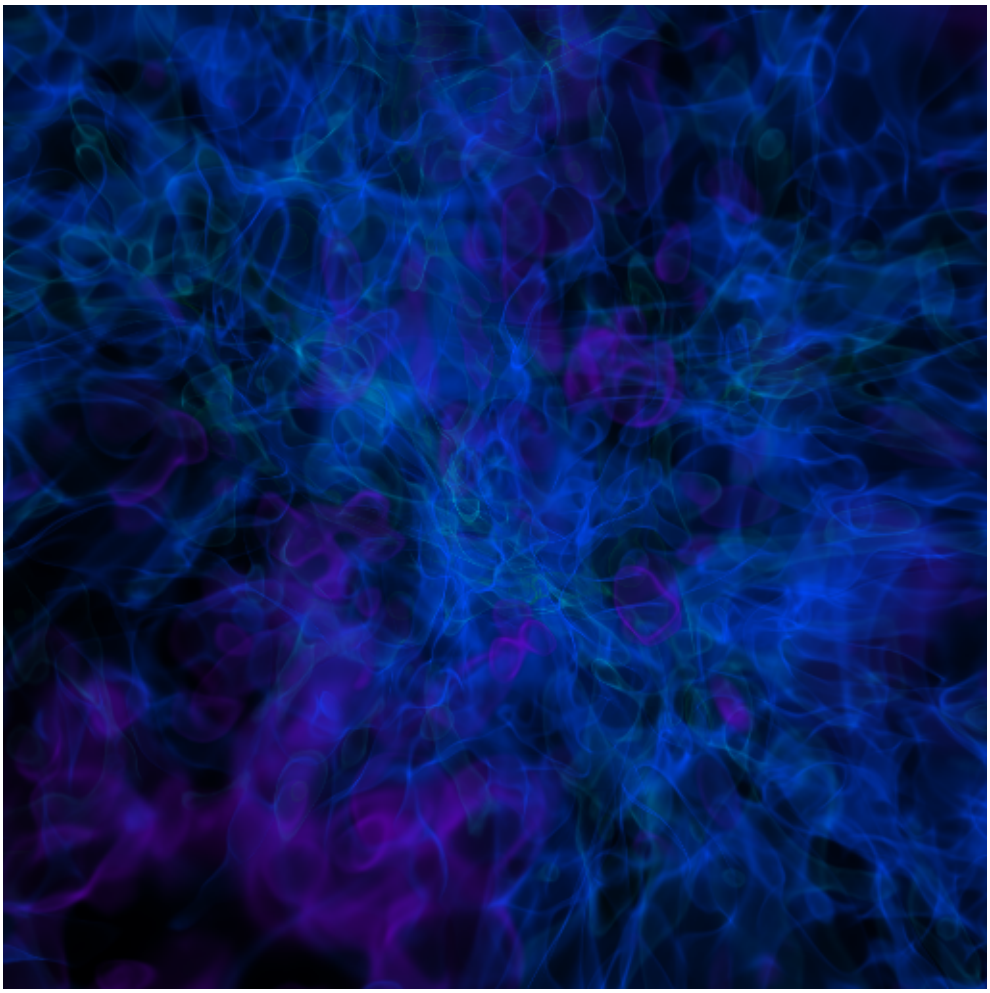


Fig. 16.6: Volume rendering of 128^3 Nyx simulation using yt. This corresponds to the same plot file used to generate the slice plot in Fig. 16.5.

One can then configure this Anaconda environment to run in a Jupyter notebook hosted on the Cori system. Currently this is available in two places: on https://ipython.nersc.gov, and on https://jupyter-dev.nersc.gov. The latter likely reflects what the stable, production environment for Jupyter notebooks will look like at NERSC, but it is still under development and subject to change. To load this custom Python kernel in a Jupyter notebook, follow the instructions at this URL under the "Custom Kernels" heading: http://www.nersc.gov/users/data-analytics/data-analytics/web-applications-for-data-analytics. After writing the appropriate `kernel.json` file, the custom kernel will appear as an available Jupyter notebook. Then one can interactively visualize AMReX plot files in the web browser.[1]

Parallel

Besides the benefit of no longer needing to move data back and forth between NERSC and one's local workstation to do visualization and analysis, an additional feature of yt which takes advantage of the computational resources at NERSC is its parallelization capabilities. yt supports both MPI- and OpenMP-based parallelization of various tasks, which are discussed here: http://yt-project.org/doc/analyzing/parallel_computation.html.

Configuring yt for MPI parallelization at NERSC is a more complex task than discussed in the official yt documentation; the command `pip install mpi4py` is not sufficient. Rather, one must compile `mpi4py` from source using the Cray compiler wrappers `cc`, `CC`, and `ftn` on Cori. Instructions for compiling `mpi4py` at NERSC are provided here: http://www.nersc.gov/users/data-analytics/data-analytics/python/anaconda-python/#toc-anchor-3. After `mpi4py` has been compiled, one can use the regular Python interpreter in the Anaconda environment as normal; when executing yt operations which support MPI parallelization, the multiple MPI processes will spawn automatically.

Although several components of yt support MPI parallelization, a few are particularly useful:

- **Time series analysis.** Often one runs a simulation for many time steps and periodically writes plot files to disk for visualization and post-processing. yt supports parallelization over time series data via the `DatasetSeries` object. yt can iterate over a `DatasetSeries` in parallel, with different MPI processes operating on different elements of the series. This page provides more documentation: http://yt-project.org/doc/analyzing/time_series_analysis.html#time-series-analysis.
- **Volume rendering**. yt implements spatial decomposition among MPI processes for volume rendering procedures, which can be computationally expensive. Note that yt also implements OpenMP parallelization in volume rendering, and so one can execute volume rendering with a hybrid MPI+OpenMP approach. See this URL for more detail: http://yt-project.org/doc/visualizing/volume_rendering.html?highlight=openmp#openmp-parallelization.
- **Generic parallelization over multiple objects.** Sometimes one wishes to loop over a series which is not a `DatasetSeries`, e.g., performing translational or rotational operations on a camera to make a volume rendering in which the field of view moves through the simulation. In this case, one is applying a set of operations on a single object (a single plot file), rather than over a time series of data. For this workflow, yt provides the `parallel_objects()` function. See this URL for more details: http://yt-project.org/doc/analyzing/parallel_computation.html#parallelizing-over-multiple-objects.

  An example of MPI parallelization in yt is shown below, where one animates a time series of plot files from an IAMR simulation while revolving the camera such that it completes two full revolutions over the span of the animation:

```
import yt
import glob
import numpy as np

yt.enable_parallelism()
```

(continues on next page)

[1] It is convenient to use the magic command `%matplotlib inline` in order to render matplotlib figures in the same browser window as the notebook, as opposed to displaying it as a new window.

(continued from previous page)

```
base_dir1 = '/global/cscratch1/sd/user/Nyx_run_p1'
base_dir2 = '/global/cscratch1/sd/user/Nyx_run_p2'
base_dir3 = '/global/cscratch1/sd/user/Nyx_run_p3'

glob1 = glob.glob(base_dir1 + '/plt*')
glob2 = glob.glob(base_dir2 + '/plt*')
glob3 = glob.glob(base_dir3 + '/plt*')

files = sorted(glob1 + glob2 + glob3)

ts = yt.DatasetSeries(files, parallel=True)

frame = 0
num_frames = len(ts)
num_revol = 2

slices = np.arange(len(ts))

for i in yt.parallel_objects(slices):
    sc = yt.create_scene(ts[i], lens_type='perspective', field='z_velocity')

    source = sc[0]
    source.tfh.set_bounds((1e-2, 9e+0))
    source.tfh.set_log(False)
    source.tfh.grey_opacity = False

    cam = sc.camera

    cam.rotate(num_revol*(2.0*np.pi)*(i/num_frames),
               rot_center=np.array([0.0, 0.0, 0.0]))

    sc.save(sigma_clip=5.0)
```

When executed on 4 CPUs on a Haswell node of Cori, the output looks like the following:

```
user@nid00009:~/yt_vis/> srun -n 4 -c 2 --cpu_bind=cores python make_yt_
↪movie.py
yt : [INFO     ] 2017-05-23 16:51:33,565 Global parallel computation␣
↪enabled: 0 / 4
yt : [INFO     ] 2017-05-23 16:51:33,565 Global parallel computation␣
↪enabled: 2 / 4
yt : [INFO     ] 2017-05-23 16:51:33,566 Global parallel computation␣
↪enabled: 1 / 4
yt : [INFO     ] 2017-05-23 16:51:33,566 Global parallel computation␣
↪enabled: 3 / 4
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: current_time     ␣
↪        = 0.103169376949795
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: domain_dimensions␣
↪        = [128 128 128]
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: domain_left_edge ␣
↪        = [ 0.  0.  0.]
P003 yt : [INFO     ] 2017-05-23 16:51:33,958 Parameters: domain_right_edge␣
↪        = [ 6.28318531  6.28318531  6.28318531]
```

(continues on next page)

(continued from previous page)

```
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: current_time     ␣
↪        = 0.0
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_dimensions␣
↪        = [128 128 128]
P002 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: current_time     ␣
↪        = 0.0687808060674485
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_left_edge ␣
↪        = [ 0.  0.  0.]
P002 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_dimensions␣
↪        = [128 128 128]
P000 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_right_edge␣
↪        = [ 6.28318531  6.28318531  6.28318531]
P002 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_left_edge ␣
↪        = [ 0.  0.  0.]
P002 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_right_edge␣
↪        = [ 6.28318531  6.28318531  6.28318531]
P001 yt : [INFO     ] 2017-05-23 16:51:33,973 Parameters: current_time     ␣
↪        = 0.0343922351851018
P001 yt : [INFO     ] 2017-05-23 16:51:33,973 Parameters: domain_dimensions␣
↪        = [128 128 128]
P001 yt : [INFO     ] 2017-05-23 16:51:33,974 Parameters: domain_left_edge ␣
↪        = [ 0.  0.  0.]
P001 yt : [INFO     ] 2017-05-23 16:51:33,974 Parameters: domain_right_edge␣
↪        = [ 6.28318531  6.28318531  6.28318531]
P000 yt : [INFO     ] 2017-05-23 16:51:34,589 Rendering scene (Can take a␣
↪while).
P000 yt : [INFO     ] 2017-05-23 16:51:34,590 Creating volume
P003 yt : [INFO     ] 2017-05-23 16:51:34,592 Rendering scene (Can take a␣
↪while).
P002 yt : [INFO     ] 2017-05-23 16:51:34,592 Rendering scene (Can take a␣
↪while).
P003 yt : [INFO     ] 2017-05-23 16:51:34,593 Creating volume
P002 yt : [INFO     ] 2017-05-23 16:51:34,593 Creating volume
P001 yt : [INFO     ] 2017-05-23 16:51:34,606 Rendering scene (Can take a␣
↪while).
P001 yt : [INFO     ] 2017-05-23 16:51:34,607 Creating volume
```

Because the `parallel_objects()` function transforms the loop into a data-parallel problem, this procedure strong scales nearly perfectly to an arbitrarily large number of MPI processes, allowing for rapid rendering of large time series of data.

16.5 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.

16.5.1 System Architecture

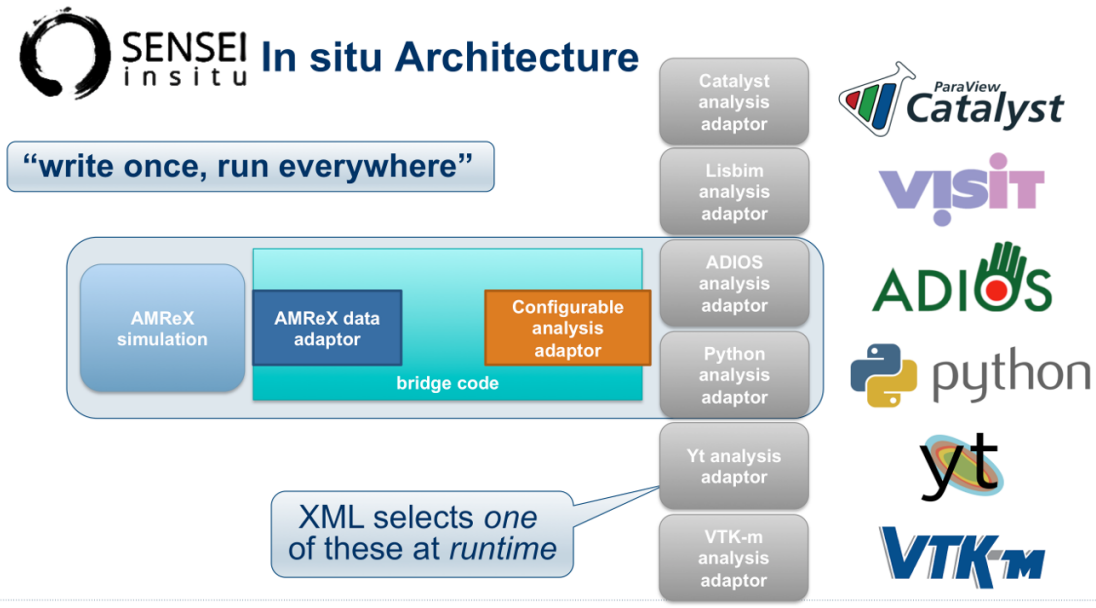


Fig. 16.7: SENSEI's in situ architecture enables use of a diverse of back ends which can be selected at run time via an XML configuration file

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. 16.7. 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.

16.5.2 AMReX Integration

AMReX codes based on `amrex::Amr` can use SENSEI simply by enabling it in the build and run via ParmParse parameters. AMReX codes based on `amrex::AmrMesh` need to additionally invoke the bridge code in `amrex::AmrMeshInSituBridge`.

16.5.3 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
```

16.5.4 Compiling with CMake

For codes making use of AMReX's CMake based build, one needs to enable SENSEI and point to the CMake configuration installed with SENSEI.

```
cmake -DAMReX_SENSEI=ON -DSENSEI_DIR=<path to install>/<lib dir>/cmake ..
```

When CMake generates the make files proceed as usual. Note: <lib dir> may be *lib* or *lib64* or something else depending on what CMake decided to use for your particular OS. See the CMake GNUInstallDirs documentation for more information.

```
make -j4 -f GNUmakefile
```

16.5.5 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:

```
sensei.enabled = 1
sensei.config = render_iso_catalyst_2d.xml
sensei.frequency = 2
```

`sensei.enabled` turns SENSEI on or off. `sensei.config` points to the SENSEI XML file which selects and configures the desired back end. `sensei.frequency` controls the number of level 0 time steps in between SENSEI processing.

16.5.6 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 IAMR to render 10 iso surfaces, shown in figure Fig. 16.8, using VisIt Libsim.

```
<sensei>
  <analysis type="libsim" frequency="1" mode="batch"
    visitdir="/usr/common/software/sensei/visit"
    session="rt_sensei_configs/visit_rt_contour_alpha_10.session"
    image-filename="rt_contour_%ts" image-width="1555" image-height="815"
    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.

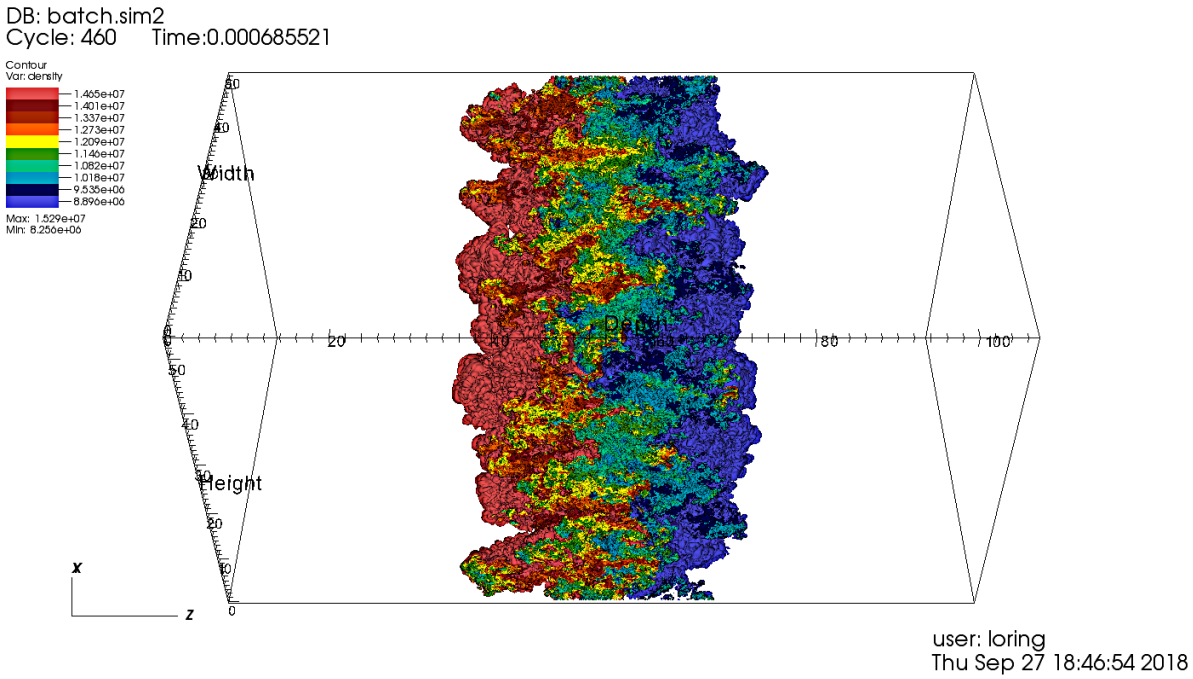


Fig. 16.8: SENSEI-Libsim in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.

The same run and visualization was repeated using ParaView Catalyst, shown in figure Fig. 16.9, by providing the following XML configuration.

```
<sensei>
  <analysis type="catalyst" pipeline="pythonscript"
    filename="rt_sensei_configs/rt_contour.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.

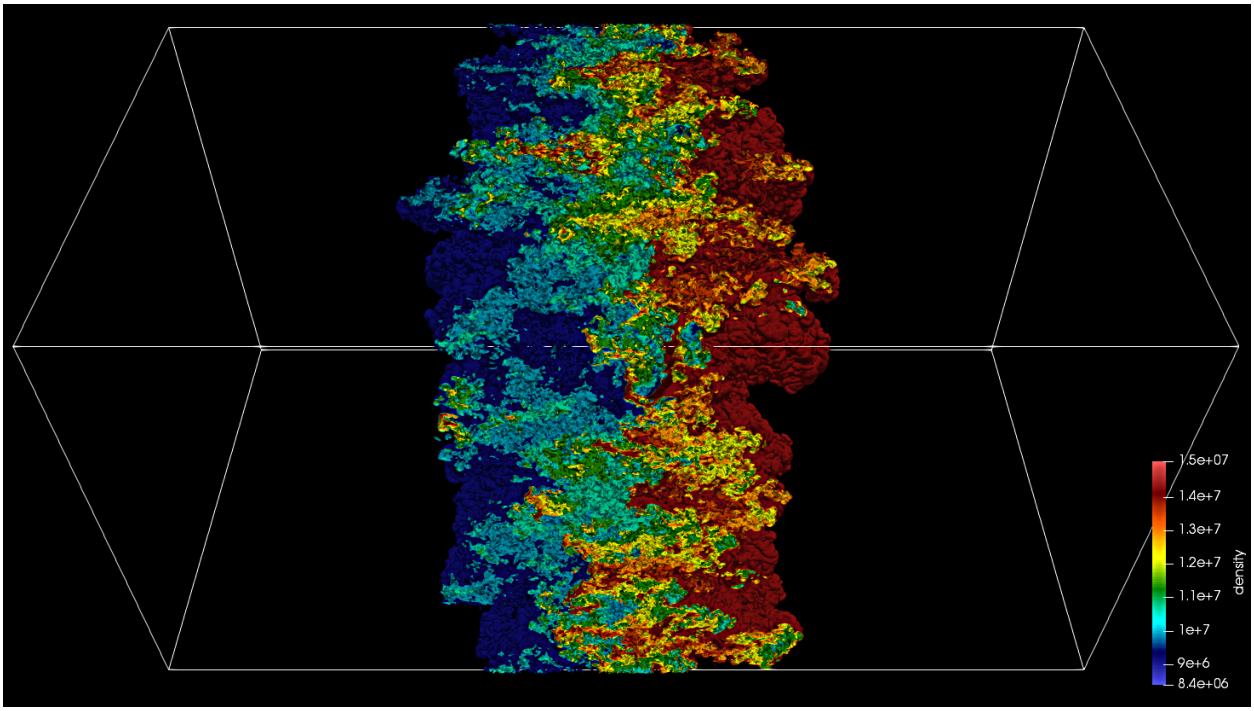


Fig. 16.9: SENSEI-Catalyst in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.

16.5.7 Obtaining SENSEI

SENSEI is hosted on github at https://github.com/SENSEI-insitu/SENSEI.git

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.

AmrLevel Tutorial with Catalyst

The following steps show how to run the tutorial with ParaView Catalyst. The simulation will periodically write images during the run.

```
ssh cori.nersc.gov
cd $SCRATCH
git clone https://github.com/AMReX-Codes/amrex.git
git clone https://github.com/AMReX-Codes/amrex-tutorials.git
cd amrex-tutorials/ExampleCodes/Amr/Advection_AmrLevel/Exec/SingleVortex
module use /usr/common/software/sensei/modulefiles
module load sensei/2.1.0-catalyst-shared
source sensei_config
vim GNUmakefile
# USE_SENSEI_INSITU=TRUE
make -j4 -f GNUmakefile
vim inputs
# sensei.enabled=1
# sensei.config=sensei/render_iso_catalyst_2d.xml
salloc -C haswell -N 1 -t 00:30:00 -q debug
```

(continues on next page)

(continued from previous page)

```
cd $SCRATCH/amrex-tutorials/ExampleCodes/Amr/Advection_AmrLevel/Exec/SingleVortex
./main2d.gnu.haswell.MPI.ex inputs
```

AmrLevel Tutorial with Libsim

The following steps show how to run the tutorial with VisIt Libsim. The simulation will periodically write images during the run.

```
ssh cori.nersc.gov
cd $SCRATCH
git clone https://github.com/AMReX-Codes/amrex.git
git clone https://github.com/AMReX-Codes/amrex-tutorials.git
cd amrex-tutorials/ExampleCodes/Amr/Advection_AmrLevel/Exec/SingleVortex
module use /usr/common/software/sensei/modulefiles
module load sensei/2.1.0-libsim-shared
source sensei_config
vim GNUmakefile
# USE_SENSEI_INSITU=TRUE
make -j4 -f GNUmakefile
vim inputs
# sensei.enabled=1
# sensei.config=sensei/render_iso_libsim_2d.xml
salloc -C haswell -N 1 -t 00:30:00 -q debug
cd $SCRATCH/amrex-tutorials/ExampleCodes/Amr/Advection_AmrLevel/Exec/SingleVortex
./main2d.gnu.haswell.MPI.ex inputs
```

CHAPTER

SEVENTEEN

POST-PROCESSING

There are utilities you can build that can read in plotfiles into a `MultiFab` and perform post-processing. Since the data is read into `MultiFab` you can perform standard `MFIter` loops to iterate over the data to perform calculations.

17.1 Post-Processing

The following is a list of tools you may find useful for processing plotfile data generated by AMReX codes.

17.1.1 WritePlotfileToASCII

This basic routine reads in a single-level plotfile and writes the entire contents to the standard output, one line at a time for each data value. After reading in the plotfile to a `MultiFab`, the program copies the data into a separate `MultiFab` with one large grid to make writing the data out sequentially an easier task.

In `amrex/Tools/Postprocessing/C_Src`, edit `GNUMakefile` to read `EBASE = WritePlotfileToASCII` and `NEEDS_f90_SRC = FALSE` and then `make` to generate an executable. To run the executable, `<executable> infile=<plotfilename>`. You can modify the cpp file to write out on certain components, coordinates, row/column formatting, etc.

17.1.2 fextract

This basic routine reads in a single-level plotfile and extracts selected contents along a 1-D axis to an ascii file.

How to build and run

In `amrex/Tools/Plotfile`, just type `make` to generate an executable. To run the executable, execute `./fextract.gnu.ex` to see the full command line and all the available options. It is possible to select the axis (`-d` flag) where the data are collected. By default the axis is taken at the center of the domain. A generic ASCII file is generated by default, which contains many details of the simulation. However data can be exported in a raw csv file with the command `-csv`.

Example

```
user@machine:~/AMReX/amrex/Tools/Plotfile(postproc_docs)$ ./fextract.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_Equil_2d_Bench/plt0000003

 slicing along x-direction at coarse grid (j,k)=(16,0) and output to /home/user/AMReX/
↪FHDeX/exec/multispec/Reg_Equil_2d_Bench/plt0000003.slice
```

This produces an ascii file of the form:

```
user@machine:~/AMReX/FHDeX/exec/multispec/Reg_Equil_2d_Bench(main)$ cat plt0000003.slice
# 1-d slice in x-direction, file: /home/user/AMReX/FHDeX/exec/multispec/Reg_Equil_2d_
↪Bench/plt0000003
# time =  0.30000000000000004
#                          x                        rho                       rho1                ␣
↪       rho2
                       0.5       2.9993686498953114      0.60059557892152249       1.
↪0502705977511799
                       1.5       3.0003554204928884      0.59935306004478783       1.
↪0508550827449006
                       2.5       3.0008794559257246       0.5990345897671786       1.
↪0500559828760208
                       3.5       2.9997442287698322      0.60001913923213179       1.
↪0508294996618532
                       4.5       3.0001395958111967      0.60021852440041579       1.
↪0487977074444519
                       5.5       3.0000989976613459      0.60022830117083248       1.
↪0489080268816791
```

17.1.3 fcompare

Compares two plotfiles, zone by zone, to machine precision and reports the maximum absolute and relative errors for each variable.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fcompare.gnu.ex` to run. Typing `./fcompare.gnu.ex` without inputs will bring up usage and options.

Example

```
user@machine:~/AMReX/amrex/Tools/Plotfile$   ./fcompare.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000000 \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000003

           variable name           absolute error            relative error
                                      (||A - B||)         (||A - B||/||A||)
 ----------------------------------------------------------------------------
 level = 0
 rho                                  0.020039805             0.00845645443
 rho1                                 0.01703166127           0.01450634203
 rho2                                 0.01737072831           0.01479513491
 rho3                                 0.01436258458           0.01436258458
 c1                                  0.003022939351           0.00610148453
 c2                                  0.003167240107          0.006392740399
 c3                                  0.006190179458          0.006190179458
 averaged_velx                      0.0001120979347           0.02141254606
 averaged_vely                      0.0001120979347           0.02141254606
```

(continues on next page)

(continued from previous page)

```
shifted_velx                    0.0001151524563             0.02145887678
shifted_vely                    0.0001151524563             0.02145887678
pres                              0.05687549245          1.797693135e+308
```

17.1.4 fboxinfo

Displays information about AMR levels and boxes. Works with 1-, 2- or 3-dimensional datasets.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fboxinfo.gnu.ex` to run. Typing `./fboxinfo.gnu.ex` without inputs will bring up usage and options.

Example

```
user@machine:~/AMReX/amrex/Tools/Plotfile$ ./fboxinfo.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000000

 plotfile: /home/user/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000000
 level   0: number of boxes =      4, volume = 100.00%
            maximum zones =        64 x      64
```

17.1.5 fvarnames

Takes a single plotfile and displays a list of the variables present.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fvarnames.gnu.ex` to run. Typing `./fvarnames.gnu.ex` without inputs will bring up usage and description.

Example

```
user@machine:~/AMReX/amrex/Tools/Plotfile$ ./fvarnames.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000000
    0   rho
    1   rho1
    2   rho2
    3   rho3
    4   c1
    5   c2
    6   c3
    7   averaged_velx
    8   averaged_vely
    9   shifted_velx
```

(continues on next page)

(continued from previous page)

```
  10   shifted_vely
  11   pres
```

17.1.6 ftime

Takes a whitespace separated list of plotfiles and returns the time for each plotfile.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./ftime.gnu.ex` to run. Typing `./ftime.gnu.ex` without inputs will bring up usage and description.

Example

```
user@machine :~/AMReX/amrex/Tools/Plotfile$ ./ftime.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000000 \
> ~/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000002 \
> ~/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000004 \
> ~/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000006

 /home/user/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000000    0
 /home/user/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000002    4.
↪000000000000001e-13
 /home/user/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000004    8.
↪000000000000002e-13
 /home/user/AMReX/FHDeX/exec/multispec/Spinodal_Charges_2d_Bench/plt0000006    1.
↪1999999999999999e-12
```

17.1.7 fsnapshot

Produces an image of a 2-d plotfile, or a slice of a 3-d plotfile.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fsnapshot.gnu.ex` to run. Typing `./fsnapshot.gnu.ex` without inputs will bring up usage and options.

Example

In this example an image of the data from the 2-d plotfile `plt0000003` is created.

```
user@silentm:~/AMReX/amrex/Tools/Plotfile$ ./fsnapshot.gnu.ex \
> -v rho -p Palette ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000003
 plotfile variable maximum = 2.349724636
 plotfile variable minimum = 1
```

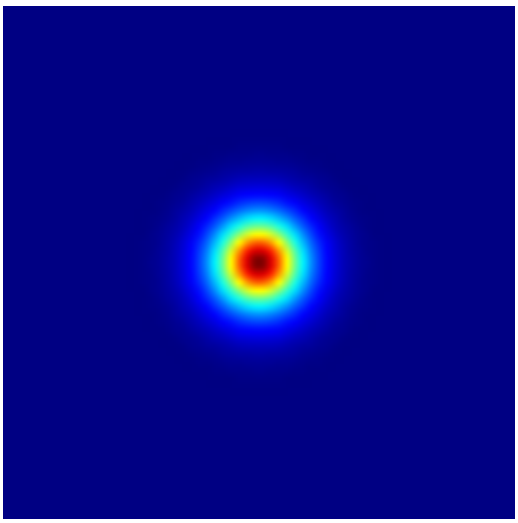
This command tells `fsnapshot` to plot the variable `rho` using the palette `Palette` which is available in the current directory, `amrex/Tools/Plotfile`. The image is created in the same directory as the plotfile folder.

```
user@machine:~/AMReX/amrex/Tools/Plotfile$ ls ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_
↪2d_Bench/
 plt0000000  plt0000003  plt0000003.rho.ppm
```

The image is produced in the portable pixmap format (.ppm). It can be displayed using the command `display` from ImageMagick as seen below.

```
user@machine:~/AMReX/amrex/Tools/Plotfile$ display \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000003.rho.ppm
```

This should produce a window to view the image. (The example here is enlarged for clarity.):



17.1.8 fnan

Takes a single plot file and reports whether each variable contains NaN values.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fnan.gnu.ex` to run. Typing `./fnan.gnu.ex` without inputs will bring up usage and description.

Example

```
user@machine:~/AMReX/amrex/Tools/Plotfile$ ./fnan.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000003
 rho             : clean
 rho1            : clean
 rho2            : clean
 rho3            : clean
 c1              : clean
 c2              : clean
```

(continues on next page)

(continued from previous page)

```
 c3           : clean
 averaged_velx : clean
 averaged_vely : clean
 shifted_velx  : clean
 shifted_vely  : clean
 pres          : clean
```

In this example, there were no NaN values found in the variable data.

17.1.9 fextrema

Report the extrema (min/max) for each variable in a plotfile.

How to build and run

In `amrex/Tools/Plotfile`, type `make` and then `./fextrema.gnu.ex` to run. Typing `./fextrema.gnu.ex` without inputs will bring up usage and options.

Example

```
user@:~/AMReX/amrex/Tools/Plotfile(postproc_docs)$ ./fextrema.gnu.ex \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000000 \
> ~/AMReX/FHDeX/exec/multispec/Reg_DetBubble_2d_Bench/plt0000003
#                     time |rho                                                 |rho1                ␣
↪                          |rho2                                                   |rho3              ␣
↪                             |c1                                                     |c2              ␣
↪                                |c3                                                     |averaged_
↪velx                              |averaged_vely                                                    ␣
↪|shifted_velx                                  |shifted_vely                                          ␣
↪   |pres                                        |
#                          |         min                    max          |         min                ␣
↪             max          |         min                    max          |         min                ␣
↪                max          |         min                    max          |                    ␣
↪min                  max          |         min                    max          |                ␣
↪   min                    max          |         min                    max          |            ␣
↪      min                    max          |         min                    max                   ␣
↪|         min                    max          |
                      0                         1          2.369764441         8.
↪277319027e-17          1.174083806         8.277319027e-17          1.174083806                    ␣
↪   0.02159682815                    1         8.277319027e-17         0.4954432542                ␣
↪    8.277319027e-17         0.4954432542          0.009113491527                    1             ␣
↪       -0.005235152063       0.005235152063         -0.005235152063       0.
↪005235152063         -0.005366192156       0.005366192156         -0.005366192156                 ␣
↪ 0.005366192156                         0                    0
                    0.03                          1          2.349724636         8.
↪277319027e-17          1.157052145         8.277319027e-17          1.156713078                    ␣
↪   0.03595941273                    1         8.277319027e-17         0.4924203149                ␣
↪    8.277319027e-17         0.4922760141          0.01530367099                    1             ␣
↪       -0.005172583789       0.005172583789         -0.005172583789       0.
↪005172583789         -0.005287367803       0.005287367803         -0.005287367803                 ␣
↪ 0.005287367803         -0.004924487345        0.05687549245
```

(continues on next page)

(continued from previous page)

CHAPTER

EIGHTEEN

DEBUGGING

Debugging is an art. Everyone has their own favorite method. Here we offer a few tips we have found to be useful.

To help debugging, AMReX handles various signals in the C standard library raised in the runs. This gives us a chance to print out more information using Linux/Unix backtrace capability. The signals include segmentation fault (or "segfault"), interruption by the user (control-c), assertion errors, and floating point exceptions (NaNs, divided by zero and overflow). The handling of segfault, assertion errors and interruption by control-C are enabled by default. Note that `AMREX_ASSERT()` is only on when compiled with `DEBUG=TRUE` or `USE_ASSERTION=TRUE` in GNU make, or with `-DCMAKE_BUILD_TYPE=Debug` or `-DAMReX_ASSERTIONS=YES` in CMake. The trapping of floating point exceptions is not enabled by default unless the code is compiled with `DEBUG=TRUE` in GNU make, or with `-DCMAKE_BUILD_TYPE=Debug` or `-DAMReX_FPE=YES` in CMake to turn on compiler flags if supported. Alternatively, one can always use runtime parameters to control the handling of floating point exceptions: `amrex.fpe_trap_invalid` for NaNs, `amrex.fpe_trap_zero` for division by zero and `amrex.fpe_trap_overflow` for overflow. To more effectively trap the use of uninitialized values, AMReX also initializes `FArrayBox`s in `MulitFab`s and arrays allocated by `bl_allocate` to signaling NaNs when it is compiled with `TEST=TRUE` or `DEBUG=TRUE` in GNU make, or with `-DCMAKE_BUILD_TYPE=Debug` in CMake. One can also control the setting for `FArrayBox` using the runtime parameter, `fab.init_snan`.

One can get more information than the backtrace of the call stack by instrumenting the code. Here is an example. You know the line `Real rho = state(cell,0);` is causing a segfault. You could add a print statement before that. But it might print out thousands (or even millions) of line before it hits the segfault. What you could do is the following,

```
#include <AMReX_BLBackTrace.H>

std::ostringstream ss;
ss << "state.box() = " << state.box() << " cell = " << cell;
BL_BACKTRACE_PUSH(ss.str()); // PUSH takes std::string

Real rho = state(cell,0);  // state is a Fab, and cell is an IntVect.

BL_BACKTRACE_POP(); // One can omit this line.  In that case,
                    // there is an implicit POP when "PUSH" is
                    // out of scope.
```

When it hits the segfault, you will only see the last pint out.

Writing a `MultiFab` to disk with

```
VisMF::Write(const FabArray<FArrayBox>& mf, const std::string& name)
```

in `AMReX_VisMF.H` and examining it with `Amrvis` (section *Amrvis*) can be helpful as well. In `AMReX_MultiFabUtil.H`, function

```
void print_state(const MultiFab& mf, const IntVect& cell, const int n=-1,
                 const IntVect& ng = IntVect::TheZeroVector());
```

can output the data for a single cell. `n` is the component, with the default being to print all components. `ng` is the number of ghost cells to include.

Valgrind is one of our favorite debugging tools. For MPI runs, one can tell Valgrind to output to different files for different processes. For example,

```
mpiexec -n 4 valgrind --leak-check=yes --track-origins=yes --log-file=vallog.%p ./foo.
↪exe ...
```

18.1 Breaking into Debuggers

In order to break into debuggers and use modern IDEs, the backtrace signal handling described above needs to be disabled.

The following runtime options need to be set in order to prevent AMReX from catching the break signals before a debugger can attach to a crashing process:

```
amrex.throw_exception = 1
amrex.signal_handling = 0
```

This default behavior can also be modified by applications, see for example this custom application initializer.

18.2 Basic Gpu Debugging

The asynchronous nature of GPU execution can make tracking down bugs complex. The relative timing of improperly coded functions can cause variations in output and the timing of error messages may not linearly relate to a place in the code. One strategy to isolate specific kernel failures is to add `amrex::Gpu::synchronize()` or `amrex::Gpu::streamSynchronize()` after every `ParallelFor` or similar `amrex::launch` type call. These synchronization commands will halt execution of the code until the GPU or GPU stream, respectively, has finished processing all previously requested tasks, thereby making it easier to locate and identify sources of error.

18.2.1 Debuggers and Related Tools

Users may also find debuggers useful. Architecture agnostic tools include `gdb`, `hpctoolkit`, and `Valgrind`. Note that there are architecture specific implementations of `gdb` such as `cuda-gdb`, `rocgdb`, `gdb-amd`, and the Intel `gdb`. Usage of several of these variations are described in the following sections.

For advance debugging topics and tools, refer to system-specific documentation (e.g. https://docs.olcf.ornl.gov/systems/summit_user_guide.html#debugging).

18.2.2 CUDA-Specific Tests

- To test if your kernels have launched, run:

```
nvprof ./main3d.xxx
```

  If using NVIDIA Nsight Compute instead, access `nvprof` functionality with:

```
nsys nvprof ./main3d.xxx
```

- Run `nvprof -o profile%p.nvvp ./main3d.xxxx` or `nsys profile -o nsys_out.%q{SLURM_PROCID}.%q{SLURM_JOBID} ./main3d.xxx` for a small problem and examine page faults using `nvvp` or `nsight-sys $(pwd)/nsys_out.#.######.qdrep`.
- Run under `cuda-memcheck` or the newer version `compute-sanitizer` to identify memory errors.
- Run under `cuda-gdb` to identify kernel errors.
- To help identify race conditions, globally disable asynchronicity of kernel launches for all CUDA applications by setting `CUDA_LAUNCH_BLOCKING=1` in your environment variables. This will ensure that only one CUDA kernel will run at a time.

18.2.3 AMD ROCm-Specific Tests

- To test if your kernels have launched, run:

```
rocprof ./main3d.xxx
```

- Run `rocprof --hsa-trace --stats --timestamp on --roctx-trace ./main3d.xxxx` for a small problem and examine tracing using `chrome://tracing`.
- Run under `rocgdb` for source-level debugging.
- To help identify if there are race conditions, globally disable asynchronicity of kernel launches by setting `CUDA_LAUNCH_BLOCKING=1` or `HIP_LAUNCH_BLOCKING=1` in your environment variables. This will ensure only one kernel will run at a time. See the AMD ROCm docs' chicken bits section for more debugging environment variables.

18.2.4 Intel GPU Specific Tests

- To test if your kernels have launched, run:

```
./ze_tracer ./main3d.xxx
```

- Run Intel Advisor, `advisor --collect=survey ./main3d.xxx` for a small problem with 1 MPI process and examine metrics.
- Run under `gdb` with the Intel Distribution for GDB.
- To report back-end information, set `ZE_DEBUG=1` in your environment variables.

CHAPTER

NINETEEN

RUN-TIME INPUTS

19.1 Problem Definition

The following inputs must be preceded by "amr."

	Description	Type	Default
n_cell	Number of cells at level 0 in each coordinate direction	Int Int Int	None
max_level	Maximum level of refinement allowed (0 when single-level)	Int	None

The following inputs must be preceded by "geometry."

	Description	Type	Default
coord_sys	0 for Cartesian	Int	0
is_periodic	1 for true, 0 for false (one value for each coordinate direction)	Ints	0 0 0
prob_lo	Low corner of physical domain (physical not index space)	Reals	0 0 0
prob_hi	High corner of physical domain (physical not index space)	Reals	None
prob_extent	Extent of physical domain (physical not index space)	Reals	None

Note that internally `prob_lo` and `prob_hi` are the variables carried by the `Geometry` class. In the inputs file (or command line), one can specify 1) `geometry.prob_hi` only or 2) `geometry.prob_extent` only or 3) `geometry.prob_lo` and `geometry.prob_hi` or 4) `geometry.prob_lo` and `geometry.prob_extent`. If `geometry.prob_lo` is not specified then it will be 0 in each coordinate direction. If `geometry.prob_extent` is specified (and `geometry.prob_hi` is not) then internally "prob_hi" will be set to "prob_lo" + "prob_extent".

19.2 Time Stepping

The following inputs must be preceded by "amr." Note that if both are specified, both criteria are used and the simulation still stop when the first criterion is hit. In the case of unsteady flow, the simulation will stop when either the number of steps reaches max_step or time reaches stop_time. In the case of unsteady flow, the simulation will stop when either the tolerance (difference between subsequent steps) is reached or the number of iterations reaches the maximum number specified.

	Description	Type	Default
max_step	Maximum number of time steps to take	Int	-1
stop_time	Maximum time to reach	Real	-1.0

19.3 Gridding and Load Balancing

The following inputs must be preceded by "amr" and determine how we create the grids and how often we regrid.

Parameter	Description	Type	Default
regrid_int	How often to regrid (in number of steps at level 0) if regrid_int = -1 then no regridding will occur	Int	-1
max_grid_size_x	Maximum number of cells at level 0 in each grid in x-direction	Int	32
max_grid_size_y	Maximum number of cells at level 0 in each grid in y-direction	Int	32
max_grid_size_z	Maximum number of cells at level 0 in each grid in z-direction	Int	32
blocking_factor_x	Each grid must be divisible by blocking_factor_x in x-direction (must be 1 or power of 2)	Int	8
blocking_factor_y	Each grid must be divisible by blocking_factor_y in y-direction (must be 1 or power of 2)	Int	8
blocking_factor_z	Each grid must be divisible by blocking_factor_z in z-direction (must be 1 or power of 2)	Int	8
refine_grid_layout	Split grids in half until the number of grids is no less than the number of procs. (Will be overridden if refine_grid_layout_[x,y,z] is specified)	Bool	true
refine_grid_layout_x	Allow grids to be split in the x-dimension when refining the layout. (1 to allow or 0 to disallow)	Int	1
refine_grid_layout_y	Allow grids to be split in the y-dimension when refining the layout. (1 to allow or 0 to disallow)	Int	1
refine_grid_layout_z	Allow grids to be split in the z-dimension when refining the layout. (1 to allow or 0 to disallow)	Int	1

The following inputs must be preceded by "particles".

Parameter	Description	Type	Default
max_grid_size_x	Maximum number of cells at level 0 in each grid in x-direction for grids in the ParticleBoxArray if dual_grid is true	Int	32
max_grid_size_y	Maximum number of cells at level 0 in each grid in y-direction for grids in the ParticleBoxArray if dual_grid is true	Int	32
max_grid_size_z	Maximum number of cells at level 0 in each grid in z-direction for grids in the ParticleBoxArray if dual_grid is true.	Int	32

19.4 Plotfiles and Other Output

The following inputs must be preceded by "amr" and control the frequency and naming of plotfile generation, as well as whether a plotfile should be written out immediately after restarting a simulation.

	Description	Type	Default
plot_int	Frequency of plotfile output; if -1 then no plotfiles will be written	Int	-1
plotfile_on_restart	Should we write a plotfile when we restart (only used if plot_int>0)	Bool	False
plot_file	Prefix to use for plotfile output	String	plt

19.5 Checkpoint/Restart

The following inputs must be preceded by "amr" and control checkpoint/restart.

	Description	Type	Default
restart	If present, then the name of file to restart from	String	None
check_int	Frequency of checkpoint output; if -1 then no checkpoints will be written	Int	-1
check_file	Prefix to use for checkpoint output	String	chk

CHAPTER

TWENTY

AMREX-BASED PROFILING TOOLS

AMReX-based application codes can be instrumented using AMReX-specific performance profiling tools that take into account the hierarchical nature of the mesh in most AMReX-based applications. These codes can be instrumented for varying levels of profiling detail.

Here are links to short courses (slides) on how to use the profiling tools. More details can be found in the documentation below.

Lecture 1: Introduction and TINYPROFILER

Lecture 2: Introduction to Full Profiling

Lecture 3: Using ProfVis – GUI Features

Lecture 4: Batch Options and Advanced Profiling Flags

20.1 Types of Profiling

AMReX's built-in profiling works through objects that start and stop timers based on user-placed macros or an object's constructor and destructor. The results from these timers are stored in a global list that is consolidated and printed during finalization, or at a user-defined flush point.

Currently, AMReX has two options for built-in profiling: *Tiny Profiling* and *Full Profiling*.

20.1.1 Tiny Profiling

To enable "Tiny Profiling" with GNU Make edit the options in the file `GNUMakefile` to show,

```
TINY_PROFILE = TRUE
PROFILE      = FALSE
```

If building with CMake, set the following CMake flags,

```
AMReX_TINY_PROFILE = ON
AMReX_BASE_PROFILE = OFF
```

Note: If you set `PROFILE = TRUE` (or `AMReX_BASE_PROFILE = ON`) to enable full profiling then this will override the `TINY_PROFILE` flag and tiny profiling will be disabled.

Output

At the end of a run, a summary of exclusive and inclusive function times will be written to `stdout`. This output includes the minimum and maximum (over processes) time spent in each routine as well as the average and the maximum percentage of total run time. See the sample output below.

```
TinyProfiler total time across processes [min...avg...max]: 1.765...1.765...1.765
--------------------------------------------------------------------------------
Name                          NCalls   Excl. Min   Excl. Avg   Excl. Max   Max  %
--------------------------------------------------------------------------------
mfix_level::EvolveFluid       1        1.602       1.668       1.691       95.83%
FabArray::FillBoundary()      11081    0.02195     0.03336     0.06617      3.75%
FabArrayBase::getFB()         22162    0.02031     0.02147     0.02275      1.29%
PC<...>::WriteAsciiFile()     1        0.00292     0.004072    0.004551     0.26%


--------------------------------------------------------------------------------
Name                          NCalls   Incl. Min   Incl. Avg   Incl. Max    Max  %
--------------------------------------------------------------------------------
mfix_level::Evolve()          1        1.69        1.723       1.734        98.23%
mfix_level::EvolveFluid       1        1.69        1.723       1.734        98.23%
FabArray::FillBoundary()      11081    0.04236     0.05485     0.08826       5.00%
FabArrayBase::getFB()         22162    0.02031     0.02149     0.02275       1.29%
```

The tiny profiler automatically writes the results to `stdout` at the end of your code, when `amrex::Finalize();` is reached. However, you may want to write partial profiling results to ensure your information is saved when you may fail to converge or if you expect to run out of allocated time. Partial results can be written at user-defined points in the code by inserting the line:

```
BL_PROFILE_TINY_FLUSH();
```

Any timers that have not reached their `BL_PROFILE_VAR_STOP` call or exited their scope and deconstructed will not be included in these partial outputs. (e.g., a properly instrumented `main()` should show a time of zero in all partial outputs.) Therefore, it is recommended to place these flush calls in easily identifiable regions of your code and outside of as many profiling timers as possible, such as immediately before or after writing a checkpoint.

Also, since flush calls will print multiple, similar looking outputs to `stdout`, it is also recommended to wrap any `BL_PROFILE_TINY_FLUSH();` calls in informative `amrex::Print()` lines to ensure accurate identification of each set of timers.

20.1.2 Full Profiling

If you set `PROFILE = TRUE` then a `bl_prof` directory will be written that contains detailed per-task timings for each processor. This will be written in `nfiles` files (where `nfiles` is specified by the user). The information in the directory can be analyzed by the *AMRProfParser* tool within *Amrvis*. In addition, an exclusive-only set of function timings will be written to `stdout`.

Trace Profiling

If you set `TRACE_PROFILE = TRUE` in addition to `PROFILE = TRUE`, then the profiler keeps track of when each profiled function is called and the `bl_prof` directory will include the function call stack. This is especially useful when core functions, such as `FillBoundary` can be called from many different regions of the code. Using trace profiling allows one to specify regions in the code that can be analyzed for profiling information independently from other regions.

Communication Profiling

If you set `COMM_PROFILE = TRUE` in addition to `PROFILE = TRUE`, then the `bl_prof` directory will contain additional information about MPI communication (point-to-point timings, data volume, barrier/reduction times, etc.). `TRACE_PROFILE = TRUE` and `COMM_PROFILE = TRUE` can be set together.

The AMReX-specific profiling tools are currently under development and this documentation will reflect the latest status in the development branch.

20.2 Instrumenting C++ Code

AMReX profiler objects are created and managed through `BL_PROF` macros.

To start, you must at least instrument main(), i.e.:

```
int main(...)
{
  amrex::Initialize(argc,argv);
  BL_PROFILE_VAR("main()",pmain);

  <AMReX code block>

  BL_PROFILE_VAR_STOP(pmain);
  amrex::Finalize();
}
```

Or:

```
void main_main()
{
    BL_PROFILE("main()");

    <AMReX code block>
}

int main(...)
```

(continues on next page)

(continued from previous page)

```
{
    amrex::Initialize(argc,argv);
    main_main();
    amrex::Finalize();
}
```

You can then instrument any of your functions, or code blocks. There are four general profiler macro types available:

20.2.1 1) A scoped timer, BL_PROFILE:

These timers generate their own object names, so they can't be controlled after being defined. However, they are the cleanest and easiest to work with in many situations. They time from the point where the macro is called until the end of the enclosing scope. This macro is ideal for timing an entire function. For example:

```
void YourClass::YourFunction()
{
  BL_PROFILE("YourClass::YourFunction()");   // Timer starts here.

  < Your Function Code Block>

}    // <------ Timer goes out of scope here, calling stop and returning the function␣
↪time.
```

Note that all AMReX timers are scoped and will call "stop" when the corresponding object is destroyed. This macro is unique because it can *only* stop when it goes out of scope.

20.2.2 2) A named, scoped timer, BL_PROFILE_VAR:

In some cases, using scopes to control a timer is not ideal. In such cases, you can use the _VAR_ macros to create a named timer that can be controlled through _START_ and _STOP_ macros. _VAR_ signifies that the macro takes a variable name. For example, to time a function without scoping:

```
BL_PROFILE_VAR("Flaten::FORT_FLATENX()", anyname);  // Create and start "anyname".
  FORT_FLATENX(arg1, arg2);
BL_PROFILE_VAR_STOP(anyname);   // Stop the "anyname" timer object.
```

This can also be used to selectively time with the same scope. For example, to include Func_0 and Func_2, but not Func_1:

```
BL_PROFILE_VAR("MyFuncs()", myfuncs);  // the first one
  MyFunc_0(args);
BL_PROFILE_VAR_STOP(myfuncs);

  MyFunc_1(args);

BL_PROFILE_VAR_START(myfuncs);
  MyFunc_2(arg);
BL_PROFILE_VAR_STOP(myfuncs);
```

Remember, these are still scoped. So, the scoped timer example can be reproduced exactly with named timers by just using the _VAR macro:

```
void YourClass::YourFunction()
{
  BL_PROFILE_VAR("YourClass::YourFunction()",  pmain);   // Timer starts here.

  < Your Function Code Block>

}    // <------ Timer goes out of scope here correctly, without a STOP call.
```

20.2.3 3) A named, scoped timer that doesn't auto-start, BL_PROFILE_VAR_NS:

Sometimes, a complicated scoping may mean the profiling object needs to be defined before it's started. To create a named AMReX timer that doesn't start automatically, use the _NS_ macros. ("NS" stands for "no start"). For example, this implementation times MyFunc0 and MyFunc1 but not any of the "Additional Code" blocks:

```
{
    BL_PROFILE_VAR_NS("MyFuncs()", myfuncs);  // dont start the timer

    <Additional Code A>

    {
       BL_PROFILE_VAR_START(myfuncs);
         MyFunc_0(arg);
       BL_PROFILE_VAR_STOP(myfuncs);
    }

    <Additional Code B>

    {
       BL_PROFILE_VAR_START(myfuncs);
         MyFunc_1(arg);
       BL_PROFILE_VAR_STOP(myfuncs);

       <Additional Code C>
    }
}
```

Note: The _NS_ macro must, by necessity, also be a _VAR_ macro. Otherwise, you would never be able to turn the timer on!

20.2.4 4) Designate a sub-region to profile, BL_PROFILE_REGION:

Often, it's helpful to look at a subset of timers separately from the complete profile. For example, you may want to view the timing of a specific time step or isolate everything inside the "Chemistry" part of the code. This can be accomplished by designating profile regions. All timers within a named region will be included both in the full analysis, as well as in a separate sub-analysis.

Regions are meant to be large contiguous blocks of code, and should be used sparingly and purposefully to produce useful profiling reports. As such, the possible region options are purposefully limited.

Scoped Regions

When using the Tiny Profiler, the only available region macro is the scoped macro. To create a region that profiles the *MyFuncs* code block, including all timers in the "Additional Code" regions, add macros in the following way:

```
{
    BL_PROFILE_REGION("MyFuncs");

    <Additional Code A>

    {
       BL_PROFILE("MyFunc0");

       MyFunc_0(arg);
    }

    <Additional Code B>

    {
       BL_PROFILE("MyFunc1");

       MyFunc_1(arg);
       <Additional Code C>
    }
}
```

The `MyFuncs` region appears in the Tiny Profiler output as an additional table. The following output example, mimics the above code. In it, the region is indicated by `REG::MyFuncs`.

```
BEGIN REGION MyFuncs

----------------------------------------------------------------
Name          NCalls  Excl. Min  Excl. Avg  Excl. Max   Max %
----------------------------------------------------------------
MyFunc0         1000      4.402      4.402      4.402  14.19%
MyFunc1         1000       4.39       4.39       4.39  14.15%
REG::MyFuncs    1000     0.0168     0.0168     0.0168   0.05%
----------------------------------------------------------------

----------------------------------------------------------------
Name          NCalls  Incl. Min  Incl. Avg  Incl. Max   Max %
----------------------------------------------------------------
REG::MyFuncs    1000      8.809      8.809      8.809  28.39%
MyFunc0         1000      4.402      4.402      4.402  14.19%
MyFunc1         1000       4.39       4.39       4.39  14.15%
----------------------------------------------------------------

END REGION MyFuncs
```

Named Regions

If using the Full Profiler, named region objects are also available. Named regions allow control of start and stop points without relying on scope. These macros use slightly modified _VAR_, _START_ and _STOP_ formatting. The first argument is the name, followed by the profile variable. Names for each section can differ, but because the profiler variable will be used to group the sections into a region, it must be the same. Consider the following example:

```
{
    BL_PROFILE_REGION_VAR("RegionAC",reg_ac);
    <Code Block A>
    BL_PROFILE_REGION_VAR_STOP("RegionAC", reg_ac);

    {

       MyFunc_0(arg);
    }

    BL_PROFILE_REGION_VAR("RegionB", reg_b)
    <Code Block B>
    BL_PROFILE_REGION_VAR_STOP("RegionB", reg_b);

    {

       MyFunc_1(arg);

       BL_PROFILE_REGION_VAR_START("SecondRegionAC", reg_ac);
       <Code Block C>
       BL_PROFILE_REGION_VAR_STOP("SecondRegionAC", reg_ac);
    }
}
```

Here, <Code Block A> and <Code Block C> are grouped into one region labeled "RegionAC" for profiling. <Code Block B> is isolated in its own group. Any timers inside MyFunc_0 and MyFunc_1 are not included in the region groupings.

20.3 Instrumenting Fortran90 Code

When using the full profiler, Fortran90 functions can also be instrumented with the following calls:

```
call bl_proffortfuncstart("my_function")
...
call bl_proffortfuncstop("my_function")
```

Note that the start and stop calls must be matched before leaving the scope of the corresponding start. Moreover, it is necessary to take into account all possible code paths. Therefore, you may need to add bl_proffortfuncstop in multiple locations, such as before any returns, at the end of the function and at the point in the function where you want to stop profiling. The profiling output will only warn of any bl_proffortfuncstart calls that were not stopped with bl_proffortfuncstop calls when in debug mode.

For functions with a high number of calls, there is a lighter-weight interface,

```
call bl_proffortfuncstart_int(n)
...
call bl_proffortfuncstop_int(n)
```

where n is an integer in the range [1,mFortProfsIntMaxFuncs]. mFortProfsIntMaxFuncs is currently set to 32. The profiled function will be named FORTFUNC_n in the profiler output, unless you rename it with BL_PROFILE_CHANGE_FORT_INT_NAME(fname, int) where fname is a std::string and int is the integer n in the bl_proffortfuncstart_int/bl_proffortfuncstop_int calls. BL_PROFILE_CHANGE_FORT_INT_NAME should be called in main().

Warning: Fortran functions cannot be profiled when using the Tiny Profiler. You will need to turn on the Full Profiler to receive the results from fortran instrumentation.

20.4 Profiling Options

AMReX's communication algorithms are often regions of code that increase in wall clock time when the application is load imbalanced, due to the MPI_Wait calls in these functions. To better understand if this is occuring and by how much, you can turn on an AMReX timed synchronization with the runtime variable: amrex.use_profiler_syncs=1 This adds named timers beginning with SyncBeforeComms immediately prior to the start of the FillBoundary, ParallelCopy and particle Redistribute functions, isolating any prior load imbalance to that timer before beginning the comm operation.

This is a diagnostic tool and may slow your code down, so it is not recommended to turn this on for production runs.

Note: Note: the SyncBeforeComms timer is not equal to your load imbalance. It only captures imbalance between the comm functions and the previous sync point; there may be other load imbalances captured elsewhere. Also, the timer reports in terms of MPI rank, so if the most imbalanced rank changes throughout the simulation, the timer will be an underestimation.

The effect on the communication timers may be more helpful: they will show the time to complete communications if there was no load imbalance. This means the difference between a case with and without this profiler sync may be a more useful metric for analysis.

20.5 AMRProfParser

AMRProfParser is a tool for processing and analyzing the bl_prof database. It is a command line application that can create performance summaries, plotfiles showing point-to-point communication and timelines, HTML call trees, communication call statistics, function timing graphs, and other data products. The parser's data services functionality can be called from an interactive environment such as *Amrvis*, from a sidecar for dynamic performance optimization, and from other utilities such as the command line version of the parser itself. It has been integrated into Amrvis for visual interpretation of the data allowing Amrvis to open the bl_prof database like a plotfile but with interfaces appropriate to profiling data. AMRProfParser and Amrvis can be run in parallel both interactively and in batch mode.

CHAPTER

TWENTYONE

EXTERNAL PROFILING TOOLS

AMReX is compatible with most commonly used profiling tools. This chapter provides some selected useful documentation on implementing a few of these tools on AMReX. For additional details on running these tools, please refer to the official documentation of the tools.

21.1 CrayPat

The profiling suite available on Cray XC systems is Cray Performance Measurement and Analysis Tools ("CrayPat")[1]. Most CrayPat functionality is supported for all compilers available in the Cray "programming environments" (modules which begin "`PrgEnv-`"); however, a few features, chiefly the "Reveal" tool, are supported only on applications compiled with Cray's compiler CCE[2][3].

CrayPat supports both high-level profiling tools, as well as fine-grained performance analysis, such as reading hardware counters. The default behavior uses sampling to identify the most time-consuming functions in an application.

21.1.1 High-level application profiling

The simplest way to obtain a high-level overview of an application's performance consists of the following steps:

1. Load the `perftools-base` module, then the `perftools-lite` module. (The modules will not work if loaded in the opposite order.)
2. Compile the application with the Cray compiler wrappers `cc`, `CC`, and/or `ftn`. This works with any of the compilers available in the `PrgEnv-` modules. E.g., on the Cori system at NERSC, one can use the Intel, GCC, or CCE compilers. No extra compiler flags are necessary in order for CrayPat to work. CrayPat instruments the application, so the `perftools-` modules must be loaded before one compiles the application.
3. Run the application as normal. No special flags are required. Upon application completion, CrayPat will write a few files to the directory from which the application was launched. The profiling database is a single file with the `.ap2` suffix.
4. One can query the database in many different ways using the `pat_report` command on the `.ap2` file. `pat_report` is available on login nodes, so the analysis need not be done on a compute node. Querying the database with no arguments to `pat_report` prints several different profiling reports to STDOUT, including a list of the most time-consuming regions in the application. The output of this command can be long, so it can be convenient to pipe the output to a pager or a file. A portion of the output from `pat_report <file>.ap2` is shown below:

[1] https://pubs.cray.com/content/S-2376/6.4.6/cray-performance-measurement-and-analysis-tools-user-guide-646-s-2376
[2] https://pubs.cray.com/content/S-2179/8.5/cray-c-and-c++-reference-manual-85
[3] https://pubs.cray.com/content/S-3901/8.5/cray-fortran-reference-manual-85

```
Table 1:  Profile by Function

  Samp% |    Samp |  Imb. |  Imb. | Group
        |         |  Samp | Samp% |  Function
        |         |       |       |   PE=HIDE

 100.0% | 5,235.5 |    -- |    -- | Total
|-------------------------------------------------------------------------------
|  50.2% | 2,628.5 |    -- |    -- | USER
||------------------------------------------------------------------------------
||   7.3% |   383.0 |  15.0 |  5.0% | eos_module_mp_iterate_ne_
||   5.7% |   300.8 | 138.2 | 42.0% | amrex_deposit_cic
||   5.1% |   265.2 |  79.8 | 30.8% | update_dm_particles
||   2.8% |   147.2 |   5.8 |  5.0% | fort_fab_setval
||   2.6% |   137.2 |  48.8 | 34.9% | amrex::ParticleContainer<>::Where
||   2.6% |   137.0 |  11.0 |  9.9% | ppm_module_mp_ppm_type1_
||   2.5% |   133.0 |  24.0 | 20.4% | eos_module_mp_nyx_eos_t_given_re_
||   2.1% |   107.8 |  33.2 | 31.4% | amrex::ParticleContainer<>::IncrementWithTotal
||   1.7% |    89.2 |  19.8 | 24.2% | f_rhs_
||   1.4% |    74.0 |   7.0 | 11.5% | riemannus_
||   1.1% |    56.0 |   2.0 |  4.6% | amrex::VisMF::Write
||   1.0% |    50.5 |   1.5 |  3.8% | amrex::VisMF::Header::CalculateMinMax
||===============================================================================
|  28.1% | 1,471.0 |    -- |    -- | ETC
||------------------------------------------------------------------------------
||   7.4% |   388.8 |  10.2 |  3.4% | __intel_mic_avx512f_memcpy
||   6.9% |   362.5 |  45.5 | 14.9% | CVode
||   3.1% |   164.5 |   8.5 |  6.6% | __libm_log10_l9
||   2.9% |   149.8 |  29.2 | 21.8% | _INTERNAL_25_______src_kmp_barrier_cpp_
↪5de9139b::__kmp_hyper_barrier_gather
||===============================================================================
|  16.8% |   879.8 |    -- |    -- | MPI
||------------------------------------------------------------------------------
||   5.1% |   266.0 | 123.0 | 42.2% | MPI_Allreduce
||   4.2% |   218.2 | 104.8 | 43.2% | MPI_Waitall
||   2.9% |   151.8 |  78.2 | 45.4% | MPI_Bcast
||   2.6% |   135.0 |  98.0 | 56.1% | MPI_Barrier
||   2.0% |   105.8 |   5.2 |  6.3% | MPI_Recv
||===============================================================================
|   1.9% |    98.2 |    -- |    -- | IO
||------------------------------------------------------------------------------
||   1.8% |    93.8 |   6.2 |  8.3% | read
||===============================================================================
```

21.2 IPM - Cross-Platform Integrated Performance Monitoring

IPM provides portable profiling capabilities across HPC platforms, including support on selected Cray and IBM machines (cori and (TODO: verify it works on) summit). Running an IPM instrumented binary generates a summary of number of calls and time spent on MPI communication library functions. In addition, hardware performance counters can also be collected through PAPI.

Detailed instructions can be found at[4] and[5].

21.2.1 Building with IPM on cori

Steps:

1. Run module load ipm.
2. Build code as normal with make.
3. Re-run the link command (e.g. cut-and-paste) with `$IPM` added to the end of the line.

21.2.2 Running with IPM on cori

1. Set environment variables: `export IPM_REPORT=full IPM_LOG=full IPM_LOGDIR= <dir>`
2. Results will be printed to stdout and an xml file generated in the directory specified by `IPM_LOGDIR`.
3. Post-process the xml with `ipm_parse -html <xmlfile>`, which produces an directory with html.

21.2.3 Summary MPI Profile

Example MPI profile output:

```
##IPMv2.0.5############################################################
#
# command   : /global/cscratch1/sd/cchan2/projects/lbl/BoxLib/Tests/LinearSolvers/C_
↪CellMG/./main3d.intel.MPI.OMP.ex.ipm inputs.3d.256oo
# start     : Tue Aug 15 17:34:23 2017   host      : nid11311
# stop      : Tue Aug 15 17:34:35 2017   wallclock : 11.54
# mpi_tasks : 128 on 32 nodes            %comm     : 32.51
# mem [GB]  : 126.47                     gflop/sec : 0.00
#
#           :       [total]        <avg>          min          max
# wallclock :       1188.42         9.28         8.73        11.54
# MPI       :        386.31         3.02         2.51         4.78
# %wall     :
#   MPI     :                      32.52        24.36        41.44
# #calls    :
#   MPI     :       5031172        39306        23067        57189
# mem [GB]  :        126.47         0.99         0.98         1.00
#
#                             [time]       [count]       <%wall>
# MPI_Allreduce               225.72        567552         18.99
```

(continues on next page)

[4] http://ipm-hpc.sourceforge.net/userguide.html
[5] https://www.nersc.gov/users/software/performance-and-debugging-tools/ipm/

(continued from previous page)

```
# MPI_Waitall                92.84        397056          7.81
# MPI_Recv                   29.36           193          2.47
# MPI_Isend                  25.04       2031810          2.11
# MPI_Irecv                   4.35       2031810          0.37
# MPI_Allgather               2.60           128          0.22
# MPI_Barrier                 2.24           512          0.19
# MPI_Gatherv                 1.70           128          0.14
# MPI_Comm_dup                1.23           256          0.10
# MPI_Bcast                   1.14           256          0.10
# MPI_Send                    0.06           319          0.01
# MPI_Reduce                  0.02           128          0.00
# MPI_Comm_free               0.01           128          0.00
# MPI_Comm_group              0.00           128          0.00
# MPI_Comm_size               0.00           256          0.00
# MPI_Comm_rank               0.00           256          0.00
# MPI_Init                    0.00           128          0.00
# MPI_Finalize                0.00           128          0.00
```

The total, average, minimum, and maximum wallclock and MPI times across ranks is shown. The memory footprint is also collected. Finally, results include number of calls and total time spent in each type of MPI call.

21.2.4 PAPI Performance Counters

To collect performance counters, set `IPM_HPM=<list>`, where the list is a comma-separated list of PAPI counters. For example: `export IPM_HPM=PAPI_L2_TCA,PAPI_L2_TCM`.

For reference, here is the list of available counters on cori, which can be found by running `papi_avail`:

```
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_TLB_DM  0x80000014  Yes   No   Data translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_STL_ICY 0x80000025  Yes   No   Cycles with no instruction issue
PAPI_BR_UCN  0x8000002a  Yes   Yes  Unconditional branch instructions
PAPI_BR_CN   0x8000002b  Yes   No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  Yes   No   Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  Yes   Yes  Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_RES_STL 0x80000039  Yes   No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_LST_INS 0x8000003c  Yes   Yes  Load/store instructions completed
PAPI_L1_DCA  0x80000040  Yes   Yes  Level 1 data cache accesses
PAPI_L1_ICH  0x80000049  Yes   No   Level 1 instruction cache hits
PAPI_L1_ICA  0x8000004c  Yes   No   Level 1 instruction cache accesses
```

(continues on next page)

(continued from previous page)

```
PAPI_L2_TCH  0x80000056  Yes   Yes  Level 2 total cache hits
PAPI_L2_TCA  0x80000059  Yes   No   Level 2 total cache accesses
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
```

Due to hardware limitations, there is a limit to which counters can be collected simultaneously in a single run. Some counters may map to the same registers and thus cannot be collected at the same time.

21.2.5 Example HTML Performance Summary

Running `ipm_parse -html <xmlfile>` on the generated xml file will produce an HTML document that includes summary performance numbers and automatically generated figures. Some examples are shown here.

6470883

- Load Balance
- Communication Balance
- Message Buffer Sizes
- Communication Topology
- Switch Traffic
- Memory Usage
- Executable Info
- Host List
- Environment
- Developer Info

powered by IPM

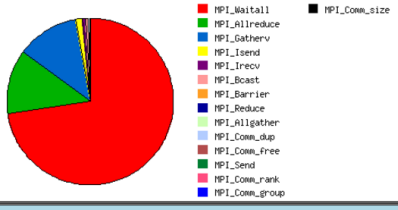
command: /global/cscratch1/sd/cchan2/projects/lbl/BoxLib/Tests/LinearSolvers/C_CellMG//./main3d.intel.MPI.OMP.ex.ipm inputs.3d.15456			
codename:	unknown	state:	unknown
username:	cchan2	group:	
host:	nid04720 (x86_64_Linux)	mpi_tasks:	4 on 1 hosts
start:	08/22/17/12:04:29	wallclock:	5.20833e+01 sec
stop:	08/22/17/12:05:20	%comm:	4.99170175468912
total memory:	28.82264 gbytes	total gflop/sec:	0
switch(send):	0 gbytes	switch(recv):	0 gbytes

Computation

Event	Count	Pop
PAPI_L2_TCA	1884011012	*
PAPI_L2_TCM	447288209	*

Communication

% of MPI Time



HPM Counter Statistics

Event	Ntasks	Avg	Min(rank)	Max(rank)
PAPI_L2_TCA	*	471002753.00	418954739 (2)	558450816 (3)
PAPI_L2_TCM	*	111822052.25	83963410 (2)	172440952 (3)

Communication Event Statistics (100.00% detail, 6.9785e-06 error)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Waitall	0	5360	5.774	1.788e-05	2.530e-02	55.53	2.77
MPI_Gatherv	114688	4	1.257	4.390e-01	4.390e-01	12.09	0.60
MPI_Allreduce	16	1908	0.684	1.288e-05	1.238e-02	6.58	0.33
MPI_Allreduce	8	10900	0.611	6.914e-06	3.187e-03	5.88	0.29
MPI_Waitall	4194304	302	0.435	1.044e-03	9.893e-03	4.18	0.21
MPI_Waitall	3670016	151	0.338	8.841e-04	1.869e-01	3.25	0.16
MPI_Waitall	2621440	151	0.176	8.039e-04	2.057e-02	1.69	0.08
MPI_Waitall	917504	135	0.160	3.550e-04	4.294e-02	1.54	0.08
MPI_Waitall	1048576	270	0.157	2.749e-04	3.719e-03	1.51	0.08
MPI_Waitall	65536	1838	0.112	5.198e-05	4.921e-04	1.07	0.05
MPI_Waitall	57344	919	0.095	5.293e-05	1.865e-02	0.92	0.05
MPI_Waitall	655360	135	0.082	2.880e-04	1.567e-02	0.79	0.04
MPI_Waitall	229376	135	0.066	1.240e-04	2.626e-02	0.64	0.03
MPI_Waitall	262144	270	0.063	1.280e-04	1.812e-03	0.61	0.03
MPI_Waitall	40960	919	0.062	4.482e-05	1.561e-02	0.60	0.03
MPI_Bcast	320	4	0.049	1.639e-02	1.639e-02	0.47	0.02
MPI_Waitall	163840	135	0.043	9.584e-05	1.981e-02	0.41	0.02
MPI_Isend	20480	3676	0.032	1.907e-06	9.990e-05	0.31	0.02
MPI_Barrier	0	16	0.027	4.540e-04	5.476e-03	0.26	0.01

Fig. 21.1: Sample performance summary generated by IPM

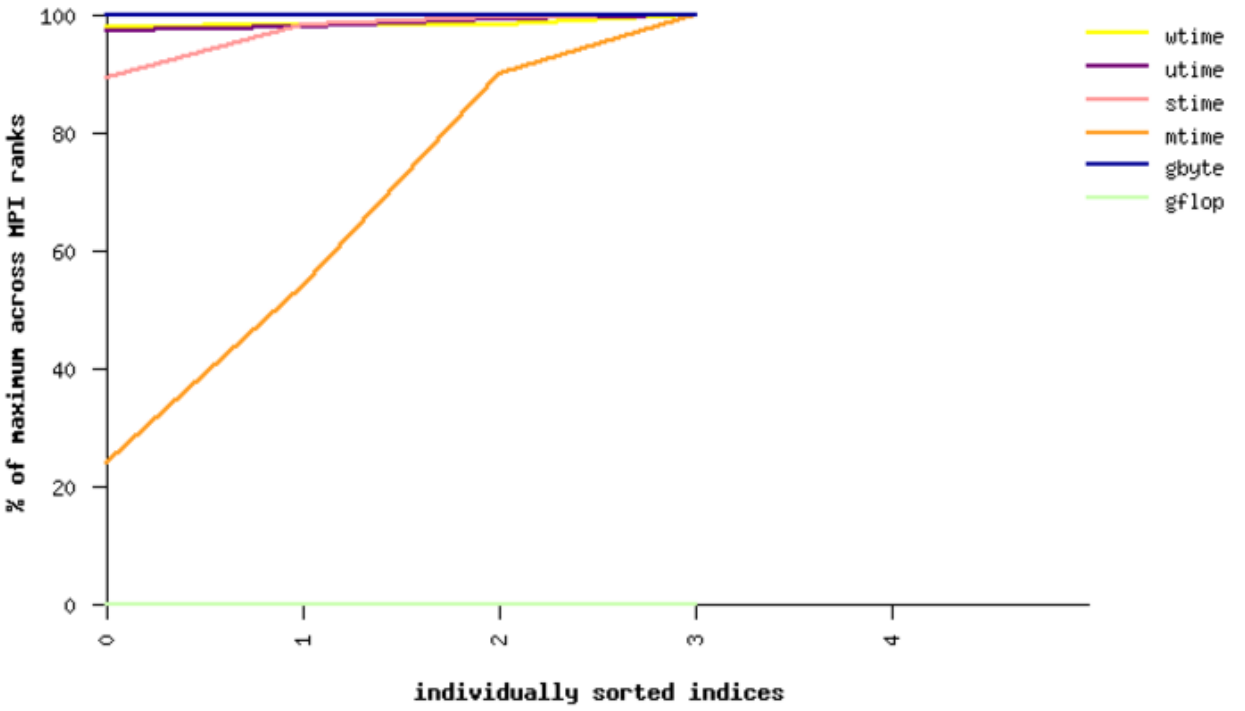
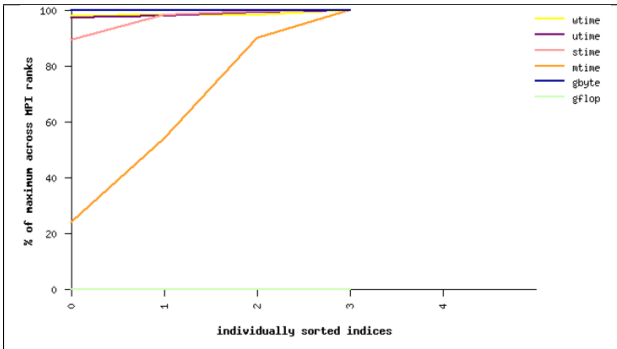
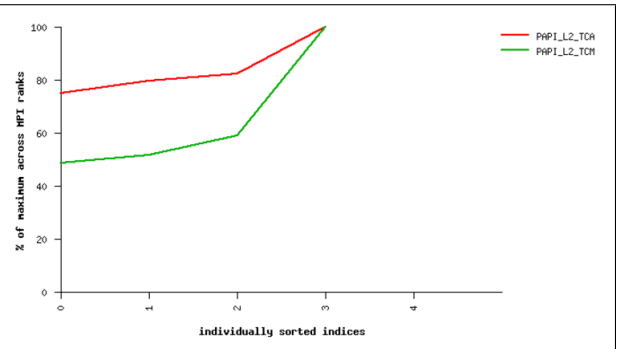
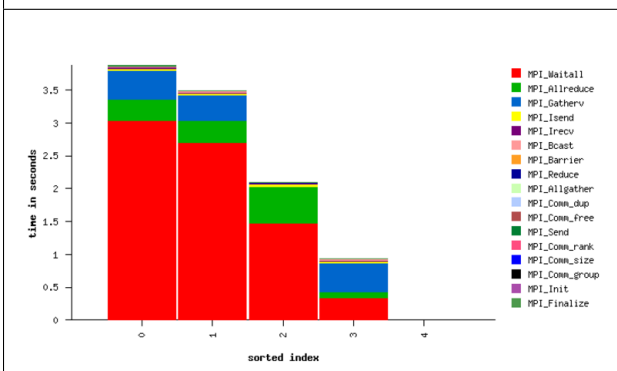
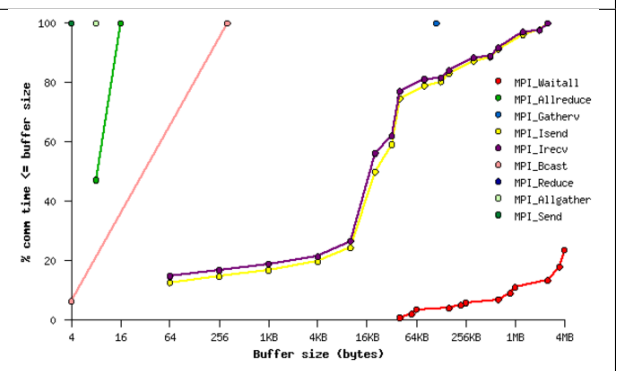


Table 21.1: Example of performance graphs generated by IPM

	
Timings	PAPI Counters
	
MPI Time by Function	MPI Time by Message Size
	

21.3 Nsight Systems

The Nsight Systems tool provides a high-level overview of your code, displaying the kernel launches, API calls, NVTX regions and more in a timeline for a clear, visual picture of the overall runtime patterns. It analyzes CPU-codes or CUDA-based GPU codes and is available on Summit and Cori in a system module.

Nsight Systems provides a variety of profiling options. This documentation will cover the most commonly used options for AMReX users to keep track of useful flags and analysis patterns. For the complete details of using Nsight Systems, refer to the Nsight Systems official documentation.

21.3.1 Profile Analysis

The most common use case of Nsight Systems for AMReX users is the creation of a qdrep file that is viewed in the Nsight Systems GUI, typically on a local workstation or machine.

To generate a qdrep file, run nsys with the `-o` option:

```
nsys profile -o <file_name> ${EXE} ${INPUTS}
```

AMReX's lambda-based launch system often makes these timelines difficult to parse, as the kernel are mangled and are difficult to decipher. AMReX's Tiny Profiler includes NVTX region markers, which can be used to mark the respective section of the Nsight Systems timeline. To include AMReX's built-in Tiny Profiler NVTX regions in Nsight Systems outputs, compile AMReX with `TINY_PROFILE=TRUE`.

Nsight Systems timelines only profile a single, contiguous block of time. There are a variety of methods to specify the specific region you would like to analyze. The most common options that AMReX users may find helpful are:

1. **Specify an NVTX region as the starting point of the analysis.**

This is done using `-c nvtx -p "region_name@*" -e NSYS_NVTX_PROFILER_REGISTER_ONLY=0`, where `region_name` is the identification string for the of the NVTX region. The additional environment variable, `-e ...` is needed because AMReX's NVTX region names currently do not use a registered string. TinyProfiler's built-in NVTX regions use the same identification string as the timer itself. For example, to start an analysis at the `do_hydro` NVTX region, run:

```
nsys profile -o <file_name> -c nvtx -p "do_hydro@*" -e NSYS_NVTX_PROFILER_REGISTER_
↪ONLY=0 ${EXE} ${INPUTS}
```

This will profile from the first instance of the specified NVTX region until the end of the application. In AMReX applications, this can be helpful to skip initialization and analyze the remainder of the code. To only analyze the specified NVTX region, add the flag `-x true`, which will end the analysis at the end of the region:

```
nsys profile -o <file_name> -c nvtx -p "do_hydro@*" -x true -e NSYS_NVTX_PROFILER_
↪REGISTER_ONLY=0 ${EXE} ${INPUTS}
```

Again, it's important to remember that Nsight Systems only analyzes a single contiguous block of time. So, this will only give you a profile for the first instance of the named region. Plan your Nsight System analyses accordingly.

2. **Specify a region with cuda profiler function calls.**

This requires manually altering your source code, but can provide better specificity in what you analyze. Directly insert `cudaProfilerStart\Stop` around the region of code you want to analyze:

```
cudaProfilerStart();

// CODE TO PROFILE
```

(continues on next page)

(continued from previous page)

```
cudaProfilerStop();
```

Then, run with `-c cudaProfilerApi`:

```
nsys profile -o <file_name> -c cudaProfilerApi ${EXE} ${INPUTS}
```

As with NVTX regions, Nsight Systems will only profile from the first call to `cudaProfilerStart()` to the first call to `cudaProfilerStop()`, so be sure to add these markers appropriately.

21.3.2 Nsight Systems GUI Tips

- When analyzing an AMReX application in the Nsight Systems GUI using NVTX regions or `TINY_PROFILE=TRUE`, AMReX users may find it useful to turn on the feature "Rename CUDA Kernels by NVTX". This will change the CUDA kernel names to match the inner-most NVTX region in which they were launched instead of the typical mangled compiler name. This will make identifying AMReX CUDA kernels in Nsight Systems reports considerably easier.

  This feature can be found in the GUI's drop down menu, under:

```
Tools -> Options -> Environment -> Rename CUDA Kernels by NVTX.
```

21.4 Nsight Compute

The Nsight Compute tool provides a detailed, fine-grained analysis of your CUDA kernels, giving details about the kernel launch, occupancy, and limitations while suggesting possible improvements to maximize the use of the GPU. It analyzes CUDA-based GPU codes and is available on Summit and Cori in system modules.

Nsight Compute provides a variety of profiling options. This documentation will focus on the most commonly used options for AMReX users, primarily to keep track of useful flags and analysis patterns. For the complete details of using Nsight Compute, refer to the Nsight compute official documentation.

21.4.1 Kernel Analysis

The standard way to run Nsight Compute on an AMReX application is to specify an output file that will be transferred to a local workstation of machine for viewing in the Nsight Compute GUI. Nsight Compute can be told to return a report file using the `-o` flag. In addition, when running with Nsight compute on an AMReX application, it is important to turn off the floating point exception trap, as it causes a runtime error. So, an entire AMReX application can be analyzed with Nsight Compute by running:

```
ncu -o <file_name> ${EXE} ${INPUTS} amrex.fpe_trap_invalid=0
```

However, this implementation should almost never used by AMReX applications, as the analysis of every kernel would be extremely lengthy and unnecessary. To analyze a desired subset of CUDA kernels, AMReX users can use the Tiny Profiler's built-in NVTX regions to narrow the scope of the analysis. Nsight Compute allows users to specify which NVTX regions to include and exclude through the `--nvtx`, `--nvtx-include` and `--nvtx-exclude` flags. For example:

```
ncu --nvtx --nvtx-include "Hydro()" --nvtx-exclude "StencilA(),StencilC()" -o kernels $
↪{EXE} ${INPUTS} amrex.fpe_trap_invalid=0
```

will return a file named `kernels` which contains an analysis of the CUDA kernels launched inside the `Hydro()` region, ignoring any kernels launched inside `StencilA()` and `StencilC()`. When using the NVTX regions built into AMReX's TinyProfiler, be aware that the application must be built with `TINY_PROFILE=TRUE` and the NVTX region names are identical to the TinyProfiler timer names.

Another helpful flag for selecting a reasonable subset of kernels for analysis is the `-c` option. This flag specifies the total number of kernels to be analyzed. For example:

```
ncu --nvtx --nvtx-include "GravitySolve()" -c 10 -o kernels ${EXE} ${INPUTS} amrex.fpe_
↪trap_invalid=0
```

will only analyze the first ten kernels inside of the `GravitySolve()` NVTX region.

For further details on how to choose a subset of CUDA kernels to analyze, or to run a more detailed analysis, including CUDA hardware counters, refer to the Nsight Compute official documentation on NVTX Filtering.

21.4.2 Roofline

As of version 2020.1.0, Nsight Compute has added the capability to perform roofline analyses on CUDA kernels to describe how well a given kernel is running on a given NVIDIA architecture. For details on the roofline capabilities in Nsight Compute, refer to the NVIDIA Kernel Profiling Guide.

To run a roofline analysis on an AMReX application, run `ncu` with the flag `--section SpeedOfLight_RooflineChart`. Again, using appropriate NVTX flags to limit the scope of the analysis will be critical to achieve results within a reasonable time. For example:

```
ncu --section SpeedOfLight_RooflineChart --nvtx --nvtx-include "MLMG()" -c 10 -o␣
↪roofline ${EXE} ${INPUTS} amrex.fpe_trap_invalid=0
```

will perform a roofline analysis of the first ten kernels inside of the region `MLMG()`, and report their relative performance in the file `roofline`, which can be read by the Nsight Compute GUI.

For further information on the roofline model, refer to the scientific literature, Wikipedia overview, NERSC documentation and tutorials.

CHAPTER

TWENTYTWO

EXTERNAL FRAMEWORKS

22.1 SUNDIALS

SUNDIALS stands for **SU**ite of **N**onlinear and **DI**fferential/**AL**gebraic equation **S**olvers. It consistes of the following six solvers:

- CVODE, for initial value problems for ODE systems
- CVODES, solves ODE systems and includes sensitivity analysis
- ARKODE, solves initial value ODE problems with Runge-Kutta methods
- IDA, solves initial value problems for differential-algebraic equation systems
- IDAS, solves differential-algebraic equation systems and includes sensitivity analysis
- KINSOL, solves nonlinear algebraic systems

AMReX provides interfaces to the SUNDIALS suite. For time integration, users can refer to the section *Using SUNDIALS* for more information. In addition, an example code demonstrating time integration with SUNDIALS can be found in the tutorials at, SUNDIALS and Time Integrators

For more information on SUNDIALS please see their readthedocs page.

22.2 SWFFT

`hacc/SWFFT`, developed by Adrian Pope et al. at Argonne National Lab, provides the functionality to perform forward and reverse Fast Fourier Transforms (FFT) within a fully parallelized framework built in C++ and F90. In the words of HACC's developers, SWFFT is a "distributed-memory, pencil-decomposed, parallel 3D FFT."[1] The SWFFT source code is also contained in the following directory within AMReX: `amrex/Src/Extern/SWFFT`.[2]

[1] https://git.cels.anl.gov/hacc/SWFFT

[2] SWFFT source code directory in AMReX: amrex/Src/Extern/SWFFT

22.2.1 Pencil Redistribution

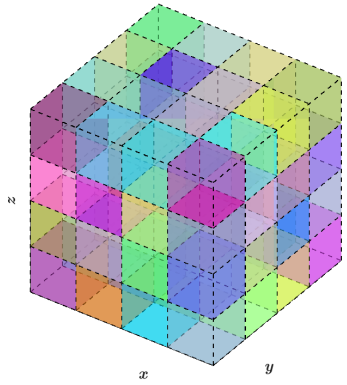
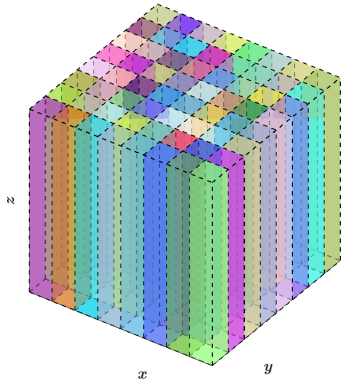
As input, SWFFT takes three-dimensional arrays of data distributed across block-structured grids, and redistributes the data into "pencil" grids in z, x, and then y, belonging to different MPI processes. After each pencil conversion, a 1D FFT is performed on the data along the pencil direction using calls to the FFTW[3] library. The `README` files in the tutorial directories specify the relationship between the number of grids and the number of MPI processes that should be used. The `hacc/SWFFT` `README` document by Adrian Pope et al. explains restrictions on grid dimensions in relation to the number of MPI processes[Page 219, 1Page 219, 2]:

> [...] A rule of thumb is that [SWFFT] generally works when the number of vertices along one side of the global 3D grid ("ng") can be factored into small primes, and when the number of MPI ranks can also be factored into small primes. I believe that all of the unique prime factors of the number of MPI ranks must be present in the set of prime factors of the grid, eg. if you have 20 MPI ranks then ng must be a multiple of 5 and 2. The `CheckDecomposition` utility is provided to check (on one rank) whether a proposed grid size and number of MPI ranks will work, which can be done before submitting a large test with `TestDfft/TestFDfft`.

The relationship between the number of processes versus global grid dimensions is determined by how the total number of grids can be factored from a three dimensional grid structure (block structured grids) into a two dimensional structure (pencil arrays), as shown in the figures below.

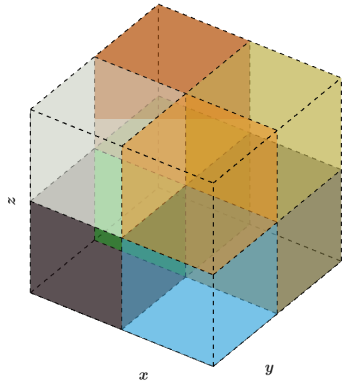
The following figures illustrate how data is distributed from block structured grids to pencil arrays within SWFFT, where the colors of each box indicate which MPI rank it belongs to:

Table 22.1: SWFFT Redistribution from $4 \times 4 \times 4$ Box Array into Pencils

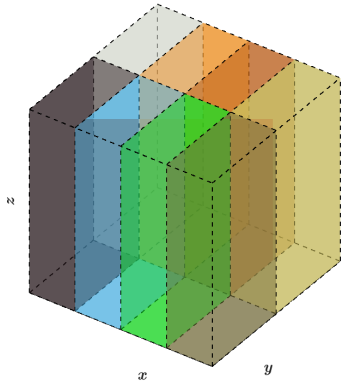
	
(a) Block structured grids: $N_x = 4, N_y = 4, N_z = 4$	(b) Z-pencils: $N_x = 8, N_y = 8, N_z = 1$

[3] http://www.fftw.org/

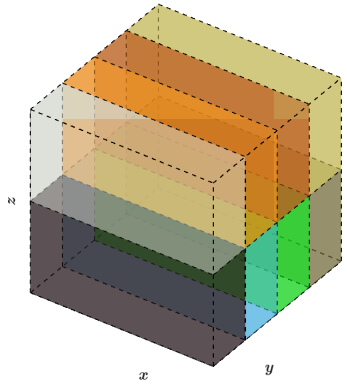
Table 22.2: SWFFT Redistribution from $2 \times 2 \times 2$ Box Array into Pencils



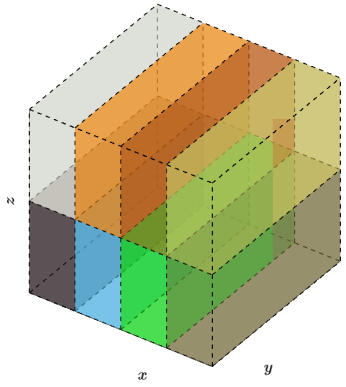
(a) Block structured grids: $N_x = 2, N_y = 2, N_z = 2$



(b) Z-pencils: $N_x = 4, N_y = 2, N_z = 1$



(c) X-pencils: $N_x = 1, N_y = 4, N_z = 2$



(d) Y-pencils: $N_x = 4, N_y = 1, N_z = 2$

Using the same number of AMReX grids as processes has been verified to work in the SWFFT Poisson and SWFFT Simple tutorials. This can be illustrated by the following equation for the total number of grids, N_b, in a regularly structured domain:

$$N_b = m_{bi} m_{bj} = n_{bi} n_{bj} n_{bk},$$

where n_{bi}, n_{bj}, and n_{bk} are the number of grids, or boxes, in the x, y, and z dimensions of the block-structured grid. Analogously, for pencil distributions, m_{bi} and m_{bj} are the number of grids along the remaining dimensions if pencils are taken in the k direction. There are many possible ways of redistributing the data, for example $m_{bi} = n_{bi} n_{bk}$ & $m_{bj} = n_{bj}$ is one possible simple configuration. However, it is evident from the figures above that the SWFFT redistribution algorithm has a more sophisticated method for finding the prime factors of the grid.

22.2.2 Tutorials

AMReX contains two SWFFT tutorials, SWFFT Poisson and SWFFT Simple:

- 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 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.

CHAPTER

TWENTYTHREE

REGRESSION TESTING

23.1 Continuous Compilation Testing

As a first line of testing, on every commit to the repository, we verify that we can compile AMReX as a library for a common set of configuration options. This operation is performed through Travis-CI. This layer of testing is deliberately limited, so that it can be run quickly on every commit. For more extensive testing, we rely on the nightly regression results.

23.2 Nightly Regression Testing

Each night, we automatically run a suite of tests, both on AMReX itself, and on a most of the major application codes that use it as a framework. We use an in-house test runner script to manage this operation, originally developed by Michael Zingale for the Castro code, and later expanded to other application codes as well. The results for each night are collected and stored on a web page; see https://ccse.lbl.gov/pub/RegressionTesting/ for the latest set of results. The runtime option `amrex.abort_on_unused_inputs` (`0` or `1`; default is `0` for false) is useful for making sure that tests always stay up to date with API changes as it will abort the application after the test run if any unused input parameters were detected.

23.3 Running the test suite locally

The test suite is mostly used internally by AMReX developers. However, if you are making a pull request to AMReX, it can be useful to run the test suite on your local machine to reduce the liklihood that your changes break some existing functionality. To run the test suite locally, you must first obtain a copy of the test runner source, available on Github here: https://github.com/AMReX-Codes/regression_testing. The test runner requires Python version 2.7 or greater. Additional information on the test suite software can be found at, https://amrex-codes.github.io/regression_testing/.

After obtaining the code, you will need a configuration file that defines which tests to run, which amrex repository to test, which branch to use, etc. A sample configuration file for AMReX is distributed with the amrex source code at `amrex/Tools/RegressionTesting/AMReX-tests.ini`. You will need to modify a few of the entries to, for example, point the test runner to the clone of amrex on your local machine. Entries you will likely want to change include:

```
testTopDir = /path/to/test/output # the tests results and benchmarks will stored here
webTopDir  = /path/to/web/output  # a web page with the test results will be written here
```

to control where the generated output will be written, and

```
[AMReX]
dir = /path/to/amrex  # the path to the amrex repository you want to test
branch = "development"
```

to control which repository and branch to test.

The test runner is a Python script and can be invoked like so:

```
python regtest.py <options> AMReX-tests.ini
```

Before you can use it, you must first generate a set of "benchmarks" - i.e. known "good" answers to the tests that will be run. If you are testing a pull request, you can generate these by running the script with a recent version of the `development` branch of AMReX. You can generate the benchmarks like so:

```
python regtest.py --make_benchmarks 'generating initial benchmarks' AMReX-tests.ini
```

Once that is finished, you can switch over to the branch you want to test in `AMReX-tests.ini`, and then re-run the script without the `--make_benchmarks` option:

```
python regtest.py AMReX-tests.ini
```

The script will generate a set of html pages in the directory specified in your `AMReX-tests.ini` file that you can examine using the browser of your choice.

For a complete set of script options, run

```
python regtest.py --help
```

A particularly useful option lets you run just a subset of the complete test suite. To run only one test, you can do:

```
python regtest.py --single_test <TestName> AMReX-tests.ini
```

To run an enumerated list of tests, do:

```
python regtest.py --tests '<TestName1> <TestName2> <TestName3>' AMReX-tests.ini
```

23.4 Adding a new test

New tests can be added to the suite by modifying the `AMReX-tests.ini` file. The easiest thing to do is start from an existing test and modify it. For example, this entry:

```
[MLMG_FI_PoisCom]
buildDir = Tests/LinearSolvers/ABecLaplacian_F
inputFile = inputs-rt-poisson-com
dim = 3
restartTest = 0
useMPI = 1
numprocs = 2
useOMP = 1
numthreads = 3
compileTest = 0
doVis = 0
```

(continues on next page)

(continued from previous page)

```
outputFile = plot
testSrcTree = C_Src
```

defines a test called `MLMG_FI_PoisCom` by specifying the appropriate build directory, inputs file, and a set of configuration options. The above options are the most commonly changed; for a full list of options, see the example configuration file at https://github.com/AMReX-Codes/regression_testing/blob/main/example-tests.ini.

CHAPTER

TWENTYFOUR

FREQUENTLY ASKED QUESTIONS

Q. Why am I getting a segmentation fault after my code runs?

A. Do you have `amrex::Initialize(); {` and `} amrex::Finalize();` at the beginning and end of your code? For all AMReX commands to function properly, including to release resources, they need to be contained between these two curly braces or in a separate function. In the Initialize and Finalize section, these commands are discussed further detail.

Q. I want to use a different compiler with GNU Make to compile AMReX. How do I do this?

A. In the file `amrex/Tools/GNUMake/Make.local` you can specify your own compile commands by setting the variables `CXX`, `CC`, `FC`, and `F90`. An example can be found at Specifying your own compiler . Additional customizations are described in the file, `amrex/Tools/GNUMake/Make.local.template`. In the same directory, `amrex/Tools/GNUMake/README.md` contains detailed information on compiler commands.

Q. I'm having trouble compiling my code.

A. AMReX developers have found that running the command `make clean` can resolve many compilation issues.

If you are working in an environment that uses a module system, please ensure you have the correct modules loaded. Typically, to do this, type `module list` at the command prompt.

Q. When I profile my code that uses GPUs with `TINY_PROFILE=TRUE` or `PROFILE=TRUE` my timings are inconsistent.

A. Due to the asynchronous nature of GPU execution, profilers might only measure the run time on CPU, if there is no explicit synchronization. For `TINY_PROFILE`, one could use `ParmParse` parameter `tiny_profiler.device_synchronize_around_region=1` to add synchronization. Note that this may degrade performance.

Q. How do I know I am getting the right answer?

A. AMReX provides support for verifying output with several tools. To briefly mention a few:

- The `print_state` function can be used to output the data of a single cell.
- `VisMF::Write` can be used to write MultiFab data to disk that can be viewed with Amrvis.
- `amrex::Print()` and `amrex::AllPrint()` are useful for printing output when using multiple processes or threads as it prevents messages from getting mixed up.
- fcompare compares two plotfiles and reports absolute and relative error.

Additional tools and discussion on this topic is contained in the section Debugging.

Q. What's the difference between `Copy` and `ParallelCopy` for `MultiFab` data?

A. `MultiFab::Copy` is for two `MultiFabs` built with the same `BoxArray` and `DistributionMapping`, whereas `ParallelCopy` is for parallel communication of two `MultiFabs` with different `BoxArray` and/or `DistributionMapping`.

Q. How do I fill ghost cells?

A. See Ghost Cells in the AMReX Source Documentation.

Q. What's the difference between `AmrCore` and `AmrLevel`? How do I decide which to use?

A. The `AmrLevel` class is an abstract base class that holds data for a single AMR level. A vector of `AmrLevel` is stored in the `Amr` class, which is derived from `AmrCore`. An application code can derive from `AmrLevel` and override functions. `AmrCore` contains the meta-data for the AMR hierarchy, but it does not contain any floating point mesh data. Instead of using `Amr`/`AmrLevel`, an application can also derive from `AmrCore`. If you want flexibility, you might choose the `AmrCore` approach, otherwise the `AmrLevel` approach might be easier because it already has a lot of built-in capabilities that are common for AMR applications.

Q. For GPU usage, how can I perform explicit host to device and device to host copies without relying on managed memory?

A. Use `The_Pinned_Arena()` (See Memory Allocation in the AMReX Source Documentation.) and

```
void htod_memcpy (void* p_d, const void* p_h, const std::size_t sz);
void dtoh_memcpy (void* p_h, const void* p_d, const std::size_t sz);
```

(continues on next page)

(continued from previous page)

```
void dtoh_memcpy (FabArray<FAB>& dst, FabArray<FAB> const& src, int scomp, int dcomp,␣
↪int ncomp);
void htod_memcpy (FabArray<FAB>& dst, FabArray<FAB> const& src, int scomp, int dcomp,␣
↪int ncomp);
```

Q. How do I generate random numbers with AMReX? Can I set the seed? Are they thread safe with MPI and OpenMP?

A. (Thread safe) Yes, `amrex::Random()` is thread safe. When OpenMP is on, each thread will have its own dedicated Random Number Generator that is totally independent of the others.

Q. Is Dirichlet boundary condition data loaded into cell-centered, or face-centered containers? How is it used in AMReX-based codes like MLMG and the advection routines in AMReX-Hydro?

A. In the cell-centered MLMG solver, the Dirichlet boundary data are stored in containers that have the information of the location of the data.

Q. How does coarse-grained OpenMP parallelism work in AMReX? How is it different from the fine-grained approach?

A. Our OpenMP strategy is explained in this paper, https://arxiv.org/abs/1604.03570.

Q. How to avoid running into `Formal parameter space overflowed` CUDA error while building complex EB geometries using AMReX implicit functions and CSG functionalities ?

A. AMReX enables logical operations and transformations to assemble basic shapes Implicit Functions into complex geometries. Each operation results in a more complex type which can eventually overflow the parameter space (4096 bytes on CUDA 11.4 for instance). To circumvent the problem, explicitly copy the object to the device and pass a device pointer function object *DevicePtrIF* into the *EB2* function using:

```
using IF_t = decltype(myComplexIF);
IF_t* dp = (IF_t*)The_Arena()->alloc(sizeof(myComplexIF));
Gpu::htod_memcpy_async(dp, &myComplexIF, sizeof(IF_t));
Gpu::streamSynchronize();
EB2::DevicePtrIF<IF_t> dp_myComplexIF{dp};
auto gshop = EB2::makeShop(dp_myComplexIF);
```

24.1 More Questions

If your question was not addressed here, you are encouraged to search and ask for help on the AMReX GitHub Discussions page.

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

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