amrex Documentation

Release 19.03-dev

AMReX Team

Feb 18, 2019

CONTENTS:

AMReX is a software framework library containing all the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications. AMReX is freely available 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. Changes are merged into the master branch 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 an issue on the AMReX GitHub webpage.

There are small stand-alone example codes that demonstrate how to use different parts of the AMReX functionality; there is extensive documentation for these tutorial codes at AMReX Tutorials

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, 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 of 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 merged into the master branch on a monthly basis. The master branch is considered the release branch. The releases are tagged 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).

2.2 Example: Hello World

The source code of this example is at `amrex/Tutorials/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/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/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,

```
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
```

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
Hello world from AMReX version 17.05-30-g5775aed933c4-dirty
```

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. 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 amrex.v=1
```

The result may look like,

```
OMP initialized with 4 OMP threads
Hello world from AMReX version 17.06-287-g51875485fe51-dirty
```

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
Hello world from AMReX version 17.06-287-g51875485fe51-dirty
```

2.3 Example: Heat Equation Solver

We now look at a more complicated example at `amrex/Tutorials/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 section *Example: HeatEquation_EX1_C*. For now let's just build and run the code, and visualizae the results.

2.3.1 Building and Running the Code

To build a 2D executable, go to `amrex/Tutorials/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.

CHAPTER

THREE

BUILDING AMREX

In this chapter, we discuss AMReX's build systems. There are three ways to use AMReX. 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 install AMReX as a library (*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*.

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/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
DEBUG	TRUE or FALSE	TRUE
DIM	1 or 2 or 3	none
USE_MPI	TRUE or FALSE	FALSE
USE_OMP	TRUE or FALSE	FALSE

At the beginning of `amrex/Tutorials/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 must also set the DIM variable to either 1, 2, or 3, depending on the dimensionality of the problem.

Variables DEBUG, USE_MPI and USE_OMP are optional with default set to TRUE, FALSE and FALSE, respectively. 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.

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 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 clean This removes the executable, .o files, and the temporarily generated files. Note that one can add additional targets to this rule using the double colon (::)

make realclean This removes all files generated by make.

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 CC and CFLAGS are reset in that directory and their values in environment variables are disregarded. 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.

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
  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 is it 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`.

3.2 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. 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.

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
```

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 Cmake and macOS

You can also specify your own compiler in cmake using the `-DCMAKE_C_COMPILER` and `-DCMAKE_CXX_COMPILER` options. 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. Normaly clang plays well with gfortran, but if there are some issues, we recommend telling cmake to use gcc for c/c++ also.

3.3.2 Customization options

AMReX configuration settings may be specified on the command line with the `-D` option. For example, one can enable OpenMP support as follows:

```
cmake -DENABLE_OMP=1 -DCMAKE_INSTALL_PREFIX=/path/to/installdir  /path/to/amrex
```

The list of available option is reported in the table on *AMReX build options* below.

Table 3.2: AMReX build options

Option Name	Description	Default	Possible values
DIM	Dimension of AMReX build	3	2, 3
ENABLE_DP	Build with double-precision reals	ON	ON, OFF
ENABLE_PIC	Build Position Independent Code	OFF	ON, OFF
ENABLE_MPI	Build with MPI support	ON	ON OFF
ENABLE_OMP	Build with OpenMP support	OFF	ON, OFF
ENABLE_FORTRAN_INTERFACES	Build Fortran API	ON	ON, OFF
ENABLE_LINEAR_SOLVERS	Build AMReX linear solvers	ON	ON, OFF
ENABLE_AMRDATA	Build data services	OFF	ON, OFF
ENABLE_PARTICLES	Build particle classes	OFF	ON OFF
ENABLE_DP_PARTICLES	Use double-precision reals in particle classes	ON	ON, OFF
ENABLE_BASE_PROFILE	Build with basic profiling support	OFF	ON, OFF
ENABLE_TINY_PROFILE	Build with tiny profiling support	OFF	ON, OFF
ENABLE_TRACE_PROFILE	Build with trace-profiling support	OFF	ON, OFF
ENABLE_COMM_PROFILE	Build with comm-profiling support	OFF	ON, OFF
ENABLE_MEM_PROFILE	Build with memory-profiling support	OFF	ON, OFF
ENABLE_PROFPARSER	Build with profile parser support	OFF	ON, OFF
ENABLE_BACKTRACE	Build with backtrace support	OFF	ON, OFF
ENABLE_FPE	Build with Floating Point Exceptions checks	OFF	ON,OFF
ENABLE_ASSERTIONS	Build with assertions turned on	OFF	ON,OFF
CMAKE_Fortran_FLAGS	User-defined Fortran flags		user-defined
CMAKE_CXX_FLAGS	User-defined C++ flags		user-defined
ENABLE_3D_NODAL_MGML	Enable 3D nodal projection	OFF	ON,OFF
ALGOIM_INSTALL_DIR	Path to Algoim installation directory		user-defined
BLITZ_INSTALL_DIR	Path to Blitz installation directory		user-defined

The option `CMAKE_BUILD_TYPE=Debug` implies `ENABLE_ASSERTION=ON`. In order to turn off assertions in debug mode, `ENABLE_ASSERTION=OFF` must be set explicitly while invoking CMake.

The options `CMAKE_Fortran_FLAGS` and `CMAKE_CXX_FLAGS` allow the user to set his 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 `FFLAGS`/ `CXXFLAGS` are not defined in the environment, `CMAKE_Fortran_FLAGS`/ `CMAKE_CXX_FLAGS` will be set to the AMReX default values defined in `/path/to/amrex/Tools/CMake/AMReX_Compilers.cmake`.

The option `ENABLE_3D_NODAL_MGML` enables AMReX 3D nodal projection. This option requires two external libraries: Blitz and Algoim. The user can provide the location of both libraries via `BLITZ_INSTALL_DIR` and `ALGOIM_INSTALL_DIR`. However, if one or both of these options is not provided, AMReX will download and build Blitz and/or Algoim automatically. It should be noted that AMReX 2D nodal projection does not require the use of external libraries.

3.3.3 Importing AMReX into your CMake project

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

```
find_package (AMReX 18 [REQUIRED] [HINTS /path/to/installdir/] )
```

In the above snippet, `18` refer to the minimum AMReX version supporting the import feature discussed here. Linking AMReX to any target defined in your CMake project is done by including the following line in the appropriate CMakeLists.txt file

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

The above snippet will take care of properly linking `<your-target-name>` to AMReX and to all the required transitive dependencies.

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/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 and Array

`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`.

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
```

4.4 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);
```

4.5 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.

`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();
```

(continues on next page)

(continued from previous page)

```
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();
```

4.6 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";
```

4.7 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 character =) is taken to be the inputs file, and the contents in the file are used to initialize the `ParmParse` database. The rest of the command-line arguments are also parsed by `ParmParse`. 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. If 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, one can run with

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

to change the value of `ncells` and `hydro.cfl`.

4.8 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);

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);
```

`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()`.

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.

4.9 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.

4.10 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.

4.10.1 IntVect

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

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

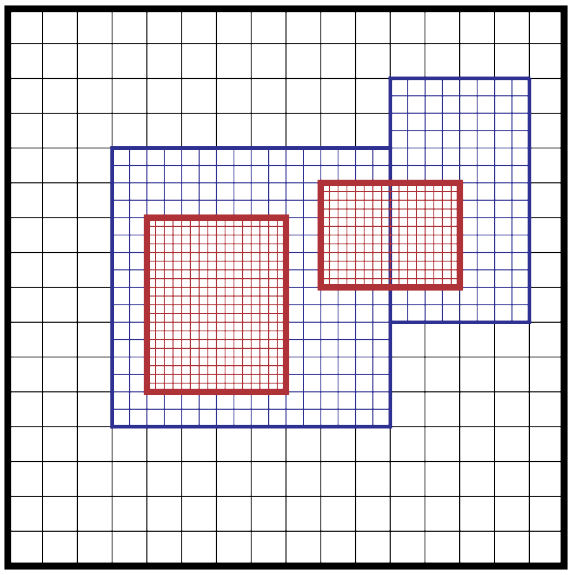


Fig. 4.1: Example of AMR grids. There are three levels in total. There are 1, 2 and 2 Boxes on levels 0, 1, and 2, respectively.

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";
```

4.10.2 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 $i + 1/2$.

4.10.3 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.

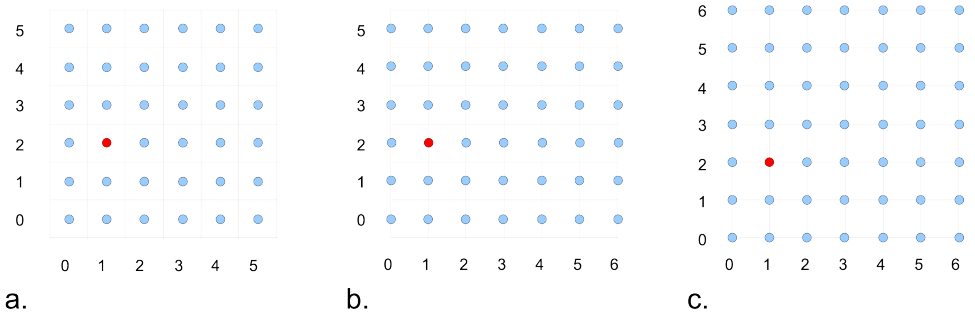


Fig. 4.2: Some of the different index types in two dimensions: (a) cell-centered, (b) x-face-centered (i.e., nodal in x-direction only), and (c) corner/nodal, i.e., nodal in all dimensions.

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";
}

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});
```

(continues on next page)

(continued from previous page)

```
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!
```

4.11 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);
```

Here the constructor takes a cell-centered Box specifying the indexing space domain, an optional argument of RealBox pointer specifying the physical domain, an optional int specifying coordinate system type, and an optional int * specifying periodicity. If a RealBox is not given, AMReX will construct one based on ParmParse parameters, geometry.prob_lo and geometry.prob_hi, where each of the parameter is an array of AMREX_SPACEDIM real numbers. It's a runtime error if this fails. The optional 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, 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 optional argument can be passed specifying periodicity in each dimension. If it is not given, 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.data());
```

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

```
const Real* problo = geom.ProbLo();     // Lower corner of the physical domain
Real yhi = geom.ProbHi(1);              // y-direction upper corner
const Real* dx = geom.CellSize();       // Cell size for each direction
const Box& domain = geom.Domain();      // Index domain
bool is_per = Geometry::isPeriodic(0); // Is periodic in x-direction?
if (Geometry::isAllPeriodic()) {}       // Periodic in all direction?
if (Geometry::isAnyPeriodic()) {}       // Periodic in any direction?
```

4.12 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 and IArrayBox*).

`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.

4.13 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.
```

4.14 BaseFab, FArrayBox and IArrayBox

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, C 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
```

For more complicated expressions that are not supported, one can write Fortran or C functions for those (see the section on *Fortran, C and C++ Kernels*). Note that BaseFab does provide operators for accessing the data directly in C++. For example, the `saxpy` example above can be done with

```
// Iterate over all components
for (int icomp=0; icomp < fab1.nComp(); ++icomp) {
    // Iterate over all cells in Box
    for (BoxIterator bit(fab1.box()); bit.ok(); ++bit) {
        // bit() returns IntVect
        fab2(bit(),icomp) = a * fab1(bit(),icomp) + fab2(bit(),icomp);
    }
}
```

But this approach is generally not recommended for performance reasons. However, it can be handy for debugging.

`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.

4.15 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 that 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 and IArrayBox*) 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{(8,8,8) (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 `BoxArrays` 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 `MultiFab`s 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).

4.16 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`.

4.16.1 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];

    // Pointer to the floating point data of this FArrayBox.
    Real* a = fab.dataPtr();

    // This is the Box on which the FArrayBox is defined.
    // Note that "abox" includes ghost cells (if there are any),
    // and is thus larger than or equal to "box".
    const Box& abox = fab.box();

    // We can now pass the information to a function that does
    // work on the region (specified by box) of the data pointed to
    // by Real* a.  The data should be viewed as multidimensional
    // with bounds specified by abox.
    // Function f1 has the signature of
    // void f1(const int*, const int*, Real*, const int*, const int*);
    f1(box.loVect(), box.hiVect(), a, abox.loVect(), abox.hiVect());
}
```

Here function f1 is usually a Fortran subroutine with ISO C binding interface like below,

```
subroutine f1(lo, hi, a, alo, ahi) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: lo(3), hi(3), alo(3), ahi(3)
  real(amrex_real),intent(inout)::a(alo(1):ahi(1),alo(2):ahi(2),alo(3):ahi(3))
  integer :: i,j,k
  do      k = lo(3), hi(3)
    do    j = lo(2), hi(2)
      do i = lo(1), hi(1)
        a(i,j,k) = ...
      end do
    end do
  end do
end subroutine f1
```

Here amrex_fort_module is a Fortran module in AMReX and amrex_real is a Fortran kind parameter that matches amrex::Real in C++. In this example, we assumed the spatial dimension is 3. In 2D, the function interface is different. In the section on *Fortran, C and C++ Kernels*, we will present a dimension-agnostic approach using macros provided by AMReX.

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() to get the number of components and pass nc to the kernel function.

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
int ncU = U.nComp();   // number of components
int ncF = F.nComp();
for (MFIter mfi(F); mfi.isValid(); ++mfi) // Loop over grids
{
    const Box& box = mfi.validbox();

    const FArrayBox& ufab = U[mfi];
    FArrayBox&       ffab = F[mfi];

    Real* up = ufab.dataPtr();
    Real* fp = ufab.dataPtr();

    const Box& ubox = ufab.box();
    const Box& fbox = ffab.box();

    // Function f2 has the signature of
    // void f2(const int*, const int*,
    //         const Real*, const int*, const int*, const int*
    //               Real*, const int*, const int*, const int*);
    // This will compute f using u as inputs.
    f2(box.loVect(), box.hiVect(),
       up, ubox.loVect(), ubox.hiVect(), &ncU,
       fp, fbox.loVect(), fbox.hiVect(), &ncF);
}
```

Here again function `f2` is usually a Fortran subroutine with ISO C binding interface like below,

```
subroutine f2(lo, hi, u, ulo, uhi, nu, f, flo, fhi, nf) bind(c)
  use amrex_fort_module, only : amrex_real
  integer, intent(in) :: lo(3),hi(3),ulo(3),uhi(3),nu,flo(3),fhi(3),nf
  real(amrex_real),intent(in   )::u(ulo(1):uhi(1),ulo(2):uhi(2),ulo(3):uhi(3),nu)
  real(amrex_real),intent(inout)::f(flo(1):fhi(1),flo(2):fhi(2),flo(3):fhi(3),nf)
  integer :: i,j,k
  do n = 1, nf
    do     k = lo(3), hi(3)
      do   j = lo(2), hi(2)
        do i = lo(1), hi(1)
          f(i,j,k,n) = ... u(...) ...
        end do
      end do
    end do
  end do
end subroutine f2
```

4.16.2 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

```
do k = kmin, kmax
  do j = jmin, jmax
    do i = imin, imax
```

(continues on next page)

(continued from previous page)

```
      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];
    Real* a = fab.dataPtr();
    const Box& abox = fab.box();

    f1(box.loVect(), box.hiVect(), a, abox.loVect(), abox.hiVect());
}
```

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();

    const FArrayBox& ufab = U[mfi];
    FArrayBox&       ffab = F[mfi];
```

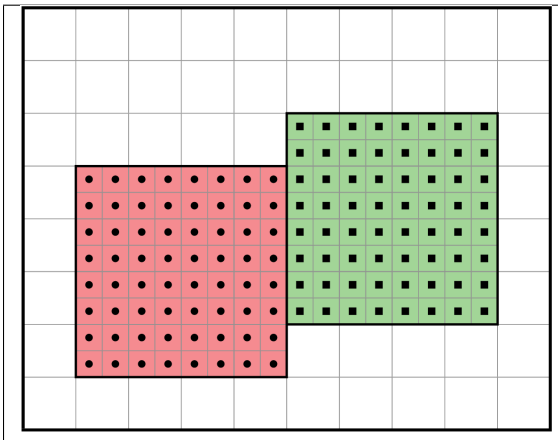
(continues on next page)

(continued from previous page)

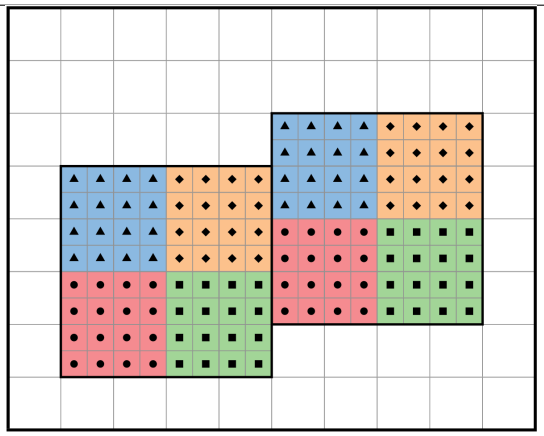
```
    Real* up = ufab.dataPtr();
    Real* fp = ufab.dataPtr();

    const Box& ubox = ufab.box();
    const Box& fbox = ffab.box();

    f2(box.loVect(), box.hiVect(),
       up, ubox.loVect(), ubox.hiVect(), &ncU,
       fp, fbox.loVect(), fbox.hiVect(), &ncF);
}
```

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

Table 4.1: 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 4.1 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:

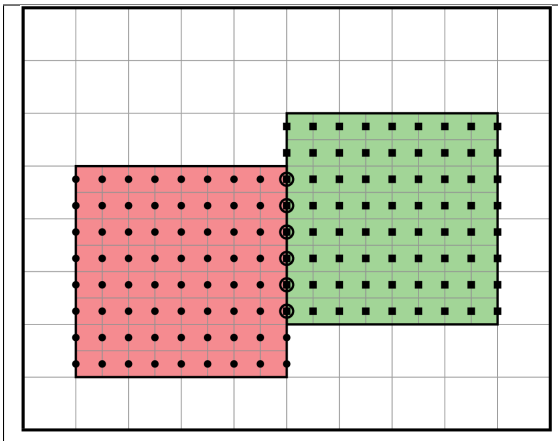
```
// MFIter loop with tiling on.
for (MFIter mfi(mf,true); mfi.isValid(); ++mfi)
{
    Box bx = mfi.validbox();      // Gets box of entire, untiled region.
    calcOverBox(bx);              // ERROR! Works on entire box, not tiled box.
                                  // Other iterations will redo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

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

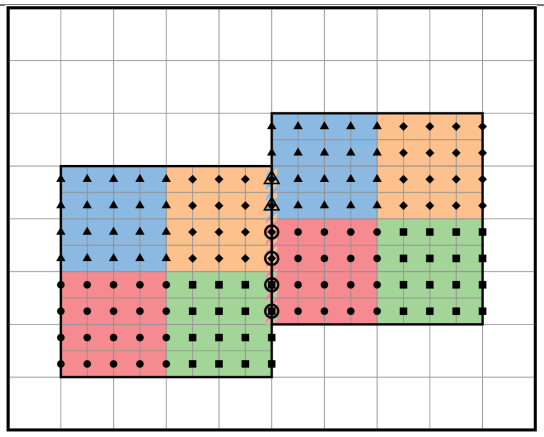
```
// No tiling in x-direction. Tile size is 16 for y and 32 for z.
for (MFIter mfi(mf,IntVect(1024000,16,32)); mfi.isValid(); ++mfi) {...}
```

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 4.2: 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 seperate 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 _OPENMP
#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 _OPENMP
#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 4.2 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 4.3: Comparing growing cell-type and face-type tile boxes.

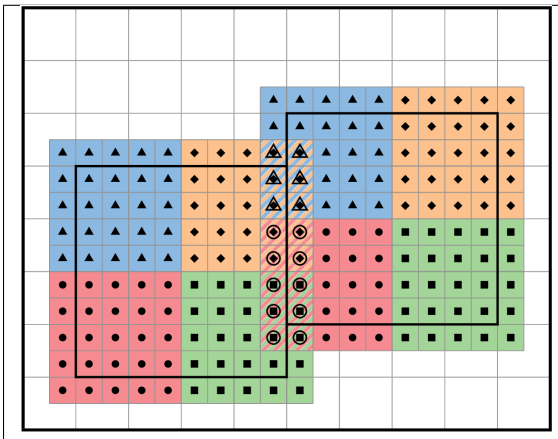
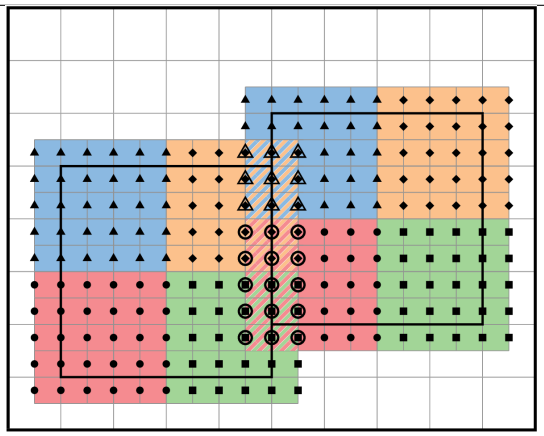
	
Example of cell-centered grown tile boxes. As indicated by symbols, there are 8 tiles and four in each grid in this example. Tiles from the same grid do not overlap. But tiles from different grids may overlap.	Example of face type grown tile boxes. As indicated by symbols, there are 8 tiles and four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

Table 4.3 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.

4.17 Fortran, C 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. Fortran is 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
```

(continues on next page)

(continued from previous page)

```
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
```

One can write the functions in C and should include the header containing the function declarations in the C source code to ensure type safety. However, we typically write these kernel functions in Fortran because of the native multi-dimensional array support by Fortran. As we have seen in the section on *MFIter and Tiling*, 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. Similar to what we have seen in the section on *MFIter and Tiling*, a matching 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 MultiFabs 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 as we have seen in the section on *MFIter and Tiling*. 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()));
}

! Fortra 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.

Writing kernels in C++ is also an option. AMReX provides a multi-dimensional array type of syntax, similar to Fortran, that is readable and easy to implement. An example is given below:

```
void f (Box const& bx, FArrayBox& fab1, FArrayBox const& fab2)
{
    const Dim3 lo = amrex::lbound(bx);
    const Dim3 hi = amrex::hbound(bx);

    Array4<Real> const& src = fab1.array();
    Array4<Real> const& dst = fab2.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::hbound` 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 Plain Old Data (POD) 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` vectors 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` and can be used in lower dimensions by passing *0* to

the higher order 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 will lead to a variety of errors, so if unsure about the independence of the iterations of a loop, test and verify before adding the macro.

4.18 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.

4.19 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. If there is more than one variable, we can create an array of BCRec objects, and pass in a pointer to the 0-index component since the arrays for all the components are contiguous in memory. Here we need to provide boundary types to each

component of the MultiFab. Below is an example of setting up Vector<BCRec> 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 (Geometry::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, $q(-i) = -q(i)$.

- We have interfaces to a fortran routine that fills ghost cells at domain boundaries based on the boundary condition type defined in the BCRec object. It is the user's responsibility to have a consisent 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. There are essentially three ways to fill the ghost cells. (refer to amrex/Tutorials/Basic/HeatEquation_EX2_C for an example).

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

// ...

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

// fills interior (but not periodic domain boundary) ghost cells
```

(continues on next page)

(continued from previous page)

```
mf.FillBoundary();

// fills physical domain boundary ghost cells for a cell-centered multifab
// except for external Dirichlet
FillDomainBoundary(mf, geom, bc);
```

`FillDomainBoundary()` is a function in `amrex/Src/Base/AMReX_BCUtil.cpp` that fills the physical domain boundary ghost cells with Fortran function `amrex_fab_filcc` except for external Dirichlet (i.e., `BCType:ext_dir`). The user can use it as a template and insert their own function for `BCType:ext_dir` like below

```
if (! grown_domain_box.contains(fab_box))
{
    amrex_fab_filcc(BL_TO_FORTRAN_FAB(fab),
                    BL_TO_FORTRAN_BOX(domain_box),
                    dx, prob_lo,
                    bc[0].data());
    user_fab_filcc(BL_TO_FORTRAN_FAB(fab),
                   BL_TO_FORTRAN_BOX(domain_box),
                   dx, prob_lo,
                   bc[0].data());
}
```

4.20 Memory Allocation

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.

4.21 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.

4.22 Debugging

Debugging is an art. Everyone has their own favorite method. Here we offer a few tips we have found to be useful.

Compiling in debug mode (e.g., make DEBUG=TRUE) and running with ParmParse parameter amrex.fpe_trap_invalid=1 can be helpful. In debug mode, many compiler debugging flags are turned on and all MultiFab data are initialized to signaling NaNs. The amrex.fpe_trap_invalid parameter will result in backtrace files when floating point exception occurs. One can then examine those files to track down the origin of the issue.

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);
```

can output the data for a single cell.

Valgrind is one of our favorite debugging tool. 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 ...
```

4.23 Example: HeatEquation_EX1_C

We now present an example of solving the heat equation. The source code tree for the heat equation example is simple, as shown in Fig. 4.3. We recommend you study main.cpp and advance.cpp to see some of the classes described

below in action.

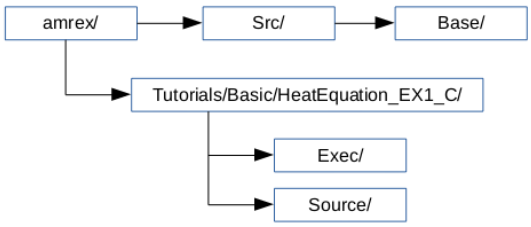


Fig. 4.3: Diagram of the source code structure for the HeatEquation_EX1_C example.

Source code tree for the HeatEquation_EX1_C example

amrex/Src/Base Contains source code for single-level simulations. Note that in `amrex/Src` there are many sub-directories, e.g., `Base`, `Amr`, `AmrCore`, `LinearSolvers`, etc. In this tutorial the only source code directory we need is `Base`.

amrex/Tutorials/HeatEquation_EX1_C/Source Contains the following source code specific to this tutorial:

1. `Make.package`: lists the source code files
2. `main.cpp`: contains the C++ `main` function
3. `advance.cpp`: advance the solution by a time step
4. `init_phi_Xd.f90, advance_Xd.f90`: fortran work functions used to initialize and advance the solution
5. `myfunc.H`: header file for C++ functions
6. `myfunc_F.H`: header file for fortran90 functions that are called in .cpp files

amrex/Tutorials/HeatEquation_EX1_C/Exec This is where you build the code with make. There is a GNUmakefile and inputs files, inputs_2d and inputs_3d.

Now we highlight a few key sections of the code. In `main.cpp` we demonstrate how to read in parameters from the inputs file:

```
// inputs parameters
{
    // ParmParse is way of reading inputs from the inputs file
    ParmParse pp;

    // We need to get n_cell from the inputs file - this is the number of cells on␣
↪each side of
    //   a square (or cubic) domain.
    pp.get("n_cell",n_cell);

    // The domain is broken into boxes of size max_grid_size
    pp.get("max_grid_size",max_grid_size);

    // Default plot_int to -1, allow us to set it to something else in the inputs file
    //  If plot_int < 0 then no plot files will be writtenq
```

(continues on next page)

(continued from previous page)

```
    plot_int = -1;
    pp.query("plot_int",plot_int);

    // Default nsteps to 10, allow us to set it to something else in the inputs file
    nsteps = 10;
    pp.query("nsteps",nsteps);

    pp.queryarr("is_periodic", is_periodic);
}
```

In `main.cpp` we demonstrate how to define a `Box` for the problem domain, and then how to chop that `Box` up into multiple boxes that define a `BoxArray` We also define a `Geometry` object that knows about the problem domain, the physical coordinates of the box, and the periodicity:

```
// make BoxArray and Geometry
BoxArray ba;
Geometry geom;
{
    IntVect dom_lo(AMREX_D_DECL(       0,        0,        0));
    IntVect dom_hi(AMREX_D_DECL(n_cell-1, n_cell-1, n_cell-1));
    Box domain(dom_lo, dom_hi);

    // Initialize the boxarray "ba" from the single box "bx"
    ba.define(domain);
    // Break up boxarray "ba" into chunks no larger than "max_grid_size" along a␣
↪direction
    ba.maxSize(max_grid_size);

   // 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 defines a Geometry object
    geom.define(domain,&real_box,CoordSys::cartesian,is_periodic.data());
}
```

In `main.cpp` we demonstrate how to build a `DistributionMapping` from the `BoxArray`, and then build `MultiFabs` with a desired number of components and ghost cells associated with each grid:

```
// Nghost = number of ghost cells for each array
int Nghost = 1;

// Ncomp = number of components for each array
int Ncomp  = 1;

// How Boxes are distrubuted among MPI processes
DistributionMapping dm(ba);

// we allocate two phi multifabs; one will store the old state, the other the new.
MultiFab phi_old(ba, dm, Ncomp, Nghost);
MultiFab phi_new(ba, dm, Ncomp, Nghost);
```

We demonstrate how to build an array of face-based `MultiFabs` :

```
// build the flux multifabs
std::array<MultiFab, AMREX_SPACEDIM> flux;
```

(continues on next page)

(continued from previous page)

```
for (int dir = 0; dir < AMREX_SPACEDIM; dir++)
{
    // flux(dir) has one component, zero ghost cells, and is nodal in direction dir
    BoxArray edge_ba = ba;
    edge_ba.surroundingNodes(dir);
    flux[dir].define(edge_ba, dm, 1, 0);
}
```

To access and/or modify data n a `MultiFab` we use the `MFIter`, where each processor loops over grids it owns to access and/or modify data on that grid:

```
// Initialize phi_new by calling a Fortran routine.
// MFIter = MultiFab Iterator
for ( MFIter mfi(phi_new); mfi.isValid(); ++mfi )
{
    const Box& bx = mfi.validbox();

    init_phi(BL_TO_FORTRAN_BOX(bx),
             BL_TO_FORTRAN_ANYD(phi_new[mfi]),
             geom.CellSize(), geom.ProbLo(), geom.ProbHi());
}
```

Note that these calls to fortran subroutines require a header in `myfunc_F.H`:

```
void init_phi(const int* lo, const int* hi,
              amrex_real* data, const int* dlo, const int* dhi,
              const amrex_real* dx, const amrex_real* prob_lo, const amrex_real* prob_
↪hi);
```

The associated fortran routines must shape the data accordinly:

```
subroutine init_phi(lo, hi, phi, philo, phihi, dx, prob_lo, prob_hi) bind(C, name=
↪"init_phi")

  use amrex_fort_module, only : amrex_real

  implicit none

  integer, intent(in) :: lo(2), hi(2), philo(2), phihi(2)
  real(amrex_real), intent(inout) :: phi(philo(1):phihi(1),philo(2):phihi(2))
  real(amrex_real), intent(in   ) :: dx(2)
  real(amrex_real), intent(in   ) :: prob_lo(2)
  real(amrex_real), intent(in   ) :: prob_hi(2)

  integer          :: i,j
  double precision :: x,y,r2

  do j = lo(2), hi(2)
     y = prob_lo(2) + (dble(j)+0.5d0) * dx(2)
     do i = lo(1), hi(1)
        x = prob_lo(1) + (dble(i)+0.5d0) * dx(1)

        r2 = ((x-0.25d0)**2 + (y-0.25d0)**2) / 0.01d0

        phi(i,j) = 1.d0 + exp(-r2)
```

(continues on next page)

(continued from previous page)

```
     end do
  end do

end subroutine init_phi
```

Ghost cells are filled using the `FillBoundary` function:

```
// Fill the ghost cells of each grid from the other grids
// includes periodic domain boundaries
phi_old.FillBoundary(geom.periodicity());
```

CHAPTER

FIVE

AMRCORE SOURCE CODE

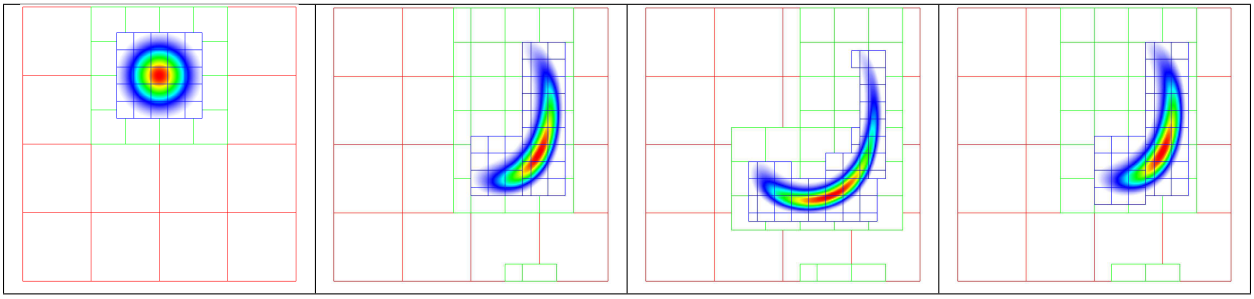
In this Chapter we give an overview of functionality contained in the `amrex/Src/AmrCore` source code. This directory contains source code for the following:

- 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/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 5.1: Time sequence ($t = 0, 0.5, 1, 1.5, 2$ s) of advection of a Gaussian profile using the SingleVortex tutorial. The red, green, and blue boxes indicate grids at AMR levels $\ell = 0, 1$, and 2.



5.1 AmrCore Source Code: Details

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

5.1.1 AmrMesh and AmrCore

For single-level simulations (see e.g., `amrex/Tutorials/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 5.2: 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/Amr/AmrCore_Advection/Source` code for a sample implementation.

5.1.2 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).

5.1.3 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
- CellQuadratic
- PCInterp
- CellConservativeQuartic

The Fortran routines that perform the actual work associated with Interpolater are contained in the files AMReX_INTERP_F.H and AMReX_INTERP_xD.F.

5.1.4 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.

5.1.5 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).

5.2 Example: Advection_AmrCore

5.2.1 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,

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

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 $\ell + 1$.

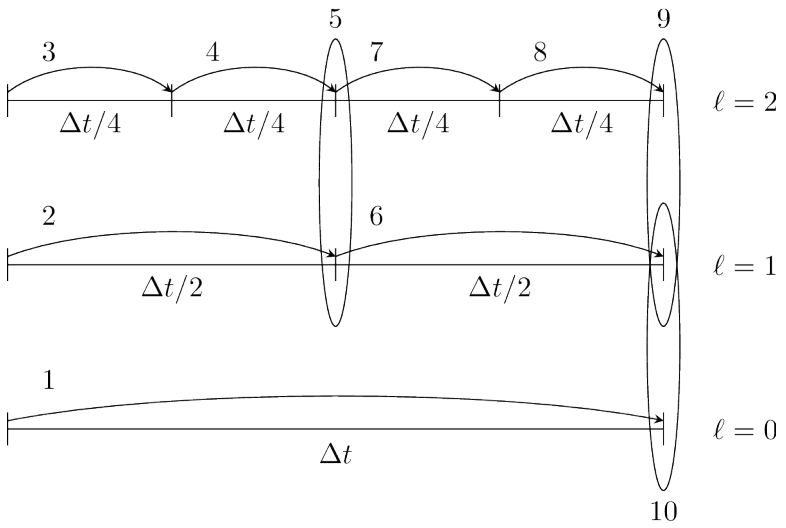


Fig. 5.1: Schematic of subcycling-in-time algorithm.

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.

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).

5.2.2 Code Structure

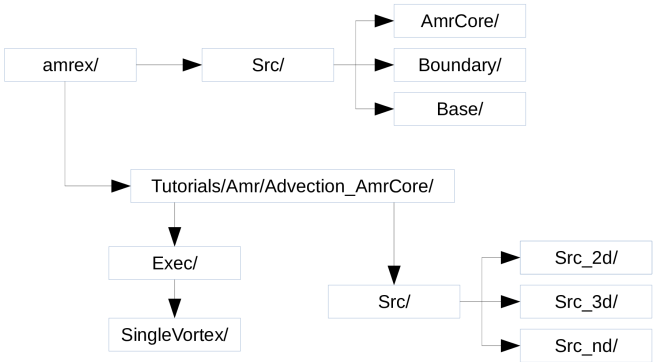


Fig. 5.2: Source code tree for the AmrAdvection_AmrCore example.

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 */
              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
        }
    }
```

5.2.3 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.

5.2.4 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]) {
      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
```

5.2.5 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)
            {
```

(continues on next page)

(continued from previous page)

```
                // 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 function 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 _OPENMP
#pragma omp parallel
#endif
```

(continues on next page)

(continued from previous page)

```
    {
        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
        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
```

(continues on next page)

(continued from previous page)

```
  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
           endif
        enddo
     enddo
  enddo

end subroutine state_error
```

5.2.6 Grid Creation

The gridding algorithm proceeds in this order, using the parameters described in the section on the *AmrCore Source Code: Details*.

1. If at level 0, the domain is initially defined by `n_cell` as specified in the inputs file. If at level greater than 0, grids are created using the Berger-Rigoutsis clustering algorithm applied to the tagged cells from the section on *Regridding*, modified to ensure that all new fine grids are divisible by `blocking_factor`.
2. Next, the grid list is chopped up if any grids are larger than `max_grid_size`. Note that because `max_grid_size` is a multiple of `blocking_factor` (as long as `max_grid_size` is greater than `blocking_factor`), the blocking_factor criterion is still satisfied.
3. Next, if `refine_grid_layout = 1` and there are more processors than grids at this level, then the grids at this level are further divided in order to ensure that no processors has less than one grid (at each level). In `AmrMesh::ChopGrids`,
   - if `max_grid_size / 2` in the `BL_SPACEDIM` direction is a multiple of `blocking_factor`, then chop the grids in the `BL_SPACEDIM` direction so that none of the grids are longer in that direction than `max_grid_size / 2`
   - If there are still fewer grids than processes, repeat the procedure in the `BL_SPACEDIM-1` direction, and again in the `BL_SPACEDIM-2` direction if necessary
   - If after completing a sweep in all coordinate directions with `max_grid_size / 2`, there are still fewer grids than processes, repeat the steps above with `max_grid_size / 4`.

5.2.7 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

SIX

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 our compressible astrophysics code, CASTRO, (available in the AMReX-Astro/Castro github repository) and 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 (not yet public) also inherits from `NavierStokesBase`.

The tutorial code in `amrex/Tutorials/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 `amrex/Tutorials/Amr/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.).

6.1 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.

6.2 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.

6.2.1 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.

6.3 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 ()
{
    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,
```

(continues on next page)

(continued from previous page)

```
                &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);

    desc_lst.setComponent(Phi_Type, 0, "phi", bc,
              StateDescriptor::BndryFunc(nullfill));
}
```

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 a fortran routine in the tutorial source code.

6.4 Example: Advection_AmrLevel

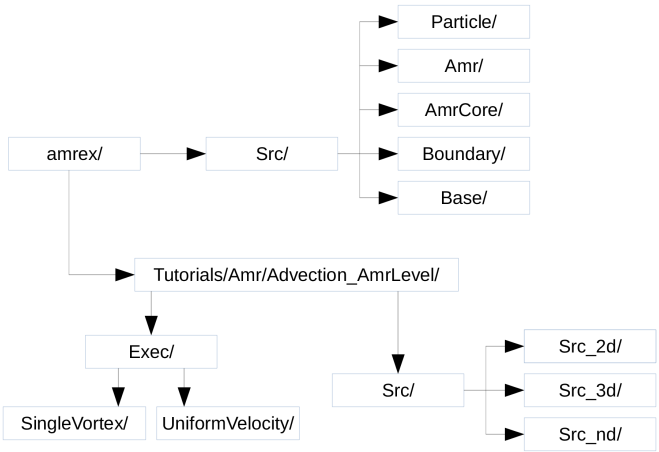


Fig. 6.1: Source code tree for the AmrAdvection_AmrLevel example.

The figure above shows the *Source code tree for the AmrAdvection_AmrLevel 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.
  - Amr/

- `Advection_AmrLevel/Src` Source code specific to this example. Most notably is the AmrLevelAdv class, which is derived from AmrLevel. 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 and UniformVelocity.
- SingleVortex and UniformVelocity 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.

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

6.5 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

SEVEN

ASYNCHRONOUS ITERATORS (AMRTASK)

Hiding communication overheads via overlapping communication with computation requires a sufficiently large amount of task parallelism. This problem is even more challenging due to various types of tasks in an AMReX program, including data parallel tasks (same workload on different data partitions) and control parallel tasks (different types of workload). This chapter introduces the API of AMReX's asynchronous iterators that can facilitate the job of identifying tasks in the applications. We have developed two iterators called FillPatch and RegionGraph Iterators, which will be described later on in this chapter. We first show how the programmer can use a runtime system to execute application codes written with these iterators.

In `amrex/Src/AmrTask/rts_impls`, we implement RTS - a runtime system that can execute asynchronous AMReX applications efficiently on large-scale systems. RTS is a black box to the application developer as showed in the following code snippet, which is the main function of a typical AMReX application running asynchronously under the control of the runtime system. The programmer first needs to use the namespace `perilla`, which covers all the C++ classes for the runtime system. To execute an AMR program (i.e. object of the Amr class), the programmer can simply create an object of RTS and pass the program object into the `Iterate` method. The runtime system will iteratively execute coarse time steps until the program completes. By default RTS links to MPI and Pthreads libraries. The programmer can also switch to other backends such as UPCXX (1-sided communication model compared to the common 2-sided model in MPI) without changing the application source code.

```
using namespace perilla;
int main (int argc, char* argv[])
{
    amrex::Initialize(argc,argv);
    ... //set up program input, e.g. start_time, stop_time, max_step
    Amr amr;
    amr.init(start_time,stop_time);
    RTS rts;
    rts.Iterate(&amr, max_step, stop_time);
    amrex::Finalize();
    return 0;
}
```

In a few functions of the Amr class, the runtime exposes multiple threads per process. As a result, the programmer needs to place sufficient memory protection for shared data within the process, e.g. when updating the state data. This multithreaded interface adds some programming cost, but is necessary for mitigating the task scheduling overhead. To avoid these programming details, the programmer can use built-in iterators, such as fillpatch iterator and task graph iterator that we next discuss. The API of these iterators is very simple, and the asynchronous code is very similar to the original code using the synchronous multifab iterator (MFIter) described earlier in chapter Basics.

7.1 FillPatch Iterator

FillPatch is an important operation commonly used in AMReX applications. This operation interpolates data in both space and time. Communication between AMR levels may incur when FillPatch interpolates data from a coarse AMR level and stores the result on the next finer level. This operation also results in communication within the same AMR level when the subcycling option is used, which requires data interpolation in time.

We develop an asynchronous version of the FillPatch operation, called Asynchronous FillPatch Iterator. Each iterator takes care of the communication with the previous and next subcycles at the same AMR level (time) and between the current and the next finer AMR levels (space and time). The iterator first automatically prepares temporary data needed for these communication activities and the data connections (aka data paths or data dependencies) among them.

Based on this setup, the programmer can design numerical solvers. This work is fairly simple. At a certain simulation time on an AMR level, the programmer can ask the runtime system which FABs have received sufficient data for advancing to the next time step. Although the FillPatch operation can be handled independently by the communication handler of the runtime system, this operation requires some computations such as packing/unpacking and extrapolation. The programmer has the freedom to dedicate a few threads from the pool of worker threads to parallelize those computations. This design choice may help the runtime process FillPatch operations faster, but may slow down the main computation. Thus, our advise to the programmer on using how many threads for the FillPatch is that it depends on the compute intensity of the actual workload. If the simulation is memory-bandwidth or network-bandwidth bound, the programmer can get the benefit from sparing more threads for doing FillPatch.

7.2 RegionGraph Iterator

We can simplify the programming work further with a new abstraction called RegionGraph Iterator a.k.a RGIter. This abstraction is a for loop (see the following code snippet), which can hide details of the asynchronous FillPatch Iterator in the init part and the graph traversal in the ++ operator. The only job required from the programmer is to specify the computations on the data, and they can easily place these computations in the loop body.

```
for (RGIter rgi(afpi_vec, upper_afpi_vec, ...); rgi.isValid(); ++rgi){
    int f = rgi.currentRegion;
    ...//computation on FAB f
}
```

The execution of RGIter is as follows. Initially, an object of RGIter (i.e. rgi) is instantiated, taking vectors of FillPatch Iterators on the current and upper AMR levels as arguments (each element of the vector corresponds to a specific time). Based on these arguments, a task dependency graph spanning two AMR levels will be established. Next, isValid() asks the runtime system for FABs that have received all dependent data. When there is such a FAB, the computations in the loop body can execute on the FAB's data. When the computations on a FAB finish, the ++ operator is called. We overload this operator to traverse to the next runnable FAB.

Note: RGIter also supports data tiling. Specifically, we overload the ++ operator so that it will traverse data tiles in a FAB before it goes to next FAB if the tiling flag in the FAB is enabled. Instead of applying the computations in the loop body on the entire FAB, it executes them on a single tile at a time.

7.3 Generated Task Graph Code

The real input to the runtime system is an AMR program containing task dependency graphs (or task graph for short). Thus, the code written with the above asynchronous iterators will be transformed into a task graph form. The definition of a task dependency graph is as follows. Each task of a graph performs some computations on an FArrayBox (FAB). Tasks are connected with each other via edges, denoting the dependency on data. A task can be executed when all data dependencies have been satisfied. The code snippet below queries runnable tasks of a task dependency graph

named regionGraph. Note that each task dependency graph is more or less a wrapper of a MultiFab. In this example, a task of regionGraph computes the body code of the while loop to update the associated FAB. Each task of this graph receives data arrived at the runtime system and injects the data into the associated FAB. After updating FAB, it lets the runtime know about the change. The runtime system uses AMR domain knowledge to establish data dependencies among tasks, and thus it can answer which tasks are runnable and how to update neighbor FABs when a current FAB changes.

```
while(!regionGraph->isGraphEmpty())
{
    f = regionGraph->getAnyFireableRegion();
    multifabCopyPull(..., f, ...); //inject arrived dependent data into the fab, if␣
↪any
    syncWorkerThreads();
    ...//compute on the fab f of multifab associated with coarseRegionGraph
    syncWorkerThreads();
    multifabCopyPush(..., f, ...); //tell the runtime that data of Fab f changed
    regionGraph->finalizeRegion(f)
}
```

The process of learning the domain knowledge is as follows. At the beginning of the program, the runtime extracts the metadata needed for establishing data dependencies among tasks of the same graph or between two different graphs. Every time the AMR grid hierarchy changes (i.e. when a few or all AMR levels regrid), the runtime re-extracts the metadata to correct the task dependency graphs. Once the metadata extraction completes, the runtime system invokes the computation on AMR levels (e.g., timeStep, initTimeStep, and postTimeStep).

7.4 Known Limitations

To realize enough task parallelism, the runtime system constructs a task dependency graph for the whole coarse time step and executes it asynchronously to the completion of the step. As a result, any request to regrid an AMR level must be foreseen before the execution of a coarse time step. If there is a regridding request during the graph execution, the runtime system simply ignores it. In the future we may relax this constraint in the programming model. However, such a support would come at a significant performance cost due to the required checkpointing and rollback activities.

CHAPTER

EIGHT

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.

8.1 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); // plt12344567
```

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.

8.2 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 `/amrex/Tutorials/Amr/Advection_AmrCore/Exec/SingleVortex`. 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";
```

(continues on next page)

(continued from previous page)

```
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) {
```

(continues on next page)

(continued from previous page)

```
        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);
    }
}
```

(continues on next page)

(continued from previous page)

```
// 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);
    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].reset(new 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

NINE

LINEAR SOLVERS

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

$$(\alpha A - \beta \nabla \cdot B \nabla)\phi = f, \tag{9.1}$$

where α and β are scalar constants, A and B 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 the canonical form. The sought solution ϕ is at either cell centers or nodes. For the cell-centered solver, A, ϕ and f are represented by cell-centered MultiFabs, and B is represented by `AMREX_SPACEDIM` face type MultiFabs. That is there are 1, 2 and 3 MultiFabs representing B for 1, 2 and 3D, respectively. For the nodal solver, A is assumed to be zero, ϕ and f are nodal, and B is cell-centered. AMReX supports both single-level solve and composite solve on multiple AMR levels.

The tutorials in `amrex/Tutorials/LinearSolvers/` show examples of using the solvers. The tutorial `amrex/Tutorials/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.

9.1 MLMG and Linear Operator Classes

`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 (9.1)).
- `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 = {});
```

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

```
// Given Geometry geom, BoxArray grids, and DistributionMapping dmap on single level
MLABecLaplacian mlabeclap({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*.

For MLABecLaplacian, we next need to call member functions

```
void setScalars (Real alpha, Real beta);
void setACoeffs (int amrlev, const MultiFab& A);
void setBCoeffs (int amrlev, const Array<MultiFab const*,AMREX_SPACEDIM>& B);
```

to set up the coefficients for equation (9.1). This is unecessary for MLPoisson, as there are no coefficients to set. For MLNodeLaplacian, one needs to call the member function

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

The int amrlev parameter should be zero for single-level solves. For multi-level solves, each level needs to be provided with A and B, 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 solver will terminate when one of these targets is met. 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 $-B\nabla\phi$.

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

9.2 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.

First we need to set physical domain boundary types via `MLLinOp` member function

```
void setDomainBC (const Array<BCType,AMREX_SPACEDIM>& lobc,  // for lower ends
                  const Array<BCType,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::reflect_odd` for reflection with sign changed.

The second step is to set up coarse/fine Dirichlet boundary conditions. This step is not always needed. But when it is, it must be completed before step 3. This step is not needed when the coarsest level in the solver covers the whole computational domain (e.g., the coarsest level is AMR level 0). Note that this step is still needed in the case that the solver is used to do a single-level solve on a fine AMR level not covering the whole domain. 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.

In the third step, for each level, we call `MLLinOp` member function

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

to set up Dirichlet boundary values. Here the `MultiFab` must have one ghost cell. Although `MultiFab` is used to pass the data, only the values in the ghost cells at Dirichlet boundaries are used. If there are no Dirichlet boundaries, we still need to make this call, but we could call it with `nullptr`. It should be emphasized that the data in `levelbcdata` for Dirichlet boundaries are assumed to be exactly on the face of the physical domain even though for cell-centered solvers the sought unknowns are at cell centers. And for cell-centered solvers, the `MultiFab` argument must have the cell-centered type.

9.3 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)` can be 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
```

(continues on next page)

(continued from previous page)

```
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 the biconjugate gradient stabilized method as the bottom solver. `MLMG` member method

```
void setBottomSolver (BottomSolver s);
```

can be used to change the bottom solver. Available choices are

- `MLMG::BottomSolver::smoother`: Smoother such as Gauss-Seidel.
- `MLMG::BottomSolver::bicgstab`: The default.
- `MLMG::BottomSolver::cg`: The conjugate gradient method. The matrix must be symmetric.
- `MLMG::BottomSolver::Hypre`: BoomerAMG in HYPRE. Currently for cell-centered only.

9.4 Curvilinear Coordinates

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 call `setMetricTerm(bool)` with `false` on the `LPInfo` object passed to the constructor of linear operators.

9.5 HYPRE

AMReX can use HYPRE BoomerAMG as a bottom solver (currently for cell-centered problems only), as we have mentioned. For challenging problems, our geometric multigrid solver may have difficulty solving, whereas an algebraic multigrid method might be more robust. We note that by default our solver 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 use HYPRE, one must include `amrex/Src/Extern/HYPRE` in the build system. For an example of using HYPRE, we refer the reader to `Tutorials/LinearSolvers/ABecLaplacian_C`.

9.6 Embedded Boundaries

AMReX support solving linear systems with embedded boundaries. See chapter *Embedded Boundaries* for more details.

CHAPTER

TEN

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.

10.1 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/Particles/ElectrostaticPIC/ElectrosticParticleContainer.H` for an example of this.

10.1.1 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;
```

10.2 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;
```

10.2.1 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

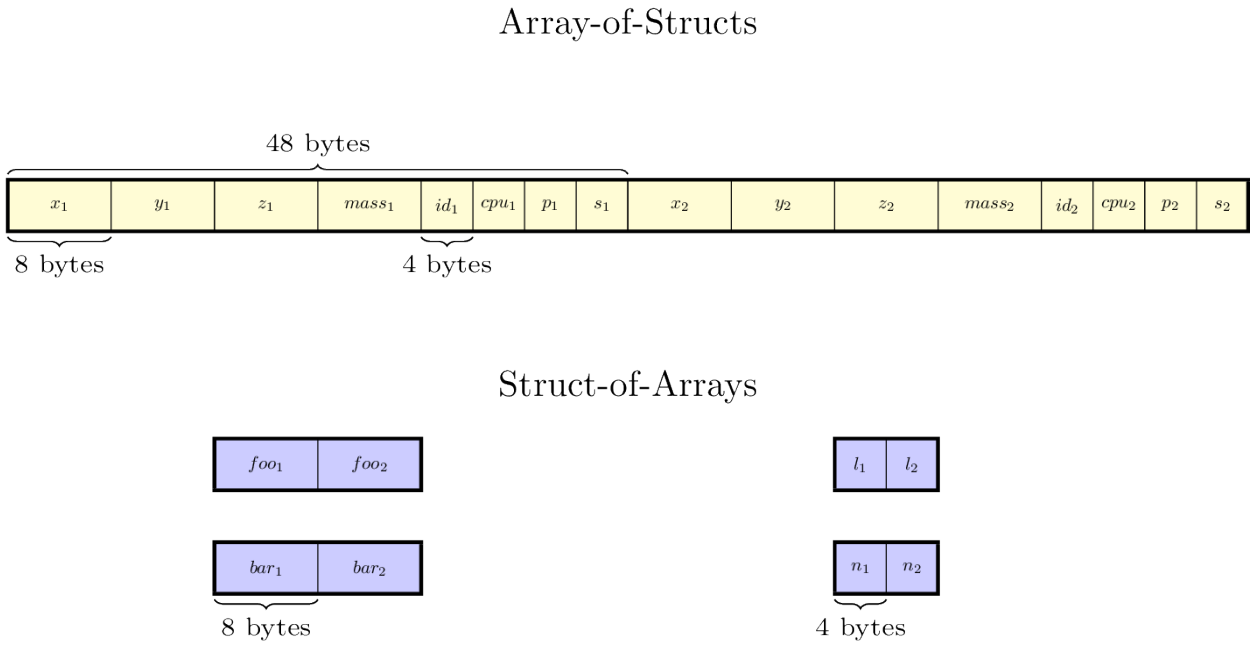


Fig. 10.1: An illustration of how the particle data for a single tile is arranged in memory. This particle container has been defined with `NStructReal = 1`, `NStructInt = 2`, `NArrayReal = 2`, and `NArrayInt = 2`. In this case, each tile in the particle container has five arrays: one with the particle 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.

10.2.2 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`.

The `ParticleContainer` stores the particle data in a manner prescribed by the set of AMR grids used to define it. If tiling is turned off, then every grid has its own Array-of-Structs and Struct-of-Arrays. Which AMR grid a particle is assigned to 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. When tiling is enabled, then each *tile* gets its own Struct-of-Arrays and Array-of-Structs instead. Note that this is different than what happens with mesh data. With mesh data, the tiling is strictly logical; the data is laid out in memory the same 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.

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 `amrex/Tutorials/Particles` for examples of this.

10.3 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]  = ...

    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).

10.4 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) {
```

(continues on next page)

(continued from previous page)

```
    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();
    Vector<Real>& real_comp0 = particle_attributes.GetRealData(0);
    Vector<int>&  int_comp1  = particle_attributes.GetIntData(1);
    for (int i = 0; i < pti.numParticles; ++i) {
        // do stuff with your SoA data...
    }
}
```

10.5 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.

10.6 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());
```

(continues on next page)

(continued from previous page)

```
for (MyParIter pti(MyPC, lev); pti.isValid(); ++pti) {
    const Box& box = Ex[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 = rho[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 `amrex/Tutorials/Particles/ElectrostaticPIC`.

10.7 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. 10.2, 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.

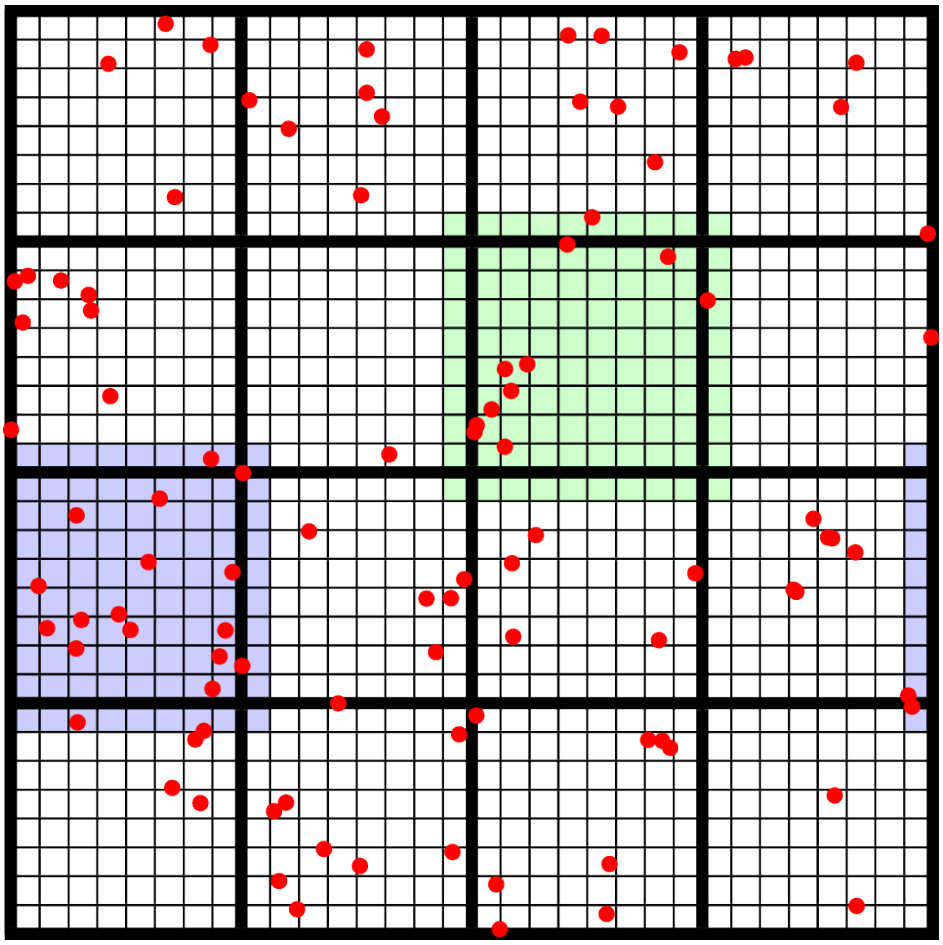


Fig. 10.2: : An illustration of filling neighbor particles for short-range force calculations. Here, we have a domain consisting of one 32×32 grid, broken up into 8×8 tiles. The number of ghost cells is taken to be 1. For the tile in green, particles on other tiles in the entire shaded region will copied 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.

For a `ParticleContainer` that does this neighbor finding, please see `NeighborParticleContainer` in `amrex/Src/Particles/AMReX_NeighborParticleContainer.H`. This `ParticleContainer` 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 `amrex/Tutorials/`

Particles/ShortRangeParticles. This tutorial computes the forces on a given tile via direct summation by passing the real and neighbor particles into a Fortran subroutine, as follows:

```
void ShortRangeParticleContainer::computeForces() {
    for (MyParIter pti(*this, lev); pti.isValid(); ++pti) {
        AoS& particles = pti.GetArrayOfStructs();
        int Np = particles.size();
        PairIndex index(pti.index(), pti.LocalTileIndex());
        int Nn = neighbors[index].size() / pdata_size;
        amrex_compute_forces(particles.data(), &Np,
                             neighbors[index].dataPtr(), &Nn);
    }
}
```

Alternatively, one can avoid doing a direct N^2 summation over the particles on a tile by binning the particles by cell and building a neighbor list. A tutorial that demonstrates this process is available at amrex/Tutorials/Particles/NeighborList. The data structure used to represent the neighbor lists is illustrated in Fig. 10.3.

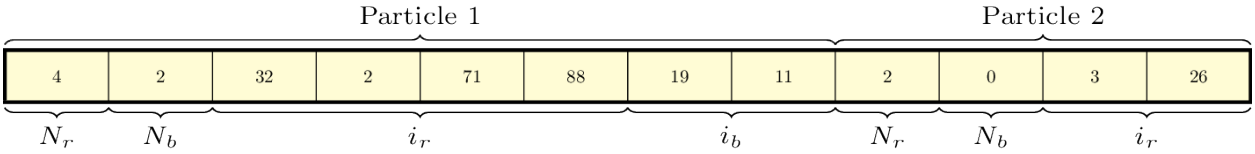


Fig. 10.3: : An illustration of the neighbor list data structure used by AMReX. The list for each tile is represented by an array of integers. The first number in the array is the number of real (i.e., not in the neighbor buffers) collision partners for the first particle on this tile, while the second 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. For an example of this in action, please see the NeighborList Tutorial.

10.8 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.

CHAPTER

ELEVEN

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, C 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*.

11.1 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 `amrex/Tutorials/Basic/HelloWorld_F` and `amrex/Tutorials/Amr/Advection_F` for examples). The libamrex approach includes the Fortran interface by default. The CMake approach does not support the Fortran interface yet.

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*), amrex_amrcore_module for AMR support (Section *3*) and amrex_octree_module for octree style AMR (Section *4*).

11.2 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
! ...
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)
```

(continues on next page)

(continued from previous page)

```
! mf2 no longer contains a valid pointer because mf1 has been destroyed.
call amrex_multifab_destroyed(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_destroyed(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.

11.3 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
  ! 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
```

Tutorials/Amr/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 Tutorials/Amr/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 Tutorials/Amr/Advection_F/.

11.4 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/Amr/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()
```

(continues on next page)

(continued from previous page)

```
  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

TWELVE

EMBEDDED BOUNDARIES

12.1 Overview of Embedded Boundary Description

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. 12.1 for an illustration.

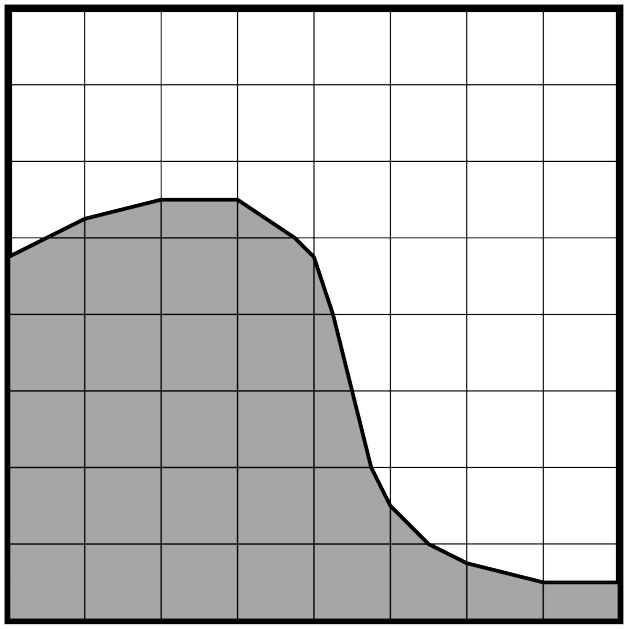


Fig. 12.1: : In the embedded boundary approach to discretizing PDEs, the (uniform) rectangular mesh is cut by the irregular shape of the computational domain. The cells in the mesh are label as regular, cut or covered.

Because this is a relatively simple grid generation technique, computational meshes for rather complex geometries can 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. In this chapter we overview a class of approaches to deal with this "small cell" problem in a robust and efficient way, and discuss the tools and data that AMReX provides in order to implement them.

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.

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.

12.1.1 Finite Volume Discretizations

Consider a system of PDEs to advance a conserved quantity U with fluxes F:

$$\frac{\partial U}{\partial t} + \nabla \cdot F = 0. \tag{12.1}$$

A conservative, finite volume discretization starts with the divergence theorm

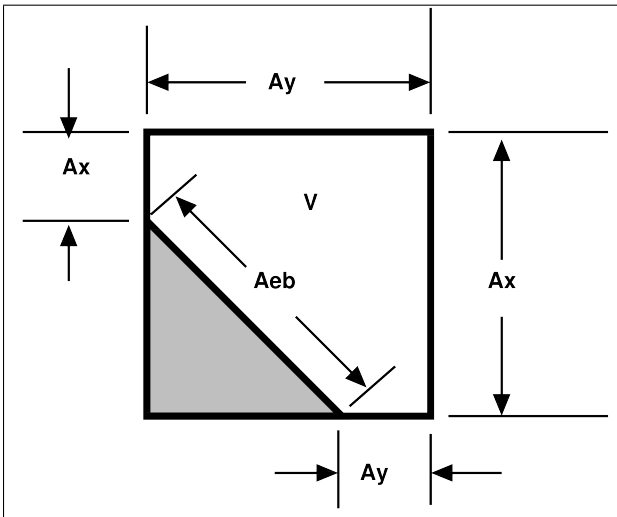
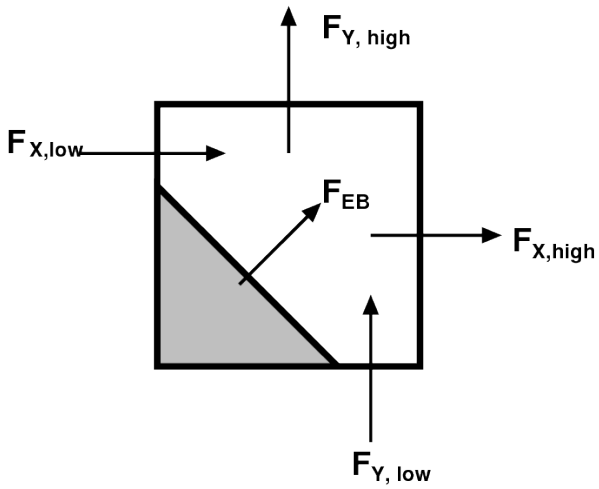
$$\int_V \nabla \cdot F dV = \int_{\partial V} F \cdot n dA.$$

In an embedded boundary cell, the "conservative divergence" is discretized (as $D^c(F)$) as follows

$$D^c(F) = \frac{1}{\kappa h}\left(\sum_{d=1}^{D}(F_{d,\mathrm{hi}}\,\alpha_{d,\mathrm{hi}} - F_{d,\mathrm{lo}}\,\alpha_{d,\mathrm{lo}}) + F^{EB}\alpha^{EB}\right). \tag{12.2}$$

Geometry is discretely represented by volumes ($V = \kappa h^d$) and apertures ($A = \alpha h^{d-1}$), where h is the (uniform) mesh spacing at that AMR level, κ is the volume fraction and α are the area fractions. Without multivalued cells the volume fractions, area fractions and cell and face centroids (see Table 12.1) are the only geometric information needed to compute second-order fluxes centered at the face centroids, and to infer the connectivity of the cells. Cells are connected if adjacent on the Cartesian mesh, and only via coordinate-aligned faces on the mesh. If an aperture, $\alpha = 0$, between two cells, they are not directly connected to each other.

Table 12.1: Illustration of embedded boundary cutting a two-dimensional cell.

	
A typical two-dimensional uniform cell that is cut by the embedded boundary. The grey area represents the region excluded from the calculation. The portion of the cell faces faces (labelled with A) through which fluxes flow are the "uncovered" regions of the full cell faces. The volume (labelled V) is the uncovered region of the interior.	Fluxes in a cut cell.

12.1.2 Small Cells And Stability

In the context of time-explicit advance methods for, say hyperbolic conservation laws, a naive discretization in time of (12.1) using (12.2),

$$U^{n+1} = U^n - \delta t D^c(F)$$

would have a time step constraint $\delta t \sim h\kappa^{1/D}/V_m$, which goes to zero as the size of the smallest volume fraction κ in the calculation. Since EB volume fractions can be arbitrarily small, this is an unacceptable constraint. One way to remedy this is to create "non-conservative" approximation to the divergence D^{nc}, which at a cell $\mathbf{i}$, can be formed as an average of the conservative divergences in the neighborhood, $N_\mathbf{i}$, of $\mathbf{i}$.

$$D^{nc}(F)_\mathbf{i} = \frac{\sum_{\mathbf{j}\in N_\mathbf{i}} \kappa_\mathbf{j} D(F)_\mathbf{j}}{\sum_{\mathbf{j}\in N_\mathbf{i}} \kappa_\mathbf{j}}$$

Incorporating this form, the solution can be updated using a *hybrid divergence*, $D^H(F) = \kappa D^c(F) + (1-\kappa)D^{nc}$:

$$U^{n+1,*} = U^n - \delta t D^H(F)$$

However, we would like our finite-volume scheme to strictly conserve the field quantities over the domain. To enforce this, we calculate δM, the mass gained or lost by not using D^c directly,

$$\delta M_\mathbf{i} = \kappa(1-\kappa)(D^c(F)_\mathbf{i} - D^{nc}(F)_\mathbf{i})$$

This "excess material" (mass, if $U = \rho$) can be *redistributed* in a time-explicit fashion to neighboring cells, $\mathbf{j} \in N_\mathbf{i}$:

$$\delta M_\mathbf{i} = \sum_{\mathbf{j} \in N_\mathbf{i}} \delta M_{\mathbf{j},\mathbf{i}}.$$

in order to preserve strict conservation over $N_\mathbf{i}$.

Note that the physics at hand may impact the optimal choice of precisely how the excess mass is distributed in this fashion. We introduce a weighting for redistribution, W,

$$\delta M_{\mathbf{j},\mathbf{i}} = \frac{\delta M_\mathbf{i} \kappa_\mathbf{j} W_\mathbf{j}}{\sum_{\mathbf{k} \in N_\mathbf{i}} \kappa_\mathbf{k} W_\mathbf{k}} \tag{12.3}$$

For all $\mathbf{j} \in N_\mathbf{i}$,

$$U_\mathbf{j}^{n+1} = U_\mathbf{j}^{n+1,*} + \frac{\delta M_\mathbf{i} W_\mathbf{j}}{\sum_{\mathbf{k} \in N_\mathbf{i}} \kappa_\mathbf{k} W_\mathbf{k}}.$$

Typically, the redistribution neighborhood for each cell is one that can be reached via a monotonic path in each coordinate direction of unit length (see, e.g., Fig. 12.2)

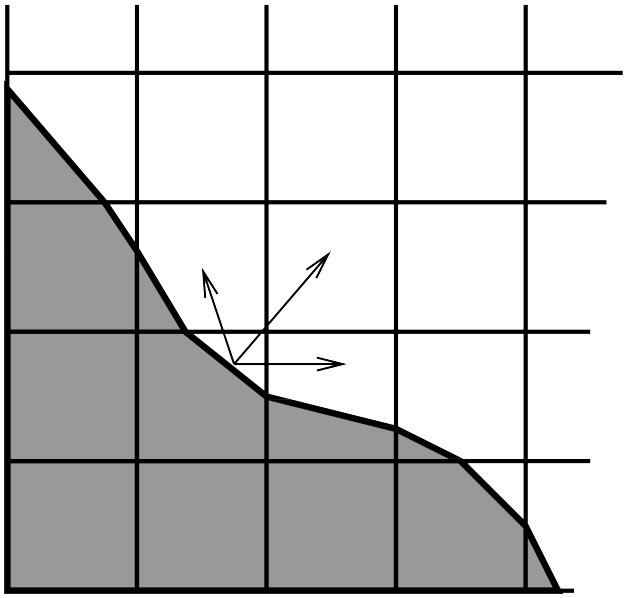


Fig. 12.2: : Redistribution illustration. Excess mass due to using a hybrid divergence D^H instead of the conservative divergence D^C is distributed to neighbor cells.

12.2 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);
```

12.2.1 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(...);
```

(continues on next page)

(continued from previous page)

```
// 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.});
```

12.2.2 EB2::GeometryShop

Given an implicit function object, say `f`, we can make a `GeometryShop` object with

```
auto shop = EB2::makeShop(f);
```

12.2.3 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.

12.3 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,
                  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 `MultiFabs` 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>
}
```

12.4 EB 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;
```

(continues on next page)

(continued from previous page)

```
// 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`, and it is zero for covered cells, one for regular cells, and in between for cut cells. Centroid is in a `MultiCutFab` with `AMREX_SPACEDIM` components with each component of the data is in the range of $[-0.5, 0.5]$. The centroid is based on each cell's local coordinates with respect to the embedded boundary. 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*). Boundary centroid is also a `MultiCutFab` with `AMREX_SPACEDIM` components, and it uses each cell's local coordinates. Area fractions and face centroids are returned in `Array` of `MultiCutFab` pointers. For each direction, area fraction is for the face of that direction. As for face centroids, 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 $[-0.5, 0.5]$.

12.4.1 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 and IArrayBox*). 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(...)
```

(continues on next page)

(continued from previous page)

```
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
```

12.5 Level Sets

In order to speed up direct interactions with embedded boundaries, AMReX also provides a way to construct level-sets representing the signed distance function from the closest EB surface. In our implementation, the level-set data is stored as a 1-component nodal `MultiFab` (cf. *FabArray, MultiFab and iMultiFab*) where each node stores its closest distance to the EB. The subroutine `amrex_eb_interp_levelset` (in `/Scr/EB/AMREX_EB_levelset_F.F90`) interpolates the level-set $\phi(\mathbf{r})$ to any position $\mathbf{r}$ from the pre-computed level-set `MultiFab`. Likewise the subroutine `amrex_eb_normal_levelset` interpolated the normal $\hat{\mathbf{n}}(\mathbf{r})$ at any position from the derivative of the level-set function $\hat{\mathbf{n}}(\mathbf{r}) = \nabla\phi(\mathrm{r})$. **Note** that since the normal is computed by taking the derivative of the interpolation function, it is discontinuous at positions corresponding to the nodal points of the level-set `MultiFab` (i.e. $\mathbf{r} = (i, j, k) \cdot h$).

At this point, AMReX does not provide a C++ interface for interpolating the level-set at a point. This is because so far the level-set was only needed while performing calculations in Fortran. The interpolation subroutines contained in `amrex_eb_levelset_module` are:

```
pure subroutine amrex_eb_interp_levelset(pos, plo,  n_refine, &
                                         phi, phlo, phhi,     &
                                         dx,  phi_interp    )
```

and

```
pure subroutine amrex_eb_normal_levelset(pos, plo,   n_refine, &
                                         phi, phlo,  phhi,     &
                                         dx,  normal         )
```

which interpolate the level-set value `phi_interp` and `normal`, respectively, at the 3-dimensional point `pos`. The nodal values of the level-set are given by the `phi` array. `dx/n_refine` is the refined cell-size of the level-set array. For example

```
use iso_c_binding    , only : c_int
use amrex_fort_module, only : c_real => amrex_real
use amrex_eb_levelset_module, only: amrex_eb_interp_levelset

! ** level-set data
!    philo, phihi - dimensions of phi array
```

(continues on next page)

(continued from previous page)

```
!    dx           - spatial discretization
!    n_refine     - refinement of phi array (wrt to dx)
integer(c_int) :: philo(3), phihi(3)
real(c_real)   :: phi(  phlo(1):phhi(1), phlo(2):phhi(2), phlo(3):phhi(3) )
real(c_real)   :: dx(3)
integer(c_int) :: n_refine

! ** interpolated level-set
!    pos      - coordinate where to interpolate
!    ls_value - interpolated level-set value (output)
real(c_real) :: pos(3), ls_value

call amrex_eb_interp_levelset(pos, plo, n_refine, phi, phlo, phhi, dx, ls_value);
```

AMReX provides collection of functions and subroutines to fill single and multi-level level-set data. For convenience, the `amrex::LSFactory` helps manage the level-set data for a single AMR level. And `amrex::LSCore` manages multi-level level-set data. These are described in further detail below.

12.5.1 A Note on Filling Level-Sets from `EBFArrayBoxFactory`

The data stored in a `EBFArrayBoxFactory`, represents the embedded boundary as a discrete collection of volume fractions, and area fractions over a grid. Here this is further simplified by thinking of the EB as a collection of planar facets. This means that for any given node in a grid, the nearest EB facet might be in another grid. Hence if the `EBFArrayBoxFactory` has `n_pad` ghost cells, then for any given grid, there could be EB facets that are `n_pad + 1` cells away, yet we would *not* "see". In other words, if the `EBFArrayBoxFactory` is defined on a grid with spacing h, then, and we do not have any EB facets in the current grid, then any node within that grid is *at least* $(n_{\text{pad}} + 1)h$ away from the nearest EB surface.

Hence, when filling a level-set, it will "max-out" at $\pm(n_{\text{pad}} + 1)h$. Hence it is recommended to think of this kind of level-set function as the point being "at least" $\phi(\mathbf{r})$ from the EB surface.

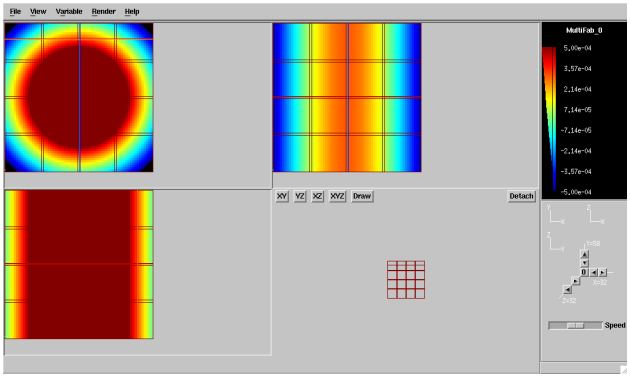


Fig. 12.3: : Example of a "local" level-set representing a cylinder. The level-set function is a (linear) signed distance function near the EB-surface, and it plateaus further away from it.

Figure Fig. 12.3 shows an example of such a local level-set description for a cylinder. Only cells that are within $\pm(n_{\text{pad}} + 1)h$ of the EB surface are filled with a level-set. The rest is filled with lower (upper) bound. If the goal is capture interactions between the EB surface and a point somewhere else, this approach usually suffices as we only need to know if we are "far enough" from the EB in most applications.

Since finding the closest distance between a point and an arbitrary surface is computationally expensive, we advice that `n_pad` is chosen as the smallest necessary number for the application.

12.5.2 Filling Level-Sets without `LSFactory`

The static function `amrex::LSFactory::fill_data` (defined in `Src/EB/AMReX_EB_levelset.cpp`) fills a `MultiFab` with the nodal level-set values and another `iMultiFab` with integer tags that are 1 whenever a node is near the EB surface. It is then left up to the application to manage the level-set `MultiFab`.

AMReX defines embedded surfaces using implicit functions (see above). Normally these implicit functions are usually *not* signed distance functions (i.e. their value at $\mathbf{r}$ is not the minimal distance to the EB surface). However, in rare cases such as the `EB2::PlaneIF`, it is. In this case, the most straight-forward way to fill a level-set. If an signed-distance implicit function is know, and stored as a `MultiFab mf_impfunc`, then we can use

```
static void fill_data (MultiFab & data, iMultiFab & valid,
                       const MultiFab & mf_impfunc,
                       int eb_pad, const Geometry & eb_geom);
```

so then the function call

```
// Fill implicit function
GShopLSFactory<EB2::CylinderIF> cylinder_lsgs(cylinder_ghsop, geom, ba, dm, 0);
std::unique_ptr<MultiFab> cylinder_mf_impfunc = cylinder_lsgs.fill_impfunc();


MultiFab ls_grid(ba, dm, 1, 0);
iMultiFab ls_valid(ba, dm, 1, 0);
amrex::LSFactory::fill_data(ls_grid, ls_valid, mf_impfunc, 2, geom_eb);
```

fills a `MultiFab ls_grid` with level-set data given the implicit function stored in the `MultiFab mf_impfunc`, and a threshold of `2*geom_eb.CellSize()`. The helper class `GShopLSFactory` converts EB2 implicit functions to `MultiFabs` (defined in `Src/EB/AMReX_EB_levelset.H`).

The much more interesting application of `amrex::LSFactory::fill_data` is filling a level-set given a `EBFArrayBoxFactory`:

```
static void fill_data (MultiFab & data, iMultiFab & valid,
                       const EBFArrayBoxFactory & eb_factory,
                       const MultiFab & eb_impfunc,
                       const IntVect & ebt_size, int ls_ref, int eb_ref,
                       const Geometry & geom, const Geometry & geom_eb);
```

which fills the `MultiFab data` with level-set data from the `EBFArrayBoxFactory eb_factory`. Here the user must still supply the EB implicit function using the `MultiFab eb_impfunc`, as this is used to determine the inside/outside when no EB facets can be found, or in special edge-cases. The user also needs to specify the tile size (`IntVect ebt_size`), the level-set and EB refinement (i.e. the grid over which `data` is defined is refined by a factor of `ls_ref/eb_ref` compared to the `eb_factory` 's grid), and the Geometries `geom` and `geom_eb` corresponding to the grids of `data` and `eb_factory` respectively.

When filling `data`, a tile-size of `ebt_size` is used. Only EB facets within a tile (plus the `eb_factory` ghost cells) are considered. Hence, chosing an appropriate `ebt_size` can significantly increase performance.

For example, the following fills a level-set with a cylinder EB (like that shown in Fig. Fig. 12.3).

```
// Define nGrow of level-set and EB
int ls_pad = 1;
int eb_pad = 2;

// Define EB
EB2::CylinderIF cylinder(radius, centre, true);
EB2::GeometryShop<EB2::CylinderIF> cylinder_gshop(cylinder);
```

(continues on next page)

(continued from previous page)

```
// Build EB
EB2::Build(cylinder_gshop, geom, max_level, max_level);
const EB2::IndexSpace & cylinder_ebis = EB2::IndexSpace::top();
const EB2::Level &      cylinder_lev  = cylinder_ebis.getLevel(geom);

// Build EB factory
EBFArrayBoxFactory eb_factory(cylinder_lev, geom, ba, dm, {eb_pad, eb_pad, eb_pad});

// Fill implicit function
GShopLSFactory<EB2::CylinderIF> cylinder_lsgs(cylinder_ghsop, geom, ba, dm, ls_pad);
std::unique_ptr<MultiFab> cylinder_mf_impfunc = cylinder_lsgs.fill_impfunc();

// Fill level-set
MultiFab ls_grid(ba, dm, 1, ls_pad);
iMultiFab ls_valid(ba, dm, 1, ls_pad);
LSFactory::fill_data(ls_grid, ls_valid, eb_factory, * cylinder_mf_impfunc,
                     ebt_size, 1, 1, geom, geom);
```

Note that in theory the `EBFArrayBoxFactory eb_factory` could be defined on a different resolution as the the `BoxArray ba`. In this case, the appropriate refinements and geometries must be specified. Also note that the thresholding behaviour (due to `eb_pad`) is specified via the `EBFArrayBoxFactory` constructor. The implicit function MultiFab needs to have the same grids as *data*.

Since this relies on the interplay of many different parameters, a number of utility functions and helper classes have been created. These are discussed in the subsequent sections.

The common operations of intersections and unions (similar to EB implicit functions, discussed in *Implicit Function*) can also be applied to level-sets. Without the use of a `LSFactory`, the functions:

```
static void intersect_data (MultiFab & data, iMultiFab & valid,
                            const MultiFab & data_in, const iMultiFab & valid_in,
                            const Geometry & geom_ls);
```

and

```
static void union_data (MultiFab & data, iMultiFab & valid,
                        const MultiFab & data_in, const iMultiFab & valid_in,
                        const Geometry & geom_ls);
```

These apply the intersection (element-wise minimum) and union (maximum) between the `MultiFab data`, and `data_in`. The result overwrites the contents of `data`. The tags stored in the `iMultiFab valid_in` determine where the intersection takes place (i.e. only cells where both `valid_in == 1` are intersected, others are ignored).

12.5.3 Using `LSFactory`

In the previous section, we've seen that the level-set and EB grids can exist on different levels of refinement. The practical reason behind this is that sometimes we want to capture interactions that are very sensitive close to EBs, but this can sometimes be difficult to keep track of. Hence the `LSFactory` can be helpful in taking care of all of these parameters.

The basic principle of the `LSFactory` (defined in `Src/EB/AMReX_EB_levelset.H`) is that it is created relative to some reference `BoxArray ba`, `Geometry geom`, and `DistributionMapping dm`. The user then specifies refinement factors `ls_ref` of the level-set data and `eb_ref` of the EB grid. Calling the constructor:

```
LSFactory(int lev, int ls_ref, int eb_ref, int ls_pad, int eb_pad,
          const BoxArray & ba, const Geometry & geom, const DistributionMapping & dm,
          int eb_tile_size = 32);
```

Then creates all appropriate grids and geometries. Note that we can also specify the tile size used internally in the `LSFactory::fill_data` function.

When a `LSFacotry` is first created, its level-set values are set to `huge(amrex_real)`. I. e. there are no surfaces, and so the level-set value is effectively infinite. It can then be filled just like in the previous section:

```
// Define refinement of level-set and EB
int ls_ref = 4;
int eb_ref = 1;

// Define nGrow of level-set and EB
int ls_pad = 1;
int eb_pad = 2;

// Define EB
EB2::CylinderIF cylinder(radius, centre, true);
EB2::GeometryShop<EB2::CylinderIF> cylinder_gshop(cylinder);

// Build level-set factory
LSFactory level_set(0, ls_ref, eb_ref, ls_pad, eb_pad, ba, geom, dm);

// Build EB
const Geometry & eb_geom = level_set.get_eb_geom()
EB2::Build(cylinder_gshop, eb_geom, max_level, max_level);

const EB2::IndexSpace & cylinder_ebis = EB2::IndexSpace::top();
const EB2::Level &      cylinder_lev  = cylinder_ebis.getLevel(eb_geom);

// Build EB factory
EBFArrayBoxFactory eb_factory(cylinder_lev, eb_geom, level_set.get_eb_ba(), dm,
                              {level_set.get_eb_pad(), level_set.get_eb_pad(),
                               level_set.get_eb_pad()});

// Fill level-set (factory)
GShopLSFactory<EB2::CylinderIF> cylinder_lsgs(cylinder_ghsop, level_set);
std::unique_ptr<MultiFab> cylinder_mf_impfunc = cylinder_lsgs.fill_impfunc();
level_set.Fill(eb_factory, * cylinder_mf_impfunc);
```

where the level-set data can now be accessed using:

```
const MultiFab * level_set_data = level_set.get_data();
```

or alternatively a copy of the data can be generated using:

```
std::unique_ptr<MultiFab> level_set_data = level_set.copy_data();
```

Both of the data above are on grids that have been refined by `ls_ref` (with respect to the `BoxArray ba`). In order to get a copy of the level-set data at the coarseness of the original grids, use:

```
std::unique_ptr<MultiFab> level_set_data_crse = level_set.coarsen_data();
```

Note however, that the level-set data is nodal data. Therefore, even though the `MultiFab level_set_data_crse` is defined on a grid with the same resolution as the `BoxArray ba`, it is defined on the nodal version of that grid.

The LSFactory is also there to make operations on the level-set easier. Intersection and Union operations with EB factories and implicit functions are available in the LSFactory class. As well as functions to regrid (updating the underlying BoxArray and DistributionMapping), copying, and inverting the level-set function.

12.5.4 Filling Multi-Level Level-Sets without LSCore

AMReX also provides code to fill the level-set function on different levels of refinement. The static function amrex::LSCoreBase::FillLevelSet, amrex::LSCoreBase::MakeNewLevelFromCoarse, and amrex::LSCoreBase::FillVolfracTags (or amrex::LSCoreBase::FillLevelSetTags for level-set tagging instead of volume-fraction tagging) fill a finer level from a coarse one. Just like the section on *Filling Level-Sets without LSFactory*, the philosophy here is to enable to user to fill a MultiFab with level-set values, and manage this data structure themselves. Later we will discuss the LSCore class, which automatically constructs multi-level level-sets.

One common problem with level-set function is that they are expensive to compute. Therefore, a strategy would be to compromise by computing the level-set function accurately near embedded boundaries (where precision is important), and at a lower resolution for from walls. The function

```
static void FillVolfracTags( int lev, TagBoxArray & tags,
                             const Vector<BoxArray> & grids,
                             const Vector<DistributionMapping> & dmap,
                             const EB2::Level & eb_lev, const Vector<Geometry> &␣
↪geom );
```

fills a TagBoxArray with tags wherever the volume fraction is between 0 and 1. This way any cut-cells a buffered of amr.n_error_buf many neighbors is tagged for refinement. If we need finer control over the tagging, the function

```
static void FillLevelSetTags( int lev, TagBoxArray & tags, const Vector<Real> &␣
↪phierr,
                              const MultiFab & levelset_data, const Vector<Geometry> &
↪ geom );
```

takes a list of threshold level-set values (Vector<Real> & phierr) and tags cells for refinement if the coarse estimate of the levelset (levelset_data) from level lev is less than phierr[lev].

The following code would then fill a multi-level hierarchy of level-sets contained in Vector<MultiFab> level_sets.

```
//_____________________________________________________________________
// Start with level zero

EBFArrayBoxFactory eb_factory(* eb_levels[0], geom[0], grids[0], dmap[0],
                              {eb_pad, eb_pad, leb_pad}, EBSupport::full);

// NOTE: reference BoxArray is not nodal
BoxArray nd_ba = amrex::convert(grids[0], IntVect::TheNodeVector());

level_sets[0].define(nd_ba, dmap[0], 1, pad);
iMultiFab valid(nd_ba, dmap[0], 1, pad);

// NOTE: implicit function data might not be on the right grids
MultiFab impfunc = MFUtil::regrid(nd_ba, dmap[0], implicit_functions[0], true);

LSFactory::fill_data(level_sets[0], valid, ebfactory, impfunc,
                     32, 1, 1, geom[0], geom[0]);
```

(continues on next page)

(continued from previous page)

```
//_____________________________________________________________________________
// Fill finer levels, using coarser level to estimate level-set

for (int lev = 1; lev < nlev; lev++) {
        // NOTE: reference BoxArray is not nodal
        BoxArray ba = amrex::convert(grids[lev], IntVect::TheNodeVector());
        level_sets[lev].reset(new MultiFab);
        iMultiFab valid(ba, dmap[lev], 1, pad);

        // Fills level_sets[lev] with coarse data
        LSCoreBase::MakeNewLevelFromCoarse( level_sets[lev], level_sets[lev-1],
                                            ba, dmap[lev], geom[lev], geom[lev-1],
                                            bcs_ls, refRatio(lev-1));

        EBFArrayBoxFactory eb_factory(* eb_levels[lev], geom[lev], grids[lev],␣
↪dmap[lev],
                                      {eb_pad, eb_pad, eb_pad}, EBSupport::full);

        // NOTE: implicit function data might not be on the right grids
        MultiFab impfunc = MFUtil::regrid(ba, dmap[lev], implicit_functions[lev]);

        IntVect ebt_size{AMREX_D_DECL(32, 32, 32)}; // Fudge factors...
        LSCoreBase::FillLevelSet(level_sets[lev], level_sets[lev], eb_factory,␣
↪impfunc,
                                 ebt_size, eb_pad, geom[lev]);
    }
```

Here the `Vector<const EB2::Level *> eb_levels` has been filled while initializing the embedded boundaries. At the same time, the implicit functions need to be saved to `Vector<MultiFab> implicit_functions`. The user also needs to specify the level-set boundary conditions in `Vector<BCRec> bcs_ls`. Note that the function `LSCoreBase::FillLevelSet` uses the coarse level-set as an upper bound to the tile size used for testing EB facets.

12.5.5 Using `LSCore`

The process described in the previous section is automated in the `LSCore` class. It is derived from `LSCoreBase`, which in turn is derived from `AmrCore` (cf. *AmrCore Source Code*). `LSCore` is a template class depending on the embedded boundary implicit function. This way, it can build new `EB2::Level` objects for every new level that is needed.

Since `LSCore` is a template class, it might lead to problems in applications where the template parameter can depend of runtime parameters. This is the reason why it derives from the base class `LSCoreBase`. `LSCore` overwrites the virtual function `MakeNewLevelFromScratch` in `LSCoreBase`. The application can then employ the following polymorphism to construct the level-set;

```
LSCoreBase * ls_core;

// sets ls_core pointer
make_my_eb(ls_core);

ls_core->InitData();
```

where the function `make_my_eb` defines the actual EB geometry:

```
void make_my_eb(LSCoreBase *& ls_core) {

    // MyIF is an EB2 Implicit Fuction
    GeometryShop<MyIF> gshop;

    // Build an EB geometry shop here

    ls_core = new LSCore<MyIF>(gshop);
}
```

Here the `make_my_eb` is only defines the EB geometry. The function call `ls_core->InitData()` constructs level hierarchy and fills it with level-set values.

12.6 Linear Solvers

Linear solvers for the canonical form (equation (9.1)) have been discussed in chapter *Linear Solvers*. Currently, AMReX supports cell-centered solver with homogeneous Neumann boundary condition on the EB. A cell-centered solver with Dirichlet boundary condition on the EB and a nodal solver are under development.

To use cell-centered solver for EB, one builds linear operator `MLEBABecLap` with `EBFArrayBoxFactory`.

```
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 specified class is essentially the same as `MLABecLaplacian`.

12.7 Tutorials

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

CHAPTER

THIRTEEN

GPU

In this chapter, we will present the GPU support in AMReX. Currently AMReX only supports Nvidia GPUs. Internally, it uses CUDA C++, but the users can use CUDA Fortran and/or OpenACC. A recent version of CUDA (e.g., >= 9) is required, and the device must have compute capability >= 6.

For complete details of CUDA, CUDA Fortran and OpenACC languages, see their respective documentations.

A number of tutorials can be found at `Tutorials/GPU/`.

13.1 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 perfomance tuning and programming model selection. AMReX uses CUDA for GPUs, but application teams can use CUDA, CUDA Fortran, OpenACC or OpenMP in their individual codes.

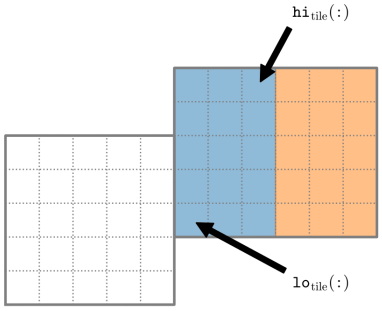
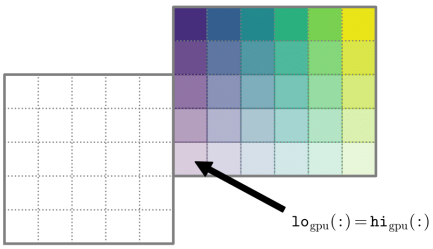
When running AMReX on a CPU system, the parallelization strategy is a combination of MPI and OpenMP using tiling, as detailed in *MFIter with Tiling*. However, tiling is ineffective on GPUs due to the overhead associated with kernel launching. Instead, efficient use of the GPU's resources is the primary concern. Improving resource efficiency allows a larger percentage of GPU threads to work simultaneously, increasing effective parallelism and decrease 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 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 CUDA threads to parallelize over a valid box at a time. This is done by using a lot of CUDA threads that only work on a few cells each. This work distribution is illustrated in Table 13.1. (Note: OpenMP is currently incompatible with AMReX builds using CUDA. This feature is under development, although most applications will have no need for this feature when initially converting to GPUs.)

Table 13.1: Comparison of OpenMP and CUDA 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.	CUDA threaded box. Each CUDA 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 CUDA extended lambdas are used to provide performance portability while making the code as understandable as possible to science-focused code teams.
- AMReX utilizes CUDA 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 temporaries, 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 with the exception of redistribution and I/O operations.
- AMReX's GPU strategy is focused on launching GPU kernels inside `MFIter` loops. By performing GPU work within `MFIter` loops, GPU work is isolated to independent data sets on simple AMReX data objects, providing consistency and safety that matches AMReX's coding methodology.
- AMReX further parallelizes GPU applications by utilizing CUDA streams. CUDA guarantees 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 iterations to be run simultaneously and maximize available GPU resources.

  The AMReX implementation of CUDA streams is illustrated in Fig. 13.1. 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.

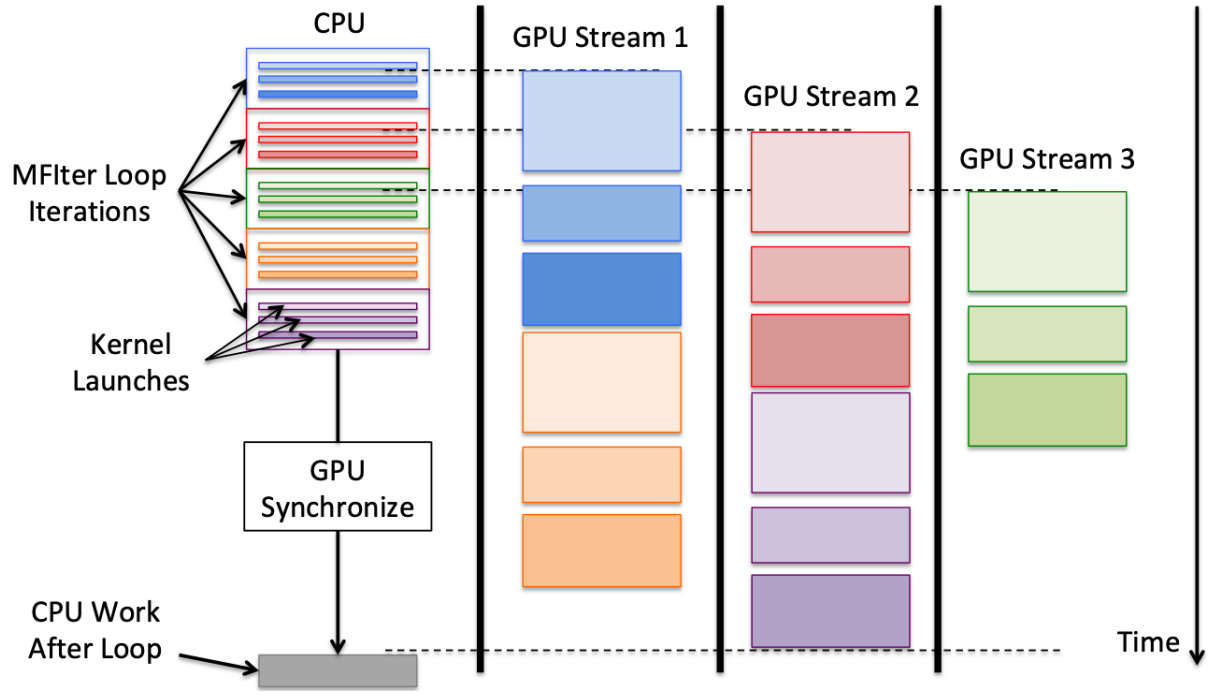


Fig. 13.1: Timeline illustration of GPU streams. Illustrates the case of an MFIter loop of five iterations with three GPU kernels each being ran with three GPU streams.

13.2 Building GPU Support

13.2.1 Building with GNU Make

To build AMReX with GPU support, add `USE_CUDA=TRUE` to the `GNUmakefile` or as a command line argument.

AMReX does not require OpenACC or CUDA Fortran, 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`.

Only IBM and PGI support CUDA Fortran, which is also built when `USE_CUDA=TRUE`.

Currently, only IBM is supported with OpenMP offloading. To use OpenMP offloading, make with `USE_OMP_OFFLOAD=TRUE`. .. NOTE: IBM OpenMP is currently not supported with CUDA.

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 `Tutorials/Basic/HelloWorld_C` to test your programming environment. 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
CUDA initialized with 1 GPU
AMReX (18.12-95-gf265b537f479-dirty) initialized
Hello world from AMReX version 18.12-95-gf265b537f479-dirty
[The         Arena] space (kilobyte): 8192
[The  Device Arena] space (kilobyte): 8192
[The Managed Arena] space (kilobyte): 8192
[The  Pinned Arena] space (kilobyte): 8192
AMReX (18.12-95-gf265b537f479-dirty) finalized
```

13.2.2 Building with CMake

CMake is currently unavailable when building with GPUs.

13.3 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`. Other important objects in the Gpu namespace include objects designed to work with GPU memory spaces, such as `AsyncFab` a temporary `FArrayBox` designed to work with CUDA streams.

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, 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. For PGI and IBM compilers, `AMREX_USE_CUDA_FORTRAN` is also defined, as well as `-DAMREX_CUDA_FORT_GLOBAL='attributes(global)'`, `-DAMREX_CUDA_FORT_DEVICE='attributes(device)'`, and `-DAMREX_CUDA_FORT_HOST='attributes(host)'` so that CUDA Fortran functions can be properly labelled. When AMReX is compiled with `USE_ACC=TRUE`, `AMREX_USE_ACC` 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.

13.4 Memory Allocation

To provide portability and improve memory allocation performance, AMReX provides a number of memory pools. When compiled without CUDA, all `Arenas` use standard `new` and `delete` operators. With CUDA, the `Arenas` each allocate with a specific type of GPU memory:

Table 13.2: Memory Arenas

Arena	Memory Type
The_Arena()	unified memory
The_Device_Arena()	device memory
The_Managed_Arena()	unified memory
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`. Therefore the data in a `MultiFab` is placed in unified memory and is accessible from both CPU host and GPU device. This allows application codes to develop their GPU capability gradually. `The_Managed_Arena()` is a separate pool of unified memory, that is distinguished from `The_Arena()` for performance reasons. If you want to print out the current memory usage of the Arenas, you can call `amrex::Arena::PrintUsage()`.

13.5 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.

13.5.1 GpuArray

`std::array` is used throughout AMReX, however its functions are not defined in device code. `GpuArray` is AMReX's built-in alternative. It is a POD (plain old data structure) that can be passed to the device by value and has device functions for the `[]` operator, `size()` and a `data()` function that returns a pointer to the underlying data. `GpuArray` can be used whenever a fixed size array needs to be passed to the GPU. `GpuArray` is also portable; when compiled without CUDA, it is simply aliased to a `std::array`.

A variety of functions have been created to return `GpuArray` instead of `std::array`, and allow direct access to GPU-ready data structures from common AMReX classes. For example, `GeometryData::CellSizeArray()`, `GeometryData::InvCellSizeArray()` and `Box::length3d()` all return `GpuArrays`.

13.5.2 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 is a one-time container, designed to take an initial value for the objects, move it to the GPU, work in a manner that allows full asynchronously between the CPU and GPU, and return a final value back to the device. If the data needs to be returned to the GPU again, it will be necessary to build a new AsyncArray.

The call to the destructor of AsyncArray is added to the GPU stream as a callback function. This guarantees the AsyncArray built in each loop iteration continues to exist until after all GPU kernels are completed without forcing the code to become serialized. 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.

A AsyncArray is used by constructing it from a reference to a host object containing an initial value, retrieving the associated device pointer, passing the pointer into an device function and copying the final value back to the CPU. An example using AsyncArray is given below, which finds the avarage value of all the boundary cells of a MultiFab:

```
// Previously defined MultiFab "multiFab".
// Find the average value of boundary cells.
{
    Real avg_val = 0;
    AsyncArray<Real> a_avg(&avg_val, 1);     // Build AsyncArray
    Real* d_avg = a_avg.data();                  // Get associated device ptr.

    long total_cells = 0;

    for (MFIter mfi(multiFab); mfi.isValid(); ++mfi)
    {
        const Box& gbx = mfi.fabbox();
        const Box& vbx = mfi.validbox();
        BoxList blst = amrex::boxDiff(gbx,vbx);
        const int nboxes = blst.size();
        if (nboxes > 0)
        {
            // Create AsyncArray for boxes describing boundary and
            //    obtain associated device pointer.
            AsyncArray<Box> async_boxes(blst.data().data(), nboxes);
            Box const* pboxes = async_boxes.data();

            long ncells = 0;
            for (const auto& b : blst) {
                ncells += b.numPts();
            }
            total_cells += ncells;

            const FArrayBox* fab = multiFab.fabPtr(mfi);
            amrex::ParallelFor ( ncells,
            [=] AMREX_GPU_DEVICE(long icell)
            {
                // Use async_boxes to calc cell for this thread.
                const Dim3 cell = amrex::getCell(pboxes, nboxes, icell).dim3();
                for (int n = strt_comp; n < strt_comp+ncomp; ++n)
                {
                    *d_avg += fab(cell.x,cell.y,cell.z,n);   // Add cell value to␣
↪total.
```

(continues on next page)

(continued from previous page)

```
                }
            });

        }
    }
    a_avg.copyToHost(&avg_val, 1);          // Return d_avg value to host in avg_val.

    avg_val = avg_val / total_cells;
}
```

Note that there are two `AsyncArrays`: one which is constructed outside the `MFIter` loop and stores the sum of the cell values and one that is constructed inside that stores the data from the `BoxArray` that defines the boundary. `avg_val` needs to be returned after the sum of cell values is completed, so an explicit call to `copyToHost` is made after the loop, but the `async_boxes` is only used on the device, so no return call is needed.

13.5.3 ManagedVector

AMReX also provides a dynamic memory allocation object for GPU managed memory: `Gpu::ManagedVector`. This class behaves identically to an `amrex::Vector`, (see *Vector and Array*), except the vector's allocator has been changed to allocate and deallocate its data in CUDA managed memory whenever `USE_CUDA=TRUE`.

While the data is managed and available on GPUs, the member functions of `Gpu::ManagedVector` are not. To use the data on the GPU, it is necessary to pass the underlying data pointer to the GPU. 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.

13.5.4 CUDA's Thrust Vectors

CUDA's Thrust library can also be used to manage dynamically sized data sets. However, if Thrust is used directly in AMReX code, it will be unable to compile for cases when `USE_CUDA=FALSE`. To alleviate this issue, `thrust::host_vector` and `thrust::device_vector` have been wrapped into the AMReX classes `Gpu::HostVector` and `Gpu::DeviceVector`. When `USE_CUDA=FALSE`, these classes revert to AMReX's Vector class. When `USE_CUDA=TRUE`, these classes become the corresponding Thrust vector.

Just like with Thrust vectors, `HostVector` and `DeviceVector` cannot be directly used on the device. For convenience, the `dataPtr()` member function has been altered to implement `thrust::raw_pointer_cast` and return the raw data pointer which can be used to access the vector's underlying data on the GPU.

It has proven useful to have a version of Thrust's `device_vector` that uses CUDA managed memory. This is provided by `Gpu::ManagedDeviceVector`.

`thrust::copy` is also commonly used in AMReX applications. It can be implemented portably using `Gpu::thrust_copy`.

`Gpu::DeviceVector` and `Gpu::ManagedDeviceVector` are configured to use the memory Arenas provided by AMReX (see sec:gpu:memory:). This means that you can create temporary versions of these containers on-the-fly without needing to performance expensive device memory allocate and free operations.

13.5.5 amrex::min and amrex::max

GPU versions of `std::min` and `std::max` are not provided in CUDA. So, AMReX provides a templated `min` and `max` with host and device versions to allow functionality on GPUs. Invoke the explicitly namespaced `amrex::min(A, B)` or `amrex::max(x, y)` to use the GPU safe implementations. These functions are variadic, so they can take any number of arguments and can be invoked with any standard data type.

13.5.6 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 templates `amrex::ReduceSum`, `amrex::ReduceMin` and `amrex::ReduceMax` can be used to implement user-defined reduction functions over `MultiFabs`. These same templates are implemented in the `MultiFab` functions, so they can be used as a reference to build a custom reduction. For example, the `MultiFab:Dot` implementation is reproduced here:

```
Real sm = amrex::ReduceSum(x, y, nghost,
[=] AMREX_GPU_HOST_DEVICE (Box const& bx, FArrayBox const& xfab, FArrayBox const&␣
↪yfab) -> Real
{
    return xfab.dot(bx,xcomp,yfab,bx,ycomp,numcomp);
});

if (!local) ParallelAllReduce::Sum(sm, ParallelContext::CommunicatorSub());

return sm;
```

`amrex::ReduceSum` takes two `MultiFabs`, `x` and `y` and returns the sum of the value returned from the given lambda function. In this case, `BaseFab::dot` is returned, yielding a sum of the dot product of each local pair of `BaseFabs`. Finally, `ParallelAllReduce` is used to sum the dot products across all MPI ranks and return the total dot product of the two `MultiFabs`.

To implement a different reduction, replace the code block inside the lambda function with the operation that should be applied, being sure to return the value to be summed, minimized, or maximized. The reduction templates have a few different interfaces to accomodate a variety of reductions. The `amrex::ReduceSum` reduction template has varieties that take either one, two or three :`MultiFabs`. `amrex::ReduceMin` and `amrex::ReduceMax` can take either one or two.

13.5.7 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.

13.5.8 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. To utilize `Geometry` on the GPUs, the data is copied 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`.

13.5.9 BaseFab, FArrayBox, IArrayBox and AsyncFab

`BaseFab<T>`, `IArrayBox` and `FArrayBox` have some GPU support. They cannot be constructed in device code, but a pointer to them can be passed to GPU kernels from CPU code. Many of their member functions can be used in device code as long as they have been allocated in device memory. Some of the device member functions include `view`, `dataPtr`, `box`, `nComp`, and `setVal`.

All `BaseFab<T>` objects in `FabArray<FAB>` are allocated in unified memory, including `IArrayBox` and `FArrayBox`, which are derived from `BaseFab`. A `BaseFab<T>` object created on the stack in CPU code cannot be used in GPU device code, because the object is in CPU memory. However, a `BaseFab` created with `new` on the heap is GPU safe, because `BaseFab` has its own overloaded `operator new` that allocates memory from `The_Arena()`, a managed memory arena. For example,

```
// We are in CPU code

FArrayBox cpu_fab(box,ncomp);
// FArrayBox* p_cpu_fab = &(cpu_fab) cannot be used in GPU device code!

FArrayBox* p_gpu_fab = new FArrayBox(box,ncomp);
// FArrayBox* p_gpu_fab can be used in GPU device code.
```

Temporary `FArrayBox`es are also available for GPU work through the `AsyncFab` class. `AsyncFab`s are async-safe and should be used whenever a temporary `FArrayBox` is needed for intermediate calculations on the GPU.

It behaves similarly to the *AsyncArray*. It contains pointers for the CPU and GPU `FArrayBox` and storage for the associated metadata to minimize data movement. The `AsyncFab` is async-safe and can be used inside of an `MFIter` loop without reducing CPU-GPU asynchronicity. It is portable, reducing to a simple `FArrayBox` pointer when ran without CUDA. An example of using `AsyncFab` is given below:

```
for (MFIter mfi(some_multifab); mfi.isValid(); ++mfi)
{
   const Box& bx = mfi.validbox();

   // Create temporary FAB with given box & number of components.
   AsyncFab q_box(bx, 1);
   FArrayBox* q_fab = q_box.fabPtr();  // Get device pointer to fab.

   amrex::launch(bx,
   {
       Calcs q_fab
   });

   amrex::launch(bx,
   {
       Uses q_fab
   });

   q_box.clear();  // Added to the stream's kernel launch stack.
                   // So, q_box remains until completed.

   amrex::launch(bx,
   {
       More work w/o q_fab.
   }
}
```

13.5.10 MultiFabs and Accessing FArrayBoxes

`MultiFabs` CANNOT be constructed or moved onto the GPU. However, the underlying `FArrayBox`es are automatically managed during the `MultiFab`'s construction. The associated metadata has two copies, one on the CPU and one managed copy designed to live on the GPU, each accessed with a different `MultiFab` member function. Users should always use the appropriate accessor to minimize data movement and optimize performance.

To access the CPU `FArrayBox` reference, use `operator[]`.

To access the GPU `FArrayBox` managed pointer, use `fabPtr()`.

```
// Multifab mf( .... );

for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox& fab = mf[mfi];                 // CPU version.
    FArrayBox* d_fab_ptr = mf.fabPtr(mfi);    // GPU version.
}
```

13.6 Kernel Launch

In this section, how to offload work to the GPU will be demonstrated. AMReX supports offloading work with CUDA, CUDA Fortran, 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 *Box*/es 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 at `amrex/Tutorials/GPU/Launch`. 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 converts the 1D floating point data array into a simple to understand 3D loop structure, similar in appearance to Fortran, while maintaining performance. The details can be found in *C++ Kernel*.

13.6.1 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. When running on GPUs, loops with a dependent ordering would run substanitally slower, due to the need for appropriate atomic operations across the GPU threads. 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 whereever possible to allow efficient utilization of GPU hardware.

However, it is important for applications to use these launches whenever appropriate because they contain numerous 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. Failing to use the nested loop launches where appropriate eliminate these optimizations and reduce performance on CPU, GPU or both systems.

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 to 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 documenation, however all code snippets will also include the macro variation for reference.

A 4D example of the launch function, ;cpp:*amrex::ParallelFor*, is given here:

```
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    Array4<Real> const& fab = mf.array(mfi);
    int ncomp = mf.nComp();

    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 Array4s 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 macro 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) {
```

(continues on next page)

(continued from previous page)

```
        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. Therefore, any local objects used in the lambda function must be non-reference objects, such as pointers.

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.;
    /   });
    */
}
```

The 3D loop launch can be used on a single component by calling the `fab` with a fixed component index: `fab(i,j,k,0)`.

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() * mf.nComp();

    amrex::ParallelFor(nitems,
    [=] AMREX_GPU_DEVICE (long idx)
    {
```

(continues on next page)

(continued from previous page)

```
        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]`.

13.6.2 Launching general kernels

To launch more general work on the GPU, AMReX provides a standard launch function: `amrex::launch`. Instead of creating an optimized 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` is a very small number of cells (~1 to 5) to allow all GPU threads to be utilized effectively.

An example of a generic function launch, including both a C++ and Fortran function launch, is shown here:

```
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& bx = mfi.tilebox();
    FArrayBox* fab = mf.fabPtr(mfi);

    amrex::launch(bx,
    [=] AMREX_GPU_DEVICE (Box const& tbx)
    {
        plusone_cudacpp(tbx, *fab);
        plusone_cudafort(BL_TO_FORTRAN_BOX(tbx),
                         BL_TO_FORTRAN_ANYD(*fab));
    }

    /* MACRO VARIATION
    /
    /   AMREX_LAUNCH_DEVICE_LAMBDA ( bx, tbx,
    /   {
    /       plusone_cudacpp(tbx, *fab);
    /       plusone_cudafort(BL_TO_FORTRAN_BOX(tbx),
    /                        BL_TO_FORTRAN_ANYD(*fab));
    /   });
}
```

`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. The `MultiFab::fabPtr` returns a CUDA managed pointer to the current `FArrayBox`, which is captured by the lambda function.

amrex::launch takes two arguments: a Box denoting the region to work over and the lambda function defining the work for each thread. The lambda function is passed the thread Box, which is calculated in the launch function and passed into the lambda. The user can select the name of the thread Box. In this example, tbx was used. Finally, the lambda also captures any local parameters needed to perform the designated work. CUDA lambda functions can only capture by value, as the information must be able to be copied onto the device. Therefore, any local objects used in the lambda function must be non-reference objects, such as pointers.

In this example, both a C++ and a Fortran function are called. These functions are labeled device functions using AMREX_GPU_DEVICE and AMREX_CUDA_FORT_DEVICE, respectively, but are otherwise identical to what would be ran on CPUs. In this example, each cell is incremented by two, first from a C++ function and then from a Fortran function. There is no guarantee the C++ function completes on all cells of a given FArrayBox before the Fortran function is implemented, but because all of the threads work on independent cells, there are no race conditions and the calculation works as expected.

13.6.3 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 MultiFab::operator[] is used to get a reference to FArrayBox rather than using MultiFab::fabPtr to get a pointer, as suggested for CUDA kernels. Using the reference is optimal when passing pointers to kernels is not required, which includes CPU code sections and pragma based GPU implementations.

The function plusone_acc is a CPU host function. The 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

!$acc kernels deviceptr(dat)
do       k = lo(3), hi(3)
   do    j = lo(2), hi(2)
      do i = lo(1), hi(1)
         dat(i,j,k) = dat(i,j,k) + 1.0_amrex_real
      end do
   end do
end do
!$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.

13.6.4 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::Device::synchronize()`.

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 indepenently, but in the expected sequence, and maximize the use of GPU parallelism.

However, AMReX uses the default CUDA stream outside of `MFIter` loops. The default stream implements implicit synchronization: it behaves as though a device synchronization was called before and after each launch. So, any launches placed outside of an `MFIter` loop will be fully synchronous across both the CPU and the GPU.

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 lamdba 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 class 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, b);    <----  Correct. Comma not interpreted by macro.
});
```

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 expected OpenMP strategy for GPUs is to utilize OpenMP in CPU regions to maintain multi-threaded parallelism on work that cannot be offloaded efficiently, while using CUDA independently in GPU regions. This means OpenMP pragmas need to be maintained when `USE_CUDA=FALSE` and turned off in locations CUDA is implemented when `USE_CUDA=TRUE`.

This can currently be implemented in preparation for an OpenMP strategy and users are highly encouraged to do so now. This prevents having to track down and label the appropriate OpenMP regions in the future and clearly labels for readers that OpenMP and GPUs are not being used at the same time. OpenMP pragmas can be turned off using the conditional pragma and `Gpu::notInLaunchRegion()`, as shown below:

```
#ifdef _OPENMP
#pragma omp parallel if (Gpu::notInLaunchRegion())
#endif
```

This should be added only to MFIter loops that contain GPU work.

13.7 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
FArrayBox q;
for (MFIter mfi(uin); mfi.isValid(); ++mfi)
{
    const Box& vbx = mfi.validbox();
    const Box& gbx = amrex::grow(vbx,1);
    q.resize(gbx);
```

(continues on next page)

(continued from previous page)

```
    // 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 vbx.
    f2(vbx, 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 (`vbx` 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 in Section *BaseFab, FArrayBox, IArrayBox and AsyncFab*, `FArrayBox`es in the two `MultiFab`s, `uin` and `uout`, are available on GPUs through unified memory. But `FArrayBox q` is built locally, so it is only available in host memory. Creating `q` as a managed object using the overloaded `new` operator:

```
FArrayBox* q = new FArrayBox;
```

does not solve the problem completely because GPU kernel calls are asynchronous from the CPU's point of view. This creates a race condition: GPU kernels in different iterations of `MFIter` will compete for access to `q`. One possible failure is a segfault when `resize` changes the size of the `q` object when the previous iteration of the loop is still using an old size.

Moving the line into the body of `MFIter` loop will make `q` a local variable to each iteration, but it has a new issue. When is `q` deleted? To the CPU, the resource of `q` should be freed at the end of the scope, otherwise there will be a memory leak. But at the end of the CPU scope, which is the end of each iteration of the `MFIter` loop, GPU kernels will still be performing work that needs it.

One way to fix this is put the temporary `FArrayBox` objects in a `MultiFab` defined outside the loop. This creates a separate `FArrayBox` for each loop iteration, eliminating the race condition. Another way is to use `AsyncFab` designed for this kind of situation. The code below shows how `AsyncFab` is used and how this MFIter loop can be rewritten for GPUs.

```
for (MFIter mfi(uin); mfi.isValid(); ++mfi)
{
    const Box& vbx = mfi.validbox();              // f2 work domain
    const Box& gbx = amrex::grow(vbx,1);          // f1 work domain
    AsyncFab q(gbx);                              // Local, GPU managed FArrayBox
    FArrayBox const* uinfab  = uin.fabPtr();      // Managed GPU capturable
    FArrayBox      * uoutfab = uout.fabPtr();     //   pointers to MultiFab's FABs.

    amrex::launch(gbx,                            // f1 GPU launch
    [=] AMREX_GPU_DEVICE (Box const& tbx)
    {
        f1(tbx, q.fab(), *uinfab);
    };

    amrex::launch(vbx,                            // f2 GPU launch
    [=] AMREX_GPU_DEVICE (Box const& tbx)
    {
        f2(tbx, *uoutfab, q.fab());
    });
}                                                 // Implicit GPU barrier after
                                                  //   all iters are launched.
```

13.8 Assertions, Error Checking and Synchronization

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 wraping 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::Device::synchronize()` and `Gpu::Device::streamSynchronize()` can be used to synchronize the device or the CUDA stream, respectively, and track down the specific launch that causes the error.

13.9 Particle Support

AMReX's GPU particle support relies on Thrust, a parallel algorithms library maintained by Nvidia. Thrust provides a GPU-capable vector container that is otherwise similar to the one in the C++ Standard Template Library, along with associated sorting, searching, and prefix summing operations. Combined with Cuda's unified memory, Thrust forms the basis of AMReX's GPU support for particles.

When compiled with `USE_CUDA=TRUE`, AMReX places all its particle data in instances of `thrust::device_vector` that have been configured using a custom memory allocator using `cudaMallocManaged`. This means that the `dataPtr` associated with particle data is managed and can be passed into GPU kernels, similar to the way it would be passed into a Fortran subroutine in typical AMReX CPU code. As with the mesh data, 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
    structs(ip)%pos(3) = structs(ip)%pos(3) + uzp(ip)*gaminv(ip)*dt
end do
!$acc end loop
```

(continues on next page)

(continued from previous page)

```
!$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 a complete example of a particle code that has been ported to GPUs using OpenACC, please see `Tutorials/Particles/ElectromagneticPIC`.

For portability, we have provided a set of Vector classes that wrap around the Thrust and STL vectors. When `USE_CUDA = FALSE`, these classes reduce to the normal `amrex::Vector`. When `USE_CUDA = TRUE`, they have different meanings. `Cuda::HostVector` is a wrapper around `thrust::host_vector`. `Cuda::DeviceVector` is a wrapper around `thrust::device_vector`, while `Cuda::ManagedDeviceVector` is a `thrust::device_vector` that lives in managed memory. These classes are useful when there are certain stages of an algorithm that will always execute on either the host or the device. For example, the following code generates particles on the CPU and copies them over to the GPU in one batch per tile:

```
for(MFIter mfi = MakeMFIter(lev); mfi.isValid(); ++mfi)
{
    const Box& tile_box  = mfi.tilebox();
    Cuda::HostVector<ParticleType> host_particles;

    for (IntVect iv = tile_box.smallEnd(); iv <= tile_box.bigEnd(); tile_box.next(iv))
    {
        < generate some particles... >
    }

    auto& particles = GetParticles(lev);
    auto& particle_tile = particles[std::make_pair(mfi.index(), mfi.
↪LocalTileIndex())];
    auto old_size = particle_tile.GetArrayOfStructs().size();
    auto new_size = old_size + host_particles.size();
    particle_tile.resize(new_size);

    Cuda::thrust_copy(host_particles.begin(),
                      host_particles.end(),
                      particle_tile.GetArrayOfStructs().begin() + old_size);
 }
```

The following example shows how to use `Cuda::DeviceVector`. Specifically, this code creates temporary device vectors for the particle x, y, and z positions, and then copies from an Array-of-Structs to a Struct-of-Arrays representation, all without copying any particle data off the GPU:

```
Cuda::DeviceVector<Real> xp, yp, zp;

for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti)
{
    pti.GetPosition(xp, yp, zp);

    < use xp, yp, zp... >
}
```

Note that the above code will cause problems if multiple streams are used to launch kernels inside the particle iterator loop. This is because the temporary variables `xp`, `yp`, and `zp` are shared between different iterations. However, if all the kernel launches happen on the default stream, so that the kernels are guaranteed to complete in order, then the

above approach will give the expected results.

Finally, AMReX's `Redistribute()`, which moves particles back to the proper grids after their positions have changed, has been ported to work on the GPU as well. It cannot be called from device code, but it can be called on particles that reside on the device and it won't trigger any unified memory traffic. As with `MultiFab` data, the MPI portion of the particle redistribute is set up to take advantange of the Cuda-aware MPI implementations available on platforms such as ORNL's Summit and Summit-dev.

13.10 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 webpages 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 asynchonous 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("MFIter: Init", mfinit);     // Profiling start
for (MFIter mfi(phi_new); mfi.isValid(); ++mfi)
{
    const Box& vbx = mfi.validbox();
    const GeometryData& geomdata = geom.data();
    FArrayBox* phiNew = phi_new.fabPtr(mfi);

    AMREX_LAUNCH_DEVICE_LAMBDA(vbx, tbx,
    {
        init_phi(BL_TO_FORTRAN_BOX(tbx),
                 BL_TO_FORTRAN_ANYD(*phiNew),
                 geomdata.CellSize(), geomdata.ProbLo(), geomdata.ProbHi());
    });

}
BL_PROFILE_STOP(mfinit);                    // Profiling stop
```

For now, this is the best way to profile GPU codes using `TinyProfiler`. If you require further profiling detail, use `nvprof`.

13.11 Performance Tips

Here are some helpful performance tips to keep in mind when working with AMReX for GPUs:

- It is important to use `fabPtr` and `operator[]` in the appropriate places to minimize unnecessary data movement. If a `FArrayBox` functions are called from a `FArrayBox*` created by on the device, the associated meta-data will be transferred back to the CPU, causing substantial slow-downs.
- 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 including their declarations in the `.H` file and using the `AMREX_INLINE` AMReX macro. Examples can be found in `Tutorials\GPU\Launch`. For example:

```
AMREX_GPU_DEVICE
AMREX_INLINE
void plusone_cudacpp (amrex::Box const& bx, amrex::FArrayBox& fab)
{
    const auto len = amrex::length(bx);  // length of box
    const auto lo  = amrex::lbound(bx);  // lower bound of box
    const auto data = fab.view(lo);  // a view starting from lo

    for          (int k = 0; k < len.z; ++k) {
        for      (int j = 0; j < len.y; ++j) {
            // We know this is safe for simd on cpu.  So let's give compiler some␣
↪help.
            AMREX_PRAGMA_SIMD
            for (int i = 0; i < len.x; ++i) {
                data(i,j,k) += 1.0;
            }
        }
    }
}
```

13.12 Limitations

GPU support in AMReX is still under development. There are some known limitations:

- By default, AMReX assumes the MPI library used is GPU aware. The communication buffers given to MPI functions are allocated in device memory.
- OpenMP is currently not compatible with building AMReX with CUDA. `USE_CUDA=TRUE` and `USE_OMP=TRUE` will fail to compile.
- CMake is not yet supported for building AMReX GPU support.
- Many multi-level functions in AMReX have not been ported to GPUs.
- Linear solvers have not been ported to GPUs.
- Embedded boundary capability has not been ported to GPUs.
- The Fortran interface of AMReX does not currently have GPU support.

CHAPTER

FOURTEEN

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.

14.1 Amrvis

Our favorite visualization tool is Amrvis. We heartily encourage you to build the `amrvis1d`, `amrvis2d`, and `amrvis3d` executables, and to try using them to visualize your data. A very useful feature is View/Dataset, which allows you to actually view the numbers in a spreadsheet that is nested to reflect the AMR hierarchy – this can be handy for debugging. You can modify how many levels of data you want to see, whether you want to see the grid boxes or not, what palette you use, etc. Below are some instructions and tips for using Amrvis; you can find additional information in Amrvis/Docs/Amrvis.tex (which you can build into a pdf using pdflatex).

1. Download and build :

   ```
   git clone https://github.com/AMReX-Codes/Amrvis
   ```

   Then cd into Amrvis/, edit the GNUmakefile by setting `COMP` to the compiler suite you have.

   Type `make DIM=1`, `make DIM=2`, or `make DIM=3` to build, resulting in an executable that looks like amrvis2d...ex.

   If you want to build amrvis with `DIM=3`, you must first download and build `volpack`:

   ```
   git clone https://ccse.lbl.gov/pub/Downloads/volpack.git
   ```

   Then cd into volpack/ and type `make`.

   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.

   You may then want to create an alias to amrvis2d, for example

   ```
   alias amrvis2d /tmp/Amrvis/amrvis2d...ex
   ```

2. Run the command `cp Amrvis/amrvis.defaults ~/.amrvis.defaults`. Then, in your copy, edit the line containing "palette" line to point to, e.g., "palette /home/username/Amrvis/Palette". The other lines

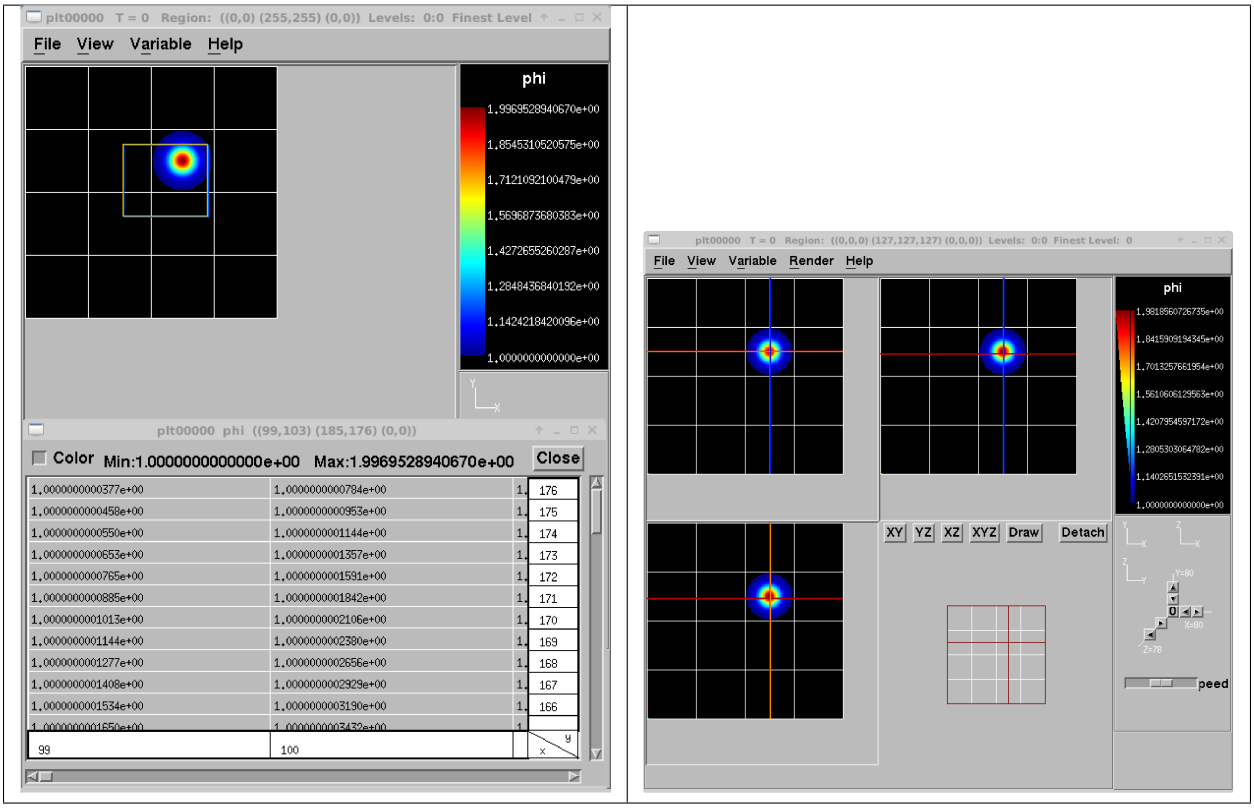
control options such as the initial field to display, the number format, widow size, etc. If there are multiple instances of the same option, the last option takes precedence.

3. Generally the plotfiles have the form pltXXXXX (the plt prefix can be changed), where XXXXX is a number corresponding to the timestep the file was output. `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 14.1). 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 and center click to get line-out plots. In 3D you can right and center click to change the planes, and the hold shift+(right or center) click to get line-out plots.

   We have created a number of routines to convert AMReX plotfile data other formats (such as matlab), but in order to properly interpret the hierarchical AMR data, each tends to have its own idiosyncrasies. If you would like to display the data in another format, please contact Marc Day (MSDay@lbl.gov) and we will point you to whatever we have that can help.

Table 14.1: 2D and 3D images generated using Amrvis



14.1.1 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.

14.2 VisIt

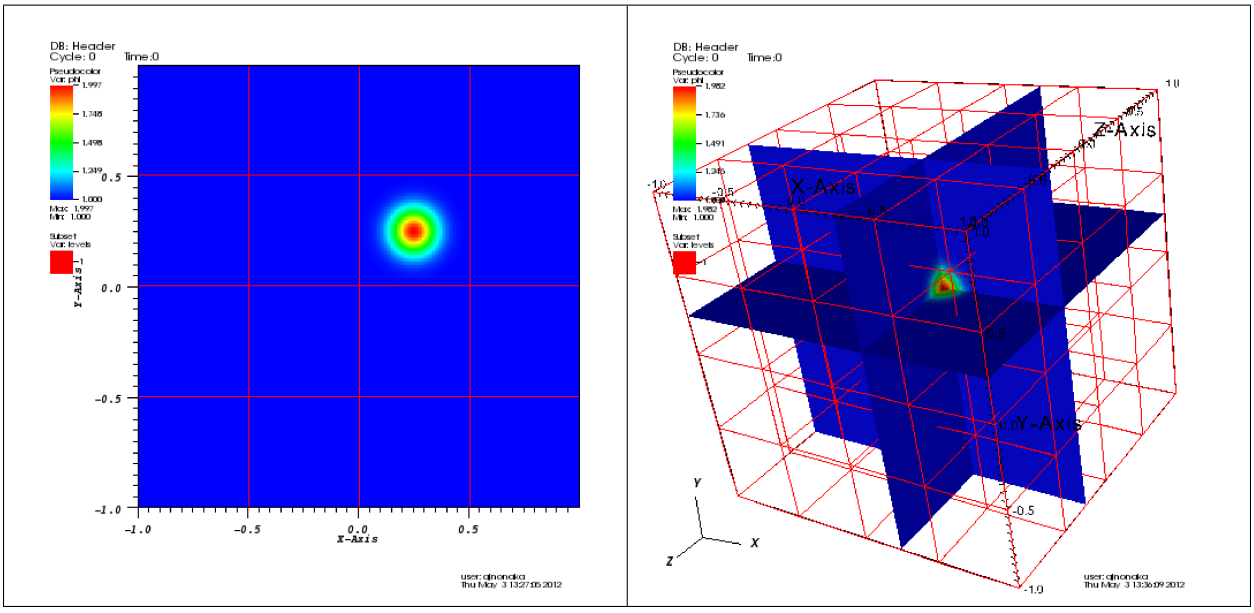
AMReX data can also be visualized by VisIt, an open source visualization and analysis software. To follow along with this example, first build and run the first heat equation tutorial code (see the section on *Example: Heat Equation Solver*).

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 the 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 " → "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 14.2.

Table 14.2: : 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 14.2.

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/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.

14.3 ParaView

The open source visualization package ParaView v5.5 can be used to view 3D plotfiles, and particle data. Download the package at https://www.paraview.org/.

To open a single plotfile (for example, you could run the `HeatEquation_EX1_C` in 3D):

1. Run ParaView v5.5, then select "File" → "Open".
2. Navigate **into** the plotfile directory, and **manually** type in "Header". ParaView will ask you about the file type – choose "Boxlib 3D Files"
3. Under the "Cell Arrays" field, select a variable (e.g., "phi") and click "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. 14.1.

To visualize particle data within plofile directories (for example, you could run the `ShortRangeParticles` example):

1. Run ParaView v5.5, 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. 14.2, and click OK.

1. 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.
2. Click "Apply" and under "Representation" select "Point Gaussian".
3. 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. 14.3.

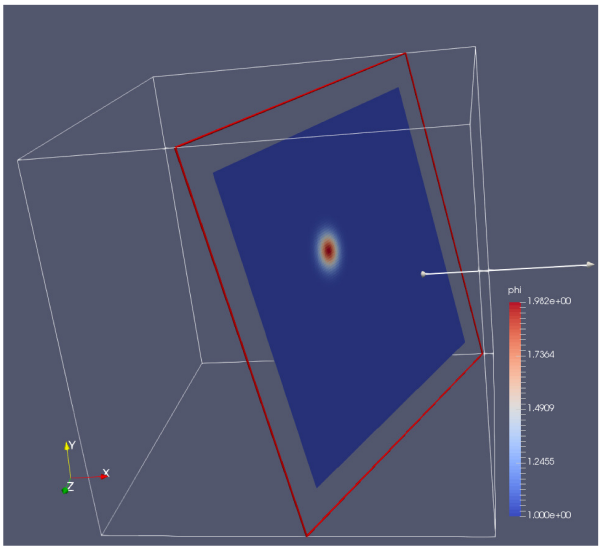


Fig. 14.1: : Plotfile image generated with ParaView

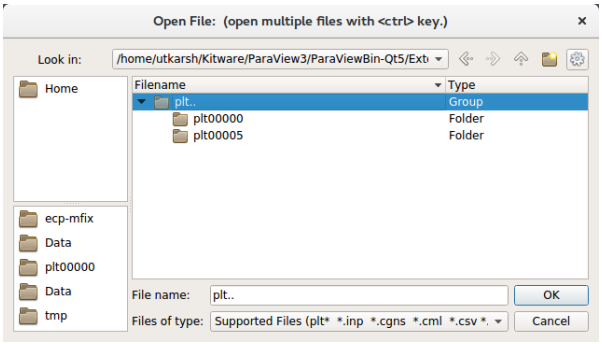


Fig. 14.2: : File dialog in ParaView showing a group of plotfile directories selected

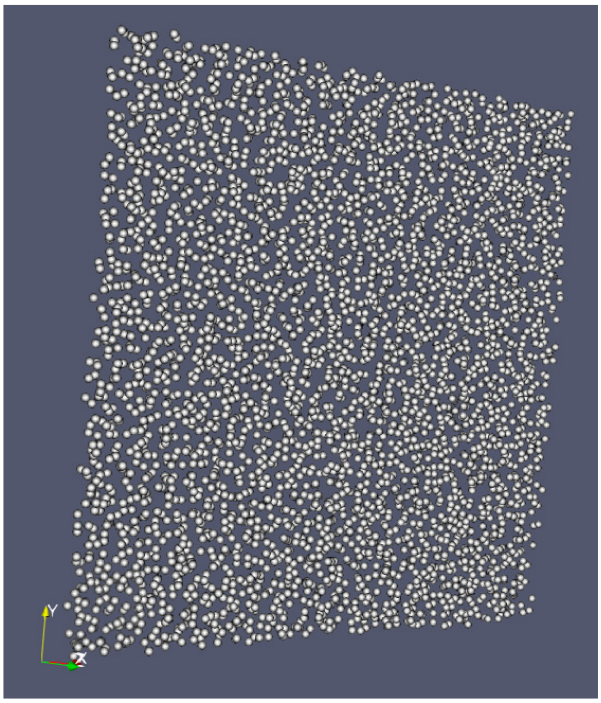


Fig. 14.3: : Particle image generated with ParaView

14.4 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.

14.4.1 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]
```

(continues on next page)

(continued from previous page)

```
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_left_edge          = [ 0.␣
↪ 0.  0.]
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. 14.4. 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
Camera:
```

(continues on next page)

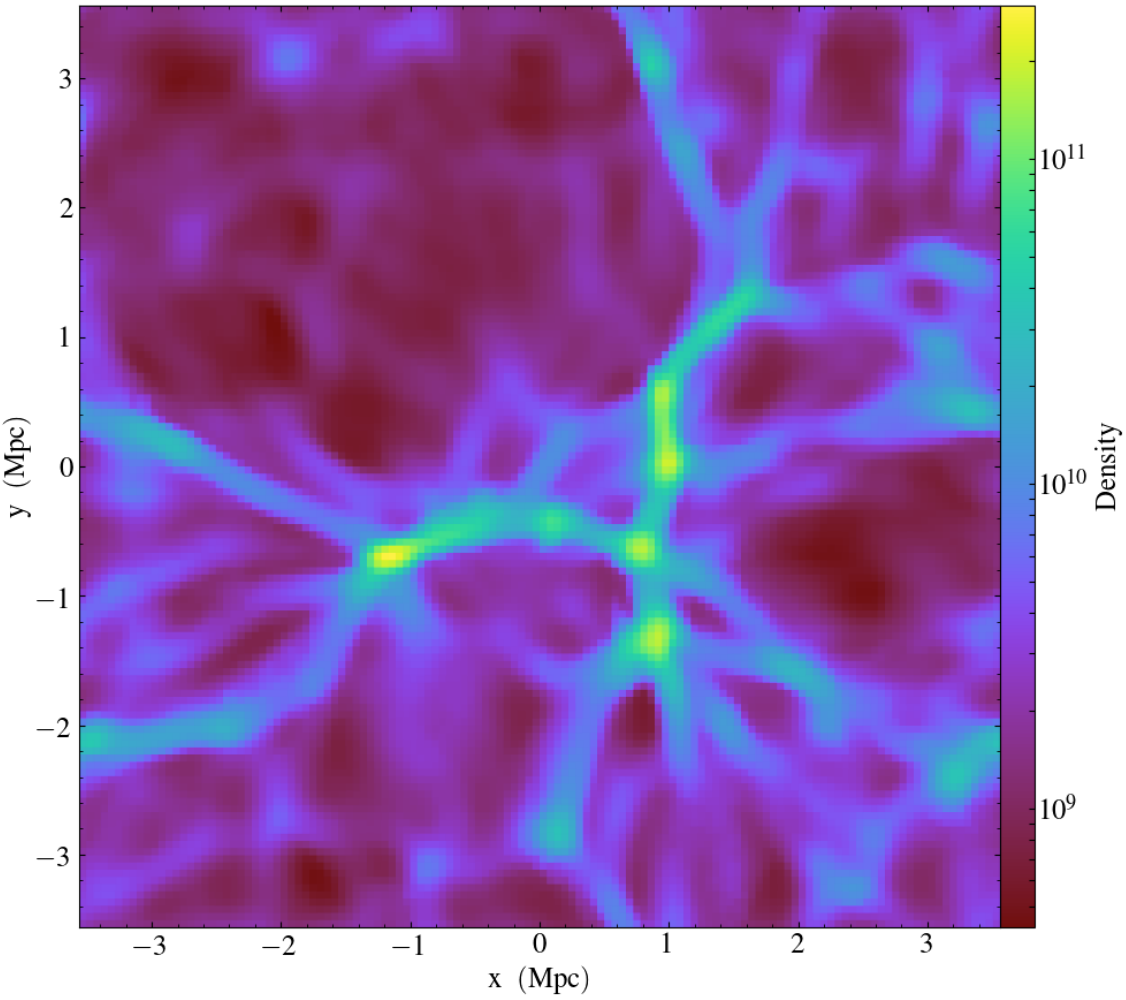


Fig. 14.4: : Slice plot of 128^3 Nyx simulation using yt.

(continued from previous page)

```
    <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. 14.5.

14.4.2 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/.

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

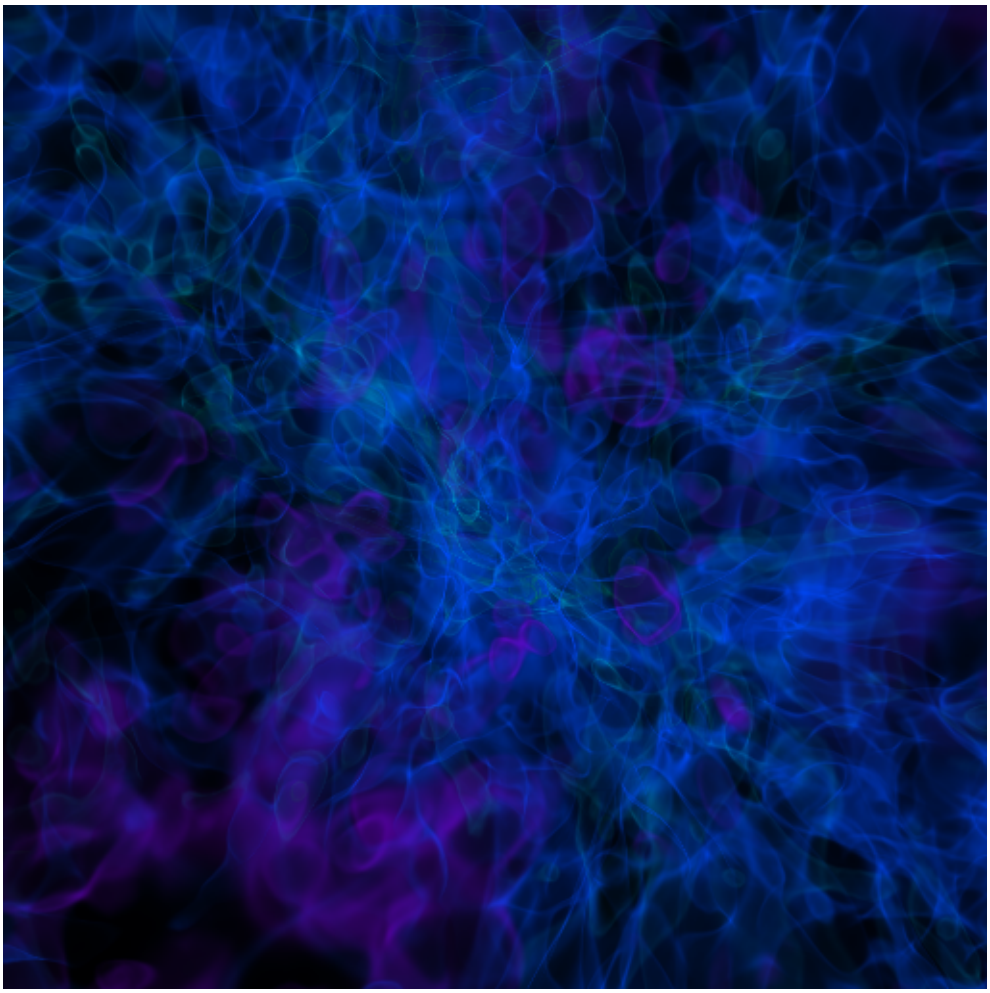


Fig. 14.5: Volume rendering of 128^3 Nyx simulation using yt. This corresponds to the same plot file used to generate the slice plot in Fig. 14.4.

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()

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'
```

(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)

```
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]
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.]
```

(continues on next page)

(continued from previous page)

```
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.

14.5 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.

14.5.1 System Architecture

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. 14.6. 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.

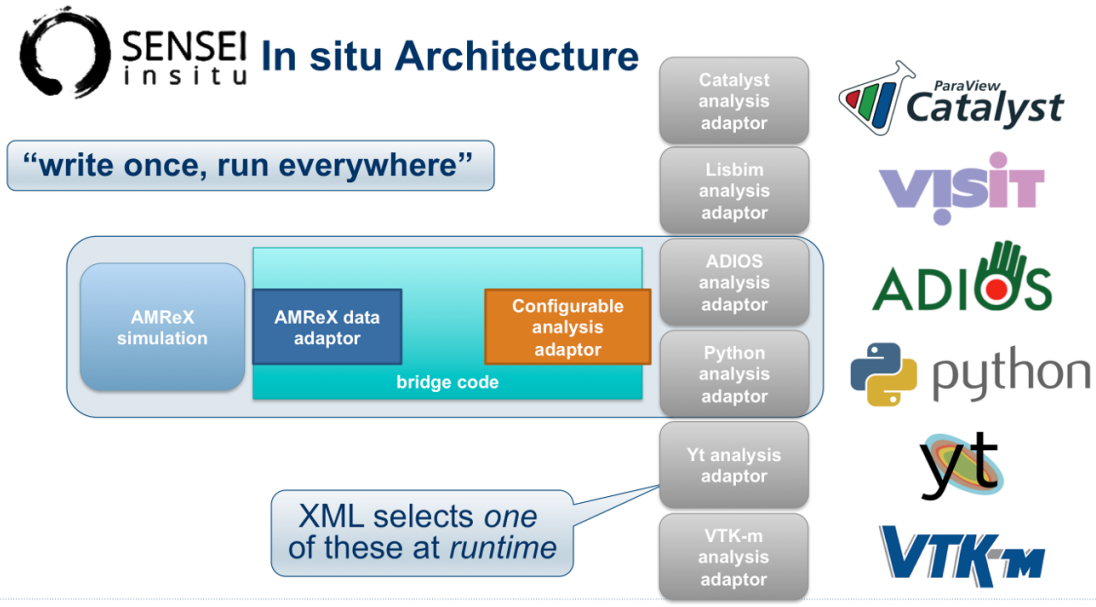


Fig. 14.6: SENSEI's in situ architecture enables use of a diverse of back ends which can be selected at run time via an XML configuration file

14.5.2 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`.

14.5.3 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
```

14.5.4 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 -DENABLE_SENSEI=ON -DSENSEI_DIR=<path to install>/lib/cmake ..
```

When CMake generates the make files proceed as usual.

```
make -j4 -f GNUmakefile
```

14.5.5 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.

14.5.6 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. 14.7, 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.

The same run and visualization was repeated using ParaView Catalyst, shown in figure Fig. 14.8, 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.

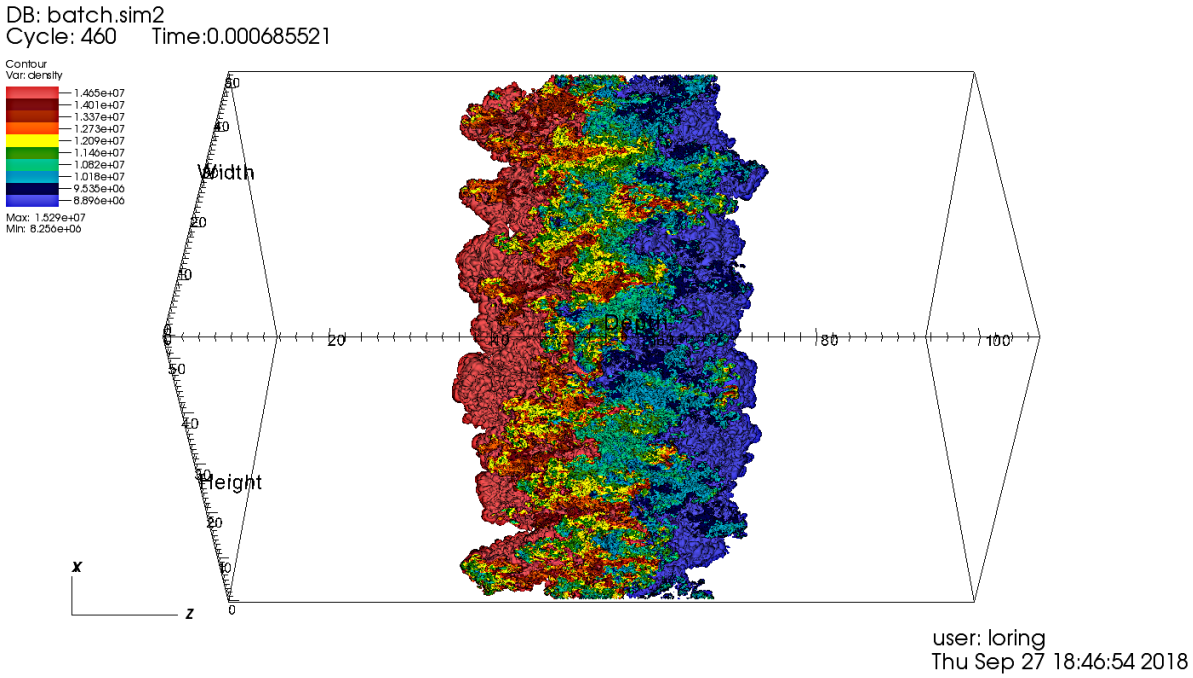


Fig. 14.7: SENSEI-Libsim in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.

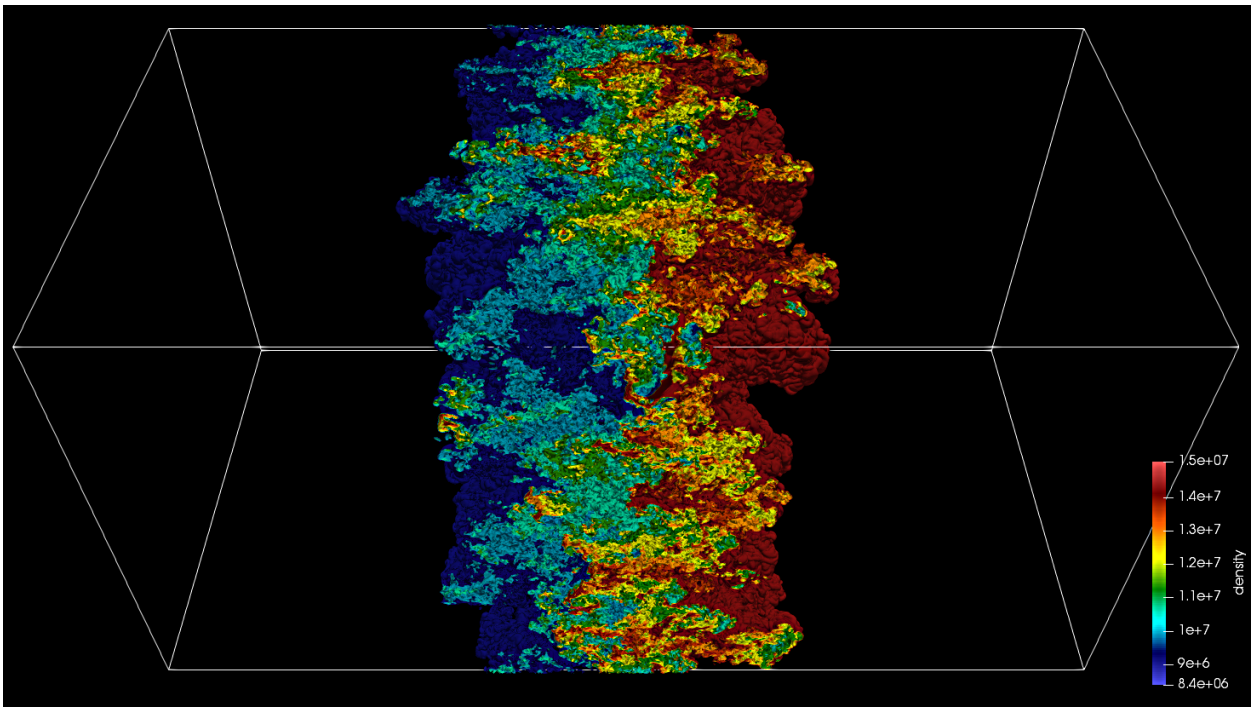


Fig. 14.8: SENSEI-Catalyst in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.

14.5.7 Obtaining SENSEI

SENSEI is hosted on Kitware's Gitlab site at https://gitlab.kitware.com/sensei/sensei It's best to checkout the latest release rather than working on the master branch.

To ease the burden of wrangling back end installs SENSEI provides two platforms with all dependencies pre-installed, a VirtualBox VM, and a NERSC Cori deployment. New users are encouraged to experiment with one of these.

SENSEI VM

The SENSEI VM comes with all of SENSEI's dependencies and the major back ends such as VisIt and ParaView installed. The VM is the easiest way to test things out. It also can be used to see how installs were done and the environment configured.

NERSC Cori

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

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
cd amrex/Tutorials/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
cd $SCRATCH/amrex/Tutorials/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
cd amrex/Tutorials/Amr/Advection_AmrLevel/Exec/SingleVortex
module use /usr/common/software/sensei/modulefiles
module load sensei/2.1.0-libsim-shared
```

(continues on next page)

(continued from previous page)

```
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
./main2d.gnu.haswell.MPI.ex inputs
```

CHAPTER

FIFTEEN

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

15.1 Types of Profiling

Currently you have two options for AMReX-specific profiling: *Tiny Profiling* and *Full Profiling*.

15.1.1 Tiny Profiling

To enable "Tiny Profiling", if using GNU Make then set

```
TINY_PROFILE = TRUE
PROFILE      = FALSE
```

in your GNUMakefile. If using cmake then set the following cmake flags

```
AMREX_ENABLE_TINY_PROFILE = ON
AMREX_ENABLE_BASE_PROFILE = OFF
```

Note that if you set `PROFILE = TRUE` (or `AMREX_ENABLE_BASE_PROFILE = ON`) then this will override the `TINY_PROFILE` flag and tiny profiling will be disabled.

At the end of the 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 *Sample Output From Tiny Profile* for sample output.

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 times 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.

15.1.2 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). In addition, an exclusive-only set of function timings will be written to stdout.

Trace Profiling

If, in addition to PROFILE = TRUE, you set TRACE_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. Part of the trace profiling is the ability to set regions in the code which can be analyzed for profiling information independently from other regions.

Communication Profiling

If, in addition to PROFILE = TRUE, you set COMM_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.

15.2 Instrumenting C++ Code

You must at least instrument main(), i.e

```
int main(...)
{
  amrex::Initialize(argc,argv);
  BL_PROFILE_VAR("main()",pmain);

  ...

  BL_PROFILE_VAR_STOP(pmain);
  amrex::Finalize();
}
```

You can then instrument any of your functions

```
void YourClass::YourFunction()
{
  BL_PROFILE_VAR("YourClass::YourFunction()",object_name);  // this name can be any␣
↪string

  // your function code
}
```

Note that you do not need to put BL_PROFILE_VAR_STOP because the profiler will go out of scope at the end of the function.

For other timers within an already instrumented function, add:

```
BL_PROFILE_VAR("Flaten::FORT_FLATENX()", anyname);  // add this before
  FORT_FLATENX(arg1, arg2);
BL_PROFILE_VAR_STOP(anyname);   // add this after, using the same name
```

if you want to use the same name within the same scope, you can use:

```
BL_PROFILE_VAR("MyFuncs()", myfuncs);  // the first one
  MyFunc_0(arg);
BL_PROFILE_VAR_STOP(myfuncs);
...
BL_PROFILE_VAR_START(myfuncs);
  MyFunc_1(arg);
BL_PROFILE_VAR_STOP(myfuncs);
```

or create a profiling variable without starting, then start/stop:

```
BL_PROFILE_VAR_NS("MyFuncs()", myfuncs);  // dont start the timer
...
BL_PROFILE_VAR_START(myfuncs);
  MyFunc_0(arg);
BL_PROFILE_VAR_STOP(myfuncs);
...
BL_PROFILE_VAR_START(myfuncs);
  MyFunc_1(arg);
BL_PROFILE_VAR_STOP(myfuncs);
```

15.3 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 and the profiling output will warn of any `bl_proffortfuncstart` calls that were not stopped with `bl_proffortfuncstop` calls (in debug mode only). You will need to add `bl_proffortfuncstop` before any returns and at the end of the function or at the point in the function you want to stop profiling.

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().

Be aware: Fortran functions cannot be profiled when using the Tiny Profiler. You will need to turn on the full profiler to recieve the results from fortran instrumentation.

15.4 Sample Output From Tiny Profile

Sample output using TINY_PROFILE = TRUE can look like the following:

```
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%
```

15.5 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

SIXTEEN

EXTERNAL PROFILING TOOLS

16.1 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.

16.1.1 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:

```
Table 1:  Profile by Function

  Samp% |    Samp |  Imb. |  Imb. | Group
        |         |  Samp | Samp% |  Function
        |         |       |       |   PE=HIDE
```

(continues on next page)

[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

(continued from previous page)

```
 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
||===========================================================================
```

16.2 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].

[4] http://ipm-hpc.sourceforge.net/userguide.html
[5] https://www.nersc.gov/users/software/performance-and-debugging-tools/ipm/

16.2.1 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.

16.2.2 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.

16.2.3 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.25600
# 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
# 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
```

(continues on next page)

(continued from previous page)

```
# 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.

16.2.4 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
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.

16.2.5 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
- Memmory Usage
- Executable Info
- Host List
- Environment
- Developer Info

powered by IPM

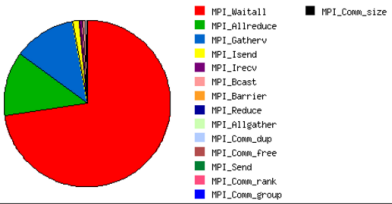
command: /global/cscratch1/sd/cchan2/projects/lbl/BoxLib/Tests/LinearSolvers/C_CellMG//main3d.intel.MPI.OMP.ex.ipm inputs.3d.15456			
codename:	unknown	state:	unknown
username:	cchan2	group:	
host:	nid04720 (x86_64_Linux)	mpi_tasks:	4 on 1 hosts
start:	08/22/17/12:04:29	wallclock:	5.20833e+01 sec
stop:	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. 16.1: Sample performance summary generated by IPM

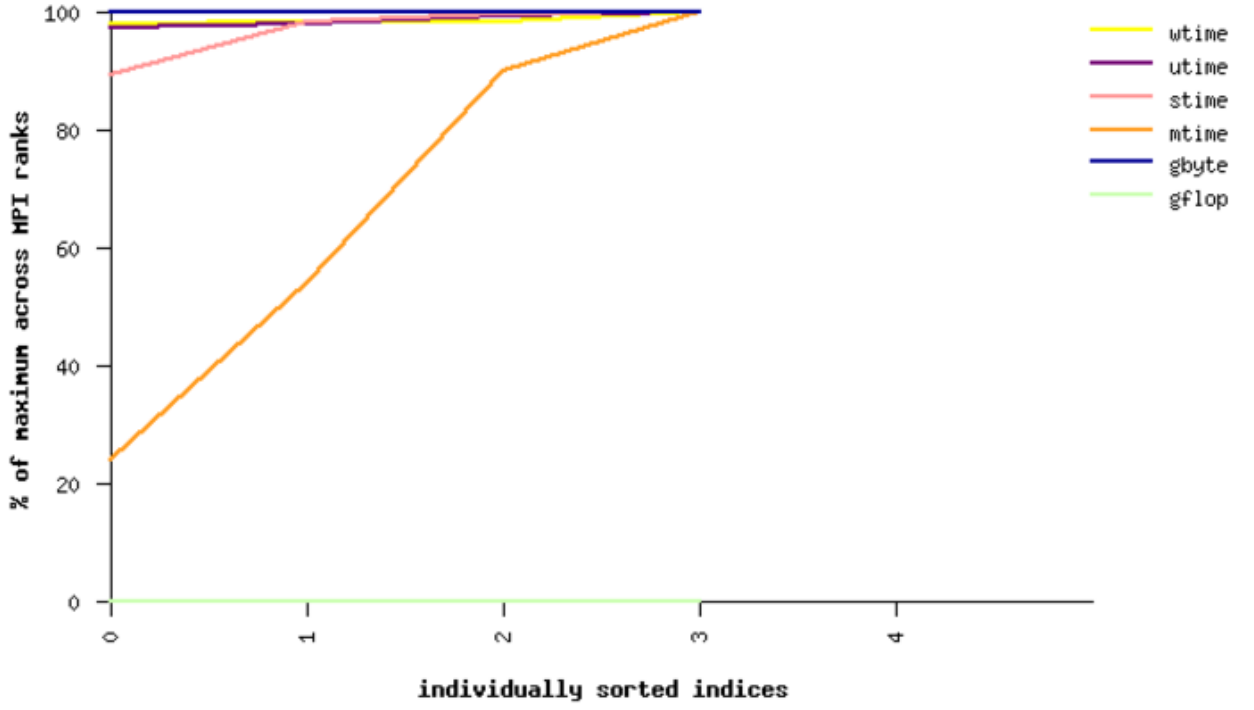
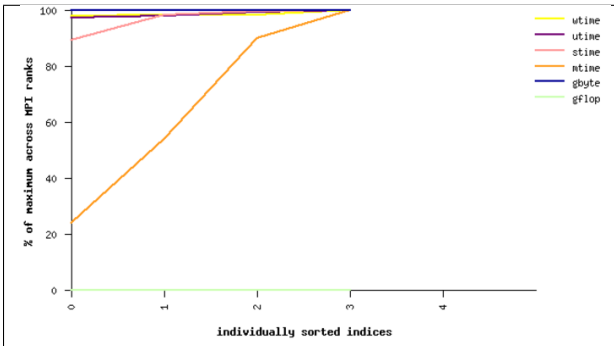
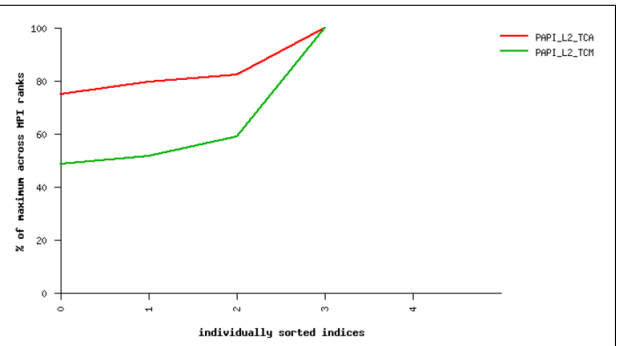


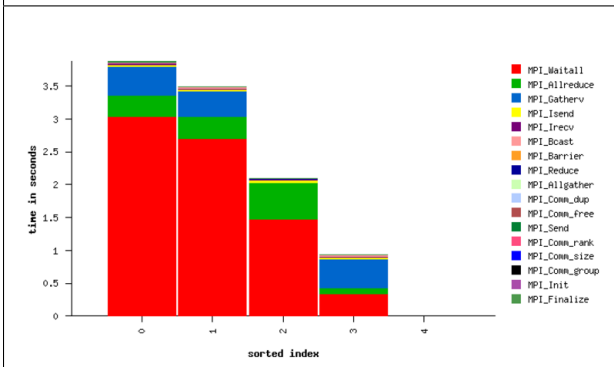
Table 16.1: Example of performance graphs generated by IPM



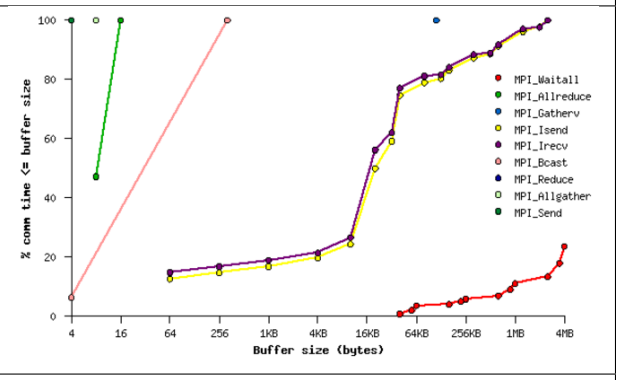
Timings



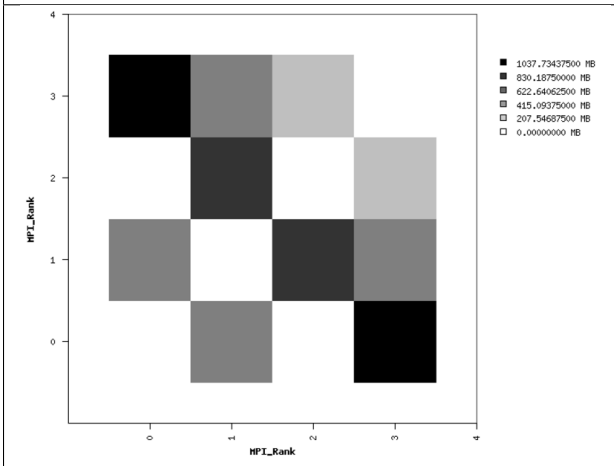
PAPI Counters



MPI Time by Function



MPI Time by Message Size



(left) Point-to-Point Communication Volume

CHAPTER

SEVENTEEN

EXTERNAL FRAMEWORKS

17.1 CVODE

AMReX supports local ODE integration using the CVODE solver,[1] which is part of the SUNDIALS framework.[2] CVODE contains solvers for stiff and non-stiff ODEs, and as such is well suited for solving e.g., the complex chemistry networks in combustion simulations, or the nuclear reaction networks in astrophysical simulations.

Most of CVODE is written in C, but many functions also come with two distinct Fortran interfaces. One interface is FCVODE, which is bundled with the stable release of CVODE. Its usage is described in the CVODE documentation.[3] However, the use of FCVODE is discouraged in AMReX due to its incompatibility with being used inside OpenMP parallel regions (which is the primary use case in AMReX applications).

The alternative, and recommended, Fortran interface to uses the `iso_c_binding` feature of the Fortran 2003 standard to implement a direct interface to the C functions in CVODE. When compiling CVODE, one need not build the CVODE library with the FCVODE interface enabled at all. Rather, the Fortran 2003 interface to CVODE is provided within AMReX itself. The CVODE tutorials provided in AMReX use this new interface.

17.1.1 Compiling AMReX with CVODE (Cray or Sundials version 2.7)

The following steps describe how to compile an AMReX application with CVODE support. On Cray systems (e.g., Cori or Edison at NERSC), Cray provides a system module called `cray-tpsl` ("Cray Third-Party Scientific Libraries") which contains the latest version of the SUNDIALS solver suite (including CVODE). Simply type `module load cray-tpsl` and set `USE_CVODE=TRUE` in the `GNUmakefile`, and AMReX will automatically link the SUNDIALS libraries.

On systems which are not Cray:

1. Obtain the CVODE source code, which is hosted here: https://computation.llnl.gov/projects/sundials/sundials-software. One can download either the complete SUNDIALS package, or just the CVODE components.
2. Unpack the CVODE / SUNDIALS tarball, and create a new "build" directory (it can be anywhere).
3. Navigate to the new, empty build directory, and type

```
cmake \
  -DCMAKE_INSTALL_PREFIX:PATH=/path/to/install/dir \
  /path/to/cvode/or/sundials/top/level/source/dir
```

[1] https://computation.llnl.gov/projects/sundials/cvode
[2] https://computation.llnl.gov/projects/sundials
[3] https://computation.llnl.gov/sites/default/files/public/cv_guide.pdf

The CMAKE_INSTALL_DIR option tells CMake where to install the libraries. Note that CMake will attempt to deduce the compilers automatically, but respects certain environment variables if they are defined, such as CC (for the C compiler), CXX (for the C++ compiler), and FC (for the Fortran compiler). So one may modify the above CMake invocation to be something like the following:

```
CC=/path/to/gcc \
CXX=/path/to/g++ \
FC=/path/to/gfortran \
  cmake \
  -DCMAKE_INSTALL_PREFIX:PATH=/path/to/install/dir \
  /path/to/cvode/or/sundials/top/level/source/dir
```

One can supply additional flags to CMake or to the compiler to customize the compilation process. Flags of interest may include CMAKE_C_FLAGS, which add the specified flags to the compile statement, e.g., -DCMAKE_C_FLAGS="-h list=a" will append the -h list=a flag to the cc statement when compiling the source code. Here one may wish to add something like "-O2 -g" to provide an optimized library that still contains debugging symbols; if one neglects debugging symbols in the CVODE library, and if a code that uses CVODE encounters a segmentation fault in the solve, then the backtrace has no information about where in the solver the error occurred. Also, if one wishes to compile only the solver library itself and not the examples that come with the source (compiling the examples is enabled by default), one can add "-DEXAMPLES_ENABLE=OFF". Users should be aware that the CVODE examples are linked dynamically, so when compiling the solver library on Cray system using the Cray compiler wrappers cc, CC, and ftn, one should explicitly disable compiling the examples via the "-DEXAMPLES_ENABLE=OFF" flag.

4. In the GNUmakefile for the application which uses the Fortran 2003 interface to , add USE_CVODE = TRUE, which will compile the Fortran 2003 interfaces and link the libraries. Note that one must define the CVODE_LIB_DIR environment variable to point to the location where the libraries are installed.

17.1.2 CVODE Tutorials

AMReX provides two CVODE tutorials in the amrex/Tutorials/CVODE directory, called EX1 and EX2. See the Tutorials CVODE documentation for more detail.

17.1.3 Compiling AMReX with Sundials version 3.X or later

The following steps describe how to compile an AMReX application with SUNDIALS_3.X support. On Cray systems (e.g., Cori or Edison at NERSC), Cray provides a system module called cray-tpsl ("Cray Third-Party Scientific Libraries") which as of this writing contains the 2.7 version of the SUNDIALS solver suite (including CVODE).

In order to use the Sundials 3.X version:

1. Obtain the CVODE source code, which is hosted here: https://computation.llnl.gov/projects/sundials/sundials-software. One can download either the complete SUNDIALS package, or just the CVODE components.
2. Unpack the CVODE / SUNDIALS tarball, and create a new "build" directory (it can be anywhere).
3. Navigate to the new, empty build directory, and type

```
cmake \
  -DCMAKE_INSTALL_PREFIX:PATH=/path/to/install/dir \
  /path/to/cvode/or/sundials/top/level/source/dir
```

The CMAKE_INSTALL_DIR option tells CMake where to install the libraries. Note that CMake will attempt to deduce the compilers automatically, but respects certain environment variables if they are defined, such as CC

(for the C compiler), CXX (for the C++ compiler), and FC (for the Fortran compiler). So one may modify the above CMake invocation to be something like the following:

```
CC=/path/to/gcc \
CXX=/path/to/g++ \
FC=/path/to/gfortran \
  cmake \
  -DCMAKE_INSTALL_PREFIX:PATH=/path/to/install/dir \
  /path/to/cvode/or/sundials/top/level/source/dir
```

One can supply additional flags to CMake or to the compiler to customize the compilation process. Flags of interest may include CMAKE_C_FLAGS, which add the specified flags to the compile statement, e.g., -DCMAKE_C_FLAGS="-h list=a" will append the -h list=a flag to the cc statement when compiling the source code. Here one may wish to add something like "-O2 -g" to provide an optimized library that still contains debugging symbols; if one neglects debugging symbols in the CVODE library, and if a code that uses CVODE encounters a segmentation fault in the solve, then the backtrace has no information about where in the solver the error occurred. Also, if one wishes to compile only the solver library itself and not the examples that come with the source (compiling the examples is enabled by default), one can add "-DEXAMPLES_ENABLE=OFF". Users should be aware that the CVODE examples are linked dynamically, so when compiling the solver library on Cray system using the Cray compiler wrappers cc, CC, and ftn, one should explicitly disable compiling the examples via the "-DEXAMPLES_ENABLE=OFF" flag.

4. In the GNUmakefile for the application which uses the Fortran 2003 interface to CVODE or ARKODE, add SUNDIALS_3x4x = TRUE, which will compile the Fortran 2003 interfaces and link the libraries. Note that one must define the CVODE_LIB_DIR environment variable to point to the location where the libraries are installed.
5. In the GNUmakefile for the application which uses the Fortran 2003 interface to ARKODE, also add USE_ARKODE_LIBS = TRUE. It is assumed that the CVODE_LIB_DIR environment variable points to the location where the ARKODE libraries are installed as well.
6. Fortran 2003 interfaces for the pgi compilers and for developmental versions of SUNDIALS are currently not supported.

17.1.4 SUNDIALS 3.X Tutorials

AMReX provides six tutorials in the amrex/Tutorials/CVODE/SUNDIALS3_finterface directory. EX1 is modeled after the CVODE Tutorial EX1 showing use with AMReX. The four EX_cv_* tutorials are based on examples provided with the interface, which are more closely modeled after CVODE examples. The EX_ark_analytic_fp tutorial is based on the EX_cv_analytic_fp tutorial, but uses ARKODE instead of CVODE.

AMReX provides three tutorials in the amrex/Tutorials/CVODE/SUNDIALS3_cppversion directory. These are versions of EX1 which operate on a packed version of the data. EX1_SERIAL_NVEC packs a box worth of equations into a serial NVector, uses CVODE to solve, and then unpacks the solution back into the box it came from. EX1_CUDA_NVEC uses the cuda NVector implementation instead. EX1_GPU_PRAGMA uses the cuda NVector, and the gpu pragma functionality.

17.2 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]

17.2.1 Pencil Redistribution

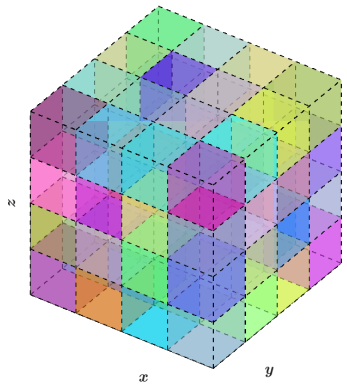
As input, SWFFT takes three-dimensional arrays of data distributed across block-structured grids, and redistributes the data into "pencil" grids in z, x, and then y, belonging to different MPI processes. After each pencil conversion, a 1D FFT is performed on the data along the pencil direction using 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[1][2]:

> [...] A rule of thumb is that [SWFFT] generally works when the number of vertexes 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`.

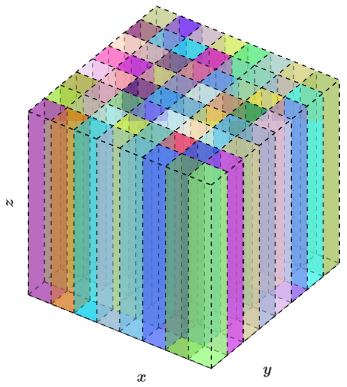
The relationship between the number of processes versus global grid dimensions is determined by how the total number of grids can be factored from a three dimensional grid structure (block structured grids) into a two dimensional structure (pencil arrays), as shown in the figures below.

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 17.1: SWFFT Redistribution from $4 \times 4 \times 4$ Box Array into Pencils



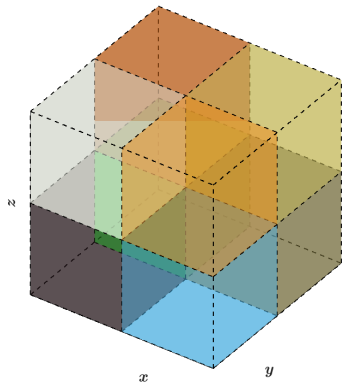
(a) Block structured grids: $N_x = 4, N_y = 4, N_z = 4$



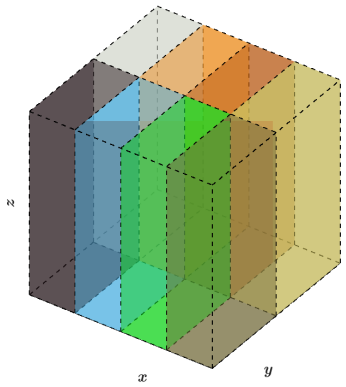
(b) Z-pencils: $N_x = 8, N_y = 8, N_z = 1$

[1] https://xgitlab.cels.anl.gov/hacc/SWFFT
[2] SWFFT source code directory in AMReX: amrex/Src/Extern/SWFFT
[3] http://www.fftw.org/

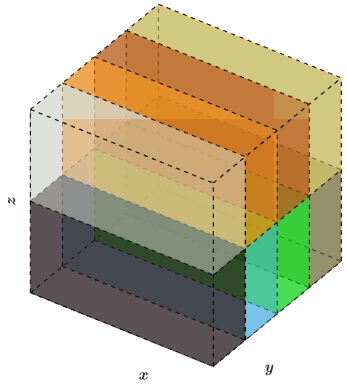
Table 17.2: SWFFT Redistribution from $2 \times 2 \times 2$ Box Array into Pencils



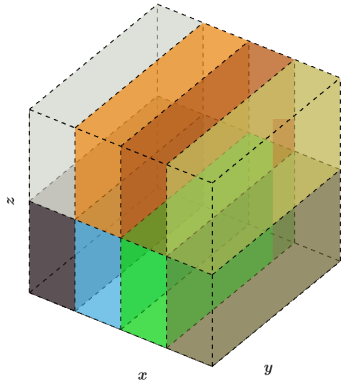
(a) Block structured grids: $N_x = 2, N_y = 2, N_z = 2$



(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.

17.2.2 Tutorials

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

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

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

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