AMReX

An adaptive mesh refinement software framework

User's Guide

AMReX Developers

July 7, 2017

Chapter Listing

Contents

List of Figures

List of Tables

Preface

Welcome to the AMReX User's Guide!

AMReX is a software framework library containing all the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications. AMReX is freely available at `https://github.com/AMReX-Codes/amrex`. The most current version of this User's Guide can be found in the AMReX git repository at `AMReX/Docs/AMReXUsersGuide`.

CHAPTER 1

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
- Parallelization via flat MPI, OpenMP, hybrid MPI/OpenMP, or MPI/MPI
- Parallel I/O
- Plotfile format supported by Amrvis, VisIt, ParaView, and yt.

Because AMReX's core is mainly written in C++, the basics of AMReX is first introduced in C++ and then their Fortran wrappers if available will be described as well.

Besides the User's Guide, there is document generated by Doxygen at `https://ccse.lbl.gov/pub/AMReX_Docs/`. Documentation on migration from BoxLib is available at `Docs/Migration`.

CHAPTER 2

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 AMReX 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 `main` function of every program should be calling `amrex::Initialize` and `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, the dimension matters 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 Chapter 3.

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
```

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 Chapter 3 on how to configure the build system.

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,

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

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,

```
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. Don't worry about the details of the code. You will be able to understand the code in this example after Chapter 4.

2.3.1 Building and Running the Code

To build a 2D executable, type `make DIM=2`. This will generate an executable named `main2d.gnu.ex`. To run it, type,

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

Note that the command takes a file `inputs_2d`. When the run finishes, you will have a number of plotfiles, `plt00000`, `plt01000`, etc. The calculation solves the heat equation in 2D on a 256×256 cells domain. It runs $10,000$ steps and makes a plotfile every $1,000$ steps. These are runtime parameters that can be adjusted in `inputs_2d`.

2.4 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. Refer to Chapter 9 for how to use each of these tools.

CHAPTER 3

Building AMReX

In this chapter, we discuss AMReX's build systems. There are three ways to use AMReX. The approach used by AMReX developers uses GNU Make. There is no installation step in this approach. Application codes adopt AMReX's build system and compile AMReX while compiling their own codes. This will be discussed in more detail in Section 3.1. The second approach is to build AMReX into a library and install it (Section 3.2). Then an application code uses its own build system and links AMReX as an external library. AMReX can also be built with Cmake (Section 3.3).

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 5.1 shows a list of important variables.

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`; in tcsh one can set `setenv AMREX_HOME /path/to/amrex`.

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

Table 3.1: Important make variables

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 AMReX 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 GNUmakefile. AMReXbased 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 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 by 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., 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 a amrex/Tools/GNUMake/Make.local file to override various variables. See amrex/Tools/GNUMake/Make.local.template for an example.

3.2 Building libamrex

If an application code already has its own elaborated build system and wants to use AMReX as an external library, this might be your choice. In this approach, one runs ./configure, followed by make and make install. 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 Section 3.1 is recommended if any fine tuning is needed.

3.3 Building with CMake

An alternative to the approach described in Section 3.2 is to install AMReX as an external library by using the CMake build system. A CMake build is a two-steps process. First cmake is invoked to create configuration files and makefiles in a chosen directory (builddir). This is roughly equivalent to running ./configure (see Section 3.2). Next, the actual build and installation are performed by issuing make install from within builddir. This will install the library files in a chosen installation directory (installdir). The CMake build process is summarized as follow:

```
mkdir /path/to/builddir
cd    /path/to/builddir
cmake [options] -DCMAKE_INSTALL_PREFIX:PATH=<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 Subsection 3.3.1. Although the AMReX source could be used as build directory, we advise against doing so. After the installation is complete, `builddir` can be removed.

3.3.1 Customization options

AMReX 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=<path/to/installdir>  /path/to/amrex
```

The list of available option is reported in Table 3.2.

Option name	Description	Default Value	Possible values
CMAKE_BUILD_TYPE	Type of AMReX build	Debug	Debug,Release
BL_SPACEDIM	Dimension of AMReX build	3	2,3
ENABLE_DP	Enable double precision	1	0,1
ENABLE_MPI	Enable build with MPI	1	0,1
ENABLE_OMP	Enable build with OpenMP	0	0,1
ENABLE_PARTICLES	Include particle classes in build	0	0,1
ENABLE_DP_PARTICLES	Enable double precision for particle classes	1	0,1
ENABLE_PROFILING	Include profiling info in build	0	0,1
ENABLE_TINY_PROFILING	Include tiny-profiling info in build	0	0,1
ENABLE_BACKTRACE	Include backtrace info in build	1	0,1
AMREX_FFLAGS_OVERRIDES	User-defined Fortran flags	empty	user-defined
AMREX_CXXFLAGS_OVERRIDES	User-defined C++ flags	empty	user-defined

Table 3.2: Variables for the customization of AMReX build with CMake

CHAPTER 4

The 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 tutorials in `amrex/Tutorials/Basic/` 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 Chapter 3, 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 Array

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

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 4.13). 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
Array<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 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();
```

4.6 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() 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, texts 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

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

Array<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
```

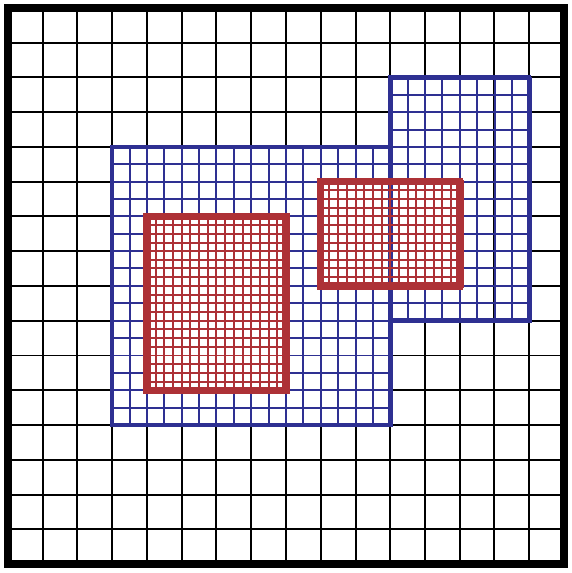

Figure 4.1: Example of AMR grids. There are three levels in total. There are 1, 2 and 2 `Box`es on levels 0, 1, and 2, respectively.

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`.

4.7 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. Figure 4.1 shows an example of AMR grids. There are three total levels in the example. In 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. Note that there is no direct parent-child connection. In this chapter, we will focus on single levels.

4.8 Box, IntVect and IndexType

Box in AMReX_Box.H is the data structure for representing a rectangular domain in indexing space. For example, in Figure 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). There are no floating-point data in the object.

4.8.1 IntVect

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

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

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

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

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

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

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

```
IntVect iv(AMREX_D_DECL(127,127,127));
IntVect coarsening_ratio(AMREX_D_DECL(2,2,2));
iv.coarsen(2);                  // Coarsen each component by 2
iv.coarsen(coarsening_ratio);  // Component-wise coarsening
const auto& iv2 = amrex::coarsen(iv, 2); // Return an IntVect w/o modifying iv
IntVect iv3 = amrex::coarsen(iv, coarsening_return); // 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";
```

4.8.2 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 $i + 1/2$.

4.8.3 Box

A `Box` is an abstraction for defining discrete regions of `AMREX_SPACEDIM`-dimensional indexing space. `Box`es have an `IndexType` and two `IntVect`s 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`

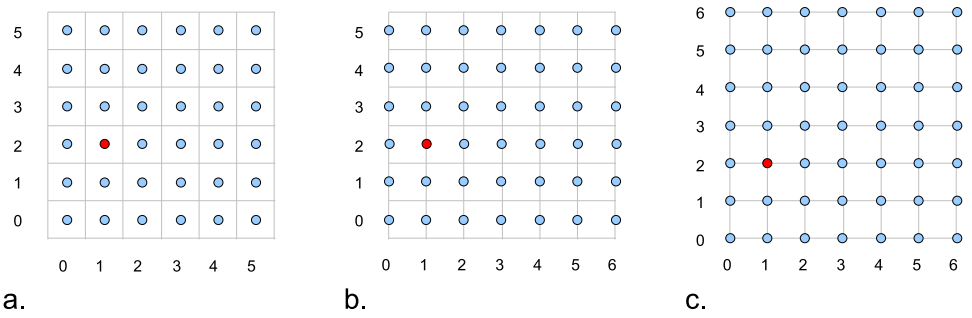

Figure 4.2: Some of the different index types in two dimensions: (a) cell-centered, (b) x-face-centered (i.e., nodal in x-direction only), and (c) corner/nodal, i.e., nodal in all dimensions.

and 1 denote cell and node, respectively. For each tuple like `(64,64,64)`, the 3 numbers are for 3 directions. The two `Box`es 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 $i - 1/2$. Figure 4.2 shows a 2D example of various index types.

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
```

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

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

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

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

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

Note that refinement and coarsening behaviors depend on the indexing type. One should think the refinement and coarsening in AMR context that refined or coarsened `Box` still covers the same physical domain. `Box uncoarsenable` in the example above is considered uncoarsenable because its coarsened version does not cover the same physical domain in the AMR context.

`Box`es can grow and they can grow in all directions or just one direction. There are a number of `grow` functions. Some are member functions of the `Box` class and others are non-member functions in the `amrex` namespace.

`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 `Box`es 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 `Box`es 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!
```

4.9 RealBox and Geometry

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

`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). `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
std::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?
```

4.10 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 (Section 4.11). Each process can then allocate memory for the floating point data on the Boxes it owns (Sections 4.13 & 4.12).

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.

4.11 DistributionMapping

DistributionMapping is a class in AMReX_DistributionMapping.H 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. A way to construct a DistributionMapping object given a BoxArray is as follows.

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

Oftentimes what one needs is 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 ParmParse parameter DistributionMapping.strategy.

KNAPSACK is a common choice that is optimized for load balance. One can also explicitly construct a distribution. DistributionMapping class allows the user to have complete control by passing an array of integers.

```
DistributionMapping dm;   // empty object
Array<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.
```

4.12 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 due to rectangular domain, 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 dimensional 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 Section 4.15). BaseFab has several functions that set the array data to a constant value (e.g., 0). 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 not supported, one can write Fortran or C functions for those (Section 4.15). 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 reason. However, it can be handy for debugging.

BaseFab and its derived classes are containers for data on Box. We recall that Box has types (Section 4.8). 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 for performance reason. However, it 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 the 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. It is however 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.

4.13 FabArray, MultiFab and iMultiFab

`FabArray<FAB>` is a class template in `AMReX_FabArray.H` for a collection of `FAB`s on the same AMR level associated with a `BoxArray` (Section 4.10). The template parameter `FAB` is usually `BaseFab<T>` or its derived classes (e.g., `FArrayBox`). However, it can also be used to hold other data structures. To construct a `FabArray`, a `BoxArray` must be provided because it is intended to hold *grid* data defined on a union of rectangular regions embedded in a uniform index space. For example, an `FabArray` object can be used to hold data for one level of the example grids of Figure 4.1.

`FabArray` is a parallel data structure that the data (i.e., `FAB`) are distributed among parallel processes. On each process, the `FabArray` contains only the `FAB` objects owned by this 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` (Section 4.11) that specifies which process owns which `Box`. For level 2 (*red*) in Figure 4.1, there are two `Box`es. Suppose there are two parallel processes, and we use a `DistributionMapping` that assigns one `Box` to each process. For `FabArray` on each process, it is built on a `BoxArray` with 2 `Box`es, but contains only one `FAB`.

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 `FabArray`s. 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, mf, ncomp, ngrow);
```

Here we define a MultiFab with 4 components and 1 ghost cell. A MultiFab contains a number of FArrayBoxes (Section 4.12) 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{(8,8,8) (15,15,15)}, the one used for constructing FArrayBox will be Box{(7,7,7) (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 alone does not have the concept of ghost cell, whereas ghost cell is a key concept of MultiFab 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, mf, 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 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 MultiFabs 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      sc   : 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 Src/Base/AMReX_MultiFab.H and Src/Base/AMReX_FabArray.H for more details. It should be noted again it is a runtime error if the two MultiFabs passed to functions like MultiFab::Copy are not built with the *same* BoxArray (including index type) and DistributionMapping.

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

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

```
mf.FillBoundary(2, 3);         // Fill 3 components starting from component 2
mf.FillBoundary(geom.periodicity(), 2, 3);
```

Note that `FillBoundary` does not modify any valid cells. Also note that `MultiFab` itself does not have the concept of periodic boundary, but `Geometry` has, and we can provide that information so that periodic boundaries can be filled as well. You might have noticed that a ghost cell could overlap with multiple valid cells from different `FArrayBox`es 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.

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 `MultiFab`s, 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. Same as `FillBoundary`, if a destination cell has multiple cells as source, it is unspecified that which source cell is used. 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 `MultiFab`s have the same number of components). Similar to `FillBoundary`, a destination cell may have multiple sources and which source is used is unspecified.

4.14 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`.

4.14.1 MFIter without Tiling

In Section 4.13, we have shown some of the arithmetic functionalities of `MultiFab`, such as adding two `MultiFab`s 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 `FArrayBox`es in `MultiFab`s. 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 a 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 assume the spatial dimension is 3. In 2D, the function interface is different. In Section 4.15, 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 component, we can call `int nc = mf.nComp()` to get the number of components and pass it 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
```

4.14.2 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
      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 Section 4.14.1 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 Section 4.14.1 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()
```

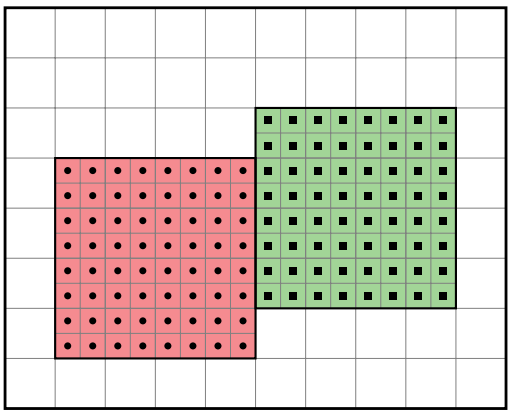

Figure 4.3: Example of cell-centered valid boxes. There are two valid boxes in this example. Each has 8^2 cells.

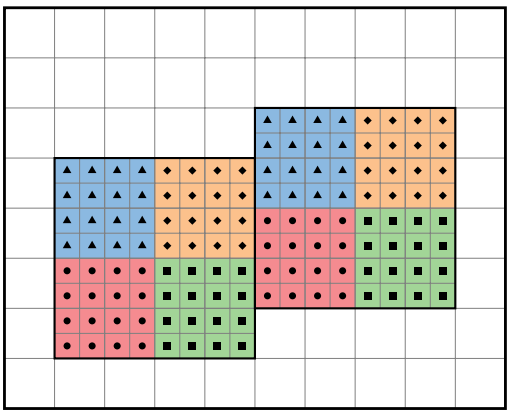

Figure 4.4: Example of cell-centered tile boxes. Each grid is *logically* broken into 4 tiles, and each tile has 4^2 cells. There are 8 tiles in total.

```
      const Box& box = mfi.tilebox();

      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();

      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.

Figures 4.3 & 4.4 show an example of the difference between `validbox` and `tilebox`. In this example, there are two grids of cell-centered index type. Function `validbox` always returns a `Box` for the valid region of an `FArrayBox` no matter whether or not tiling is enabled, whereas 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.

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 disable 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` parameter `fabarray.mfiter_tile_size`.

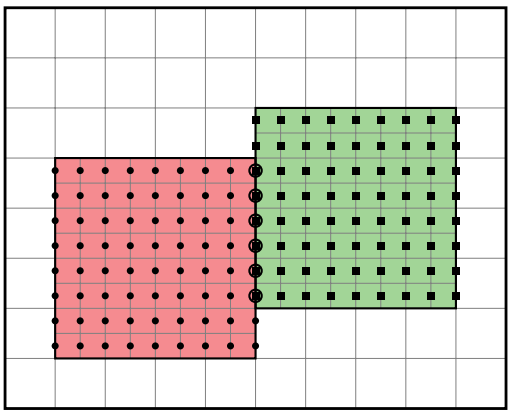

Figure 4.5: Example of face valid boxes. There are two valid boxes in this example. Each has 9×8 points. Note that points in one `Box` may overlap with points in the other `Box`. However, the memory locations for storing floating point data of those points do not overlap, because they belong to separate `FArrayBox`es.

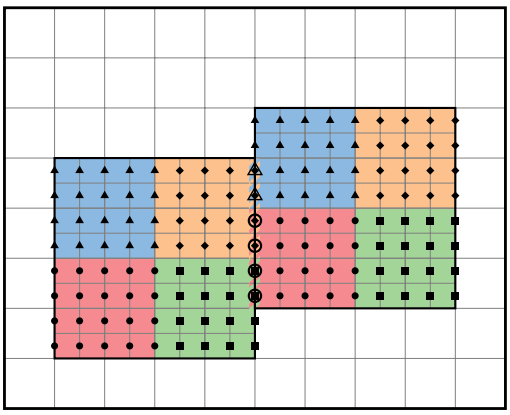

Figure 4.6: Example of face tile boxes. Each grid is *logically* broken into 4 tiles as indicated by the symbols. There are 8 tiles in total. Some tiles have 5×4 points, whereas others have 4×4 points. Points from different `Box`es may overlap, but points from different tiles of the same `Box` do not.

Usually `MFIter` is used for accessing multiple `MultiFab`s like the second example, in which two `MultiFab`s, `U` and `F`, use `MFIter` via `operator []`. These different `MultiFab`s may have different `BoxArray`s. 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 `Box`es of the same type as the `MultiFab` used in defining the `MFIter` (`F` in this example). Figures 4.5 & 4.6 show an example of non-cell-centered valid and tile boxes. Besides `validbox` and `tilebox`, `MFIter` has a number of functions returning various `Box`es. 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 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, 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 works would be wasted and for multi-threaded codes race conditions could occur. Figures 4.7 & 4.8 show examples of `growntilebox`.

These functions in `MFIter` return `Box` by value. There are two 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.
```

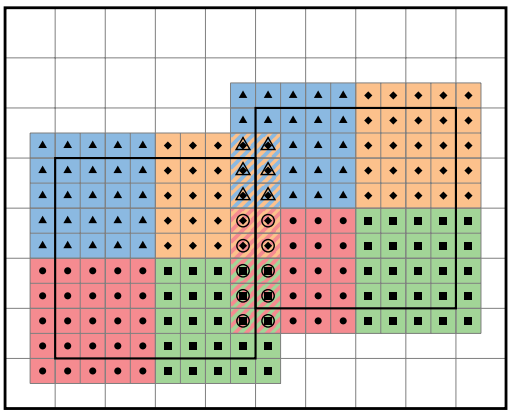

Figure 4.7: Example of cell-centered grown tile boxes. As indicated by symbols, there are 8 tiles and four in each grid in this example. Tiles from the same grid do not overlap. But tiles from different grids may overlap.

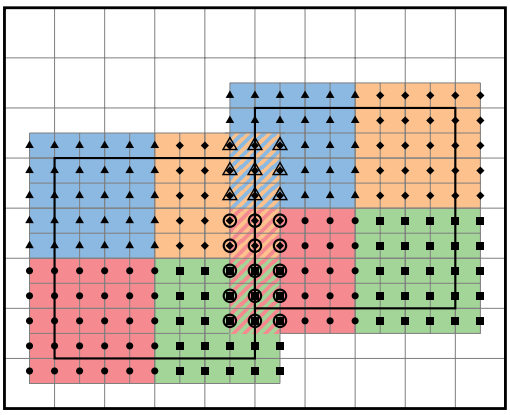

Figure 4.8: Example of face type grown tile boxes. As indicated by symbols, there are 8 tiles and four in each grid in this example. Tiles from the same grid do not overlap even though they have face index type.

```
Box bx2 = mfi.validbox();
bx2.surroundingNodes();
```

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

4.15 Calling Fortran or C

In Section 4.14, we have shown that a typical pattern for working with `MultiFabs` is 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. To C++, these kernel functions are C functions, whose function signatures are typically declared in a header file named `*_f.H` or `*_F.H`. We recommend the users to follow this convention. Examples of these function declarations are as follows.

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

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 Section 4.14, 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 Section 4.14, 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 C++ part, because when C++ passes an integer pointer pointing to an array of three integers Fortran can treat it as a 2-element integer array.

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

```
subroutine f2d(u, ulo1, ulo2, uhi1, uhi2) bind(c)
  use amrex_fort_module, only : amrex_real
```

```
  integer, intent(in) :: ulo1, ulo2, uhi1, uhi2
  real(amrex_real),intent(inout)::u(ulo1:uhi1,ulo2:uhi2)
end subroutine f2d

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

Here for simplicity we have omitted passing the tile `Box`.

Usually `MultiFab`s have multiple components. Thus we often also need to pass the number of component into Fortran functions. We can obtain the number by calling the `MultiFab::nComp()` function, and pass it to Fortran as we have seen in Section 4.14. 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
```

4.16 Boundary

AMReX uses `MultiFab` as the data container for floating point data on multiple `Box`es on a single AMR level. Each rectangular `Box` has its own boundaries. A `MultiFab` can have ghost cells for storing data outside its grid `Box` boundaries. This allows us to perform stencil type of operations on regular arrays. There are three basic types of boundaries: (1) interior boundary; (2) coarse/fine boundary; and (3) physical boundary. 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.

Interior boundary is the border among the grid `Box`es themselves. For example, in Figure 4.1, the two blue grid `Box`es on level 1 share an interior boundary that is 6 cells long. For a `MultiFab` with ghost cells on level 1, we can use the `MultiFab::FillBoundary` function introduced in Section 4.13 to fill ghost cells at the interior boundary with valid cell data from other `Box`es.

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 Section 5.2.3.

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. To fill ghost cells for cell-centered data at the physical boundary, we can call

```
  void FillDomainBoundary (MultiFab& phi, const Geometry& geom,
```

```
                                const Array<BCRec>& bc);
```

Here we need to provide boundary types to each component of the `MultiFab`. Below is an example of setting up `Array<BCRec>` before the call.

```
// Set up BC; see Src/Base/AMReX_BC_TYPES.H for supported types
Array<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 It is the user's responsibility.

foextrap 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 unchanged, $q(-i) = -q(i)$.

If the type is set to `BCType::ext_dir`, it is then the user's responsibility to fill the ghost cells.

4.17 I/O

In this section, we will discuss parallel I/O capabilities for mesh data in AMReX. Section 7.8 will discuss I/O for particle data.

4.17.1 Plotfile

AMReX has its native plotfile format. Many visualization tools are available for AMReX plotfiles (Chapter 9). 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 Array<std::string> &varnames,
```

```
                                const Geometry &geom,
                                Real time,
                                int level_step);

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

Argument `mf` (`MultiFab` for single level and `Array<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 `MultiFab`s, 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 `Array<const MultiFab*>`, whereas the multi-level data are often stored as `Array<std::unique_ptr<MultiFab>>`. AMReX has a helper function for this and one can use it as follows,

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

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

4.17.2 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.

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 `chk00016`, `chk00016/Level_0`, and `chk00016/Level_1`, we do

```
const std::string& chkname {"chk00016"};
const std::string& subDirPrefix {"Level_"};
const int nSubDirs = 2;
const bool callBarrier = true; // Parallel barrier after directories are built.
PreBuildDirectorHierarchy(chkname, subDirPrefix, nSubDirs, 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 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,

```
if (ParallelDescriptor::IOProcessor())
{
    const std::string& chkname = "chk00016";
    std::string HeaderFileName(chkname+"/Header");
    std::ofstream HeaderFile(HeaderFileName.c_str(),
         std::ofstream::out | std::ofstream::trunc | std::ofstream::binary);
    HeaderFile.precision(std::numeric_limits<Real>::max_digits10);
    VisMF::IO_Buffer io_buffer(VisMF::IO_Buffer_Size);
    HeaderFile.rdbuf()->pubsetbuf(io_buffer.dataPtr(), io_buffer.size());

    HeaderFile << "Checkpoint version 1.0\n";
    HeaderFile << time << "\n";
    HeaderFile << domain_box << "\n";
    // HeaderFile << ......;
    box_array.writeOn(HeaderFile); // write BoxArray
    // HeaderFile << ......;
}
```

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

```
std::string HeaderFileName {"chk00016/Header"};
Array<char> fileChar;
ParallelDescriptor::ReadAndBcastFile(HeaderFileName, fileChar);
std::istringstream is(std::string{fileChar.data()}, std::istringstream::in);
// is >> ....;
BoxArray ba;
ba.readFrom(is);
// is >> ....;
```

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 and read a MultiFab.

```
const std::string name {"state"};

VisMF::Write(mf, name);  // Write MultiFab to disk

// Read the data to a new MultiFab
// WARNING: mf2 may have a completely different DistributionMapping!
MultiFab mf2;
VisMF::Read(mf2, name);

// Read the data to a MultiFab with identical
// BoxArray, DistributionMapping, and number of components and ghost cells.
MultiFab mf3(mf.boxArray(), mf.DistributionMap(), mf.nComp(), mf.nGrow());
VisMF::Read(mf3, name);
```

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

4.18 Memory Allocation

AMReX has a Fortran module, mempool_module that can be used to allocate memory for Fortran pointers. The reason that such a module exists in AMReX is memory allocation is often very slow in multi-threaded OpenMP parallel regions. 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 mempool_module, only : bl_allocate, bl_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 bl_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 bl_allocate(b, lo, hi)
! ......
call bl_deallocate(a)
call bl_deallocate(b)
```

The downside of this is we have to use pointer instead of allocatable. This means we must explicitly free the memory via bl_deallocate and we need to declare the pointers as contiguous for performance reason.

4.19 Abort and Assertion

`amrex::Abort(const char* message)` is used to terminate a run usually when something goes wrong. This function takes a message and write it to stderr. Files named like `Backtrace.rg_1_rl_1` (where `rg_1_rl_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.

`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=TRUE` 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`.

CHAPTER 5

`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 Chapter7).

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 Figure 5.1.

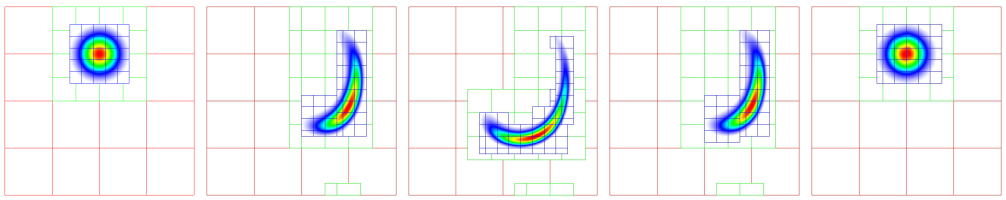

Figure 5.1: Time sequence ($t = 0, 0.5, 1, 1.5, 2$ s) of advection of a Gaussian profile using the `SingleVortex` tutorial. The red, green, and blue boxes indicate grids at AMR levels $\ell = 0, 1$, and 2.

5.1 The Advection Equation

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

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{U}). \tag{5.1}$$

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,

$$\frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n}}{\Delta t} = \frac{(\phi u)_{i+1/2,j}^{n+1/2} - (\phi u)_{i-1/2,j}^{n+1/2}}{\Delta x} + \frac{(\phi v)_{i,j+1/2}^{n+1/2} - (\phi v)_{i,j-1/2}^{n+1/2}}{\Delta y}, \tag{5.2}$$

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

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

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

Specifically, for a 3-level simulation, depicted graphically in Figure 5.2:

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 $\ell = 0, 1$.

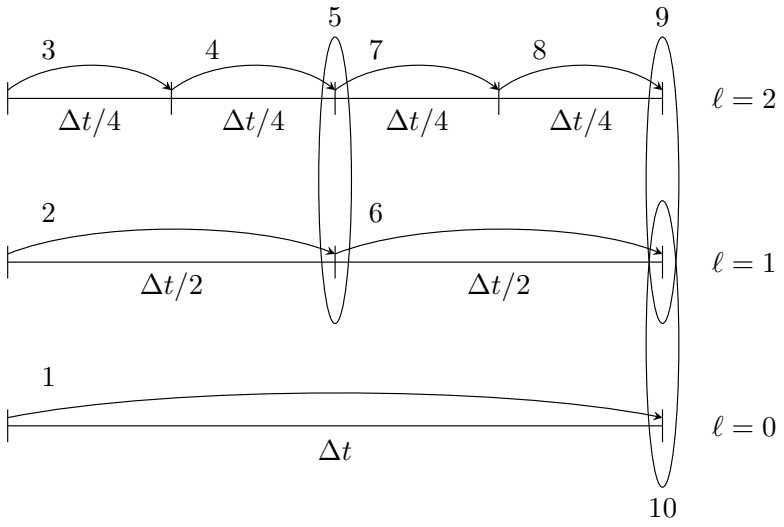

Figure 5.2: Schematic of subcycling-in-time algorithm.

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 `FABset`s 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 `AmrCore` Source Code

Here we provide a high-level overview of the source code in `amrex/Src/AmrCore`.

5.2.1 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.
    Array<IntVect> ref_ratio;        // Refinement ratios [0:finest_level-1]

    int             finest_level;    // Current finest level.

    Array<IntVect> n_error_buf;      // Buffer cells around each tagged cell.
    Array<IntVect> blocking_factor; // Blocking factor in grid generation
                                     // (by level).
    Array<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

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

The following parameters are frequently set via the inputs file or the command line. Their usage is described in Section 5.3.5

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

Table 5.1: AmrCore parameters

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.

5.2.2 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).

5.2.3 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().

A FillPatchUtil uses an Interpolator. This is largely hidden from application codes. AMReX_Interpolater.cpp contains the virtual base class Interpolater, which provides an interface for coarse-to-fine spatial interpolation operators. The fillpatch routines describe 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.

5.2.4 Using FluxRegisters

AMReX_FluxRegister.cpp/H contains the class FluxRegister, which is derived from the class BndryRegister (in 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:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \mathbf{F} \rightarrow \frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n}}{\Delta t} = \frac{F_{i+1/2,j} - F_{i-1/2,j}}{\Delta x} + \frac{F_{i,j+1/2} - F_{i,j-1/2}}{\Delta y}. \tag{5.3}$$

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.

5.2.5 AmrParticles and AmrParGDB

The AmrCore/ directory contains derived class for dealing with particles in a multi-level framework. The description of the base classes are given in Chapter 7.

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

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

5.3 Advection_AmrCore Example

5.3.1 Code Structure

Figure 5.3 shows a source code tree for the AmrAdvection_AmrCore example.

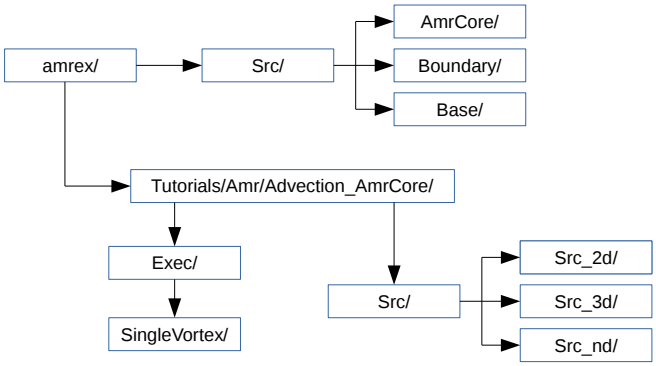

Figure 5.3: Source code tree for the AmrAdvection_AmrCore example.

- 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
        }
    }
```

5.3.2 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.

5.3.3 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
```

5.3.4 Regridding

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

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

```
void
AmrCoreAdv::timeStep (int lev, Real time, int iteration)
{
    if (regrid_int > 0)  // We may need to regrid
    {
        // regrid changes level "lev+1" so we don't regrid on max_level
        if (lev < max_level && istep[lev])
        {
```

```
            if (istep[lev] % regrid_int == 0)
            {
                // regrid could add newly refine levels
                // (if finest_level < max_level)
                // so we save the previous finest level index
                int old_finest = finest_level;
                regrid(lev, time);

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

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

```
    {
        Array<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

  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
```

5.3.5 Grid Creation

The gridding algorithm proceeds in this order, using the parameters described in Section5.2.

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 taggd cells from Section 5.3.4, 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`.

5.3.6 FillPatch

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

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

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

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

CHAPTER 6

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, publicly 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 Chapter 5, except now we use the provided libraries in 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,
                           &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 Advection_AmrLevel Example

Figure 6.1 shows a source code tree for the AmrAdvection_AmrLevel example.

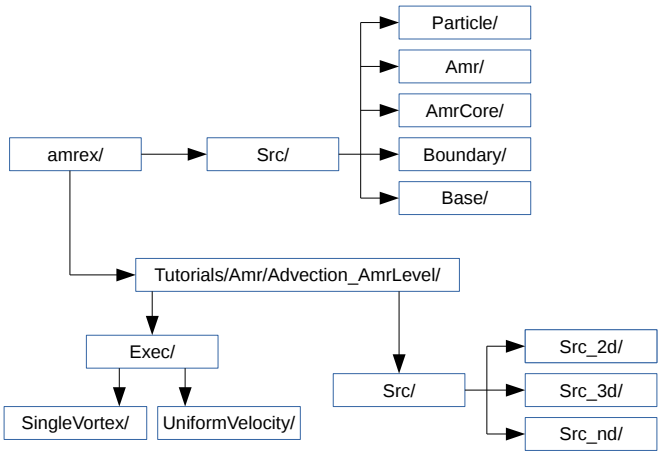

Figure 6.1: Source code tree for the AmrAdvection_AmrLevel example.

- 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 Chapter 9).

CHAPTER 7

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 automatically. In the following sections, we give an overview of AMReX's particle classes and how to use them.

7.1 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 variables 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]

[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 its currently assigned

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: x y z m vx vy vz id cpu flag1 flag2. [3]

7.1.1 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.rdata(0) = 17;
p.idata(1) = -64;
```

7.2 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;
```

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.

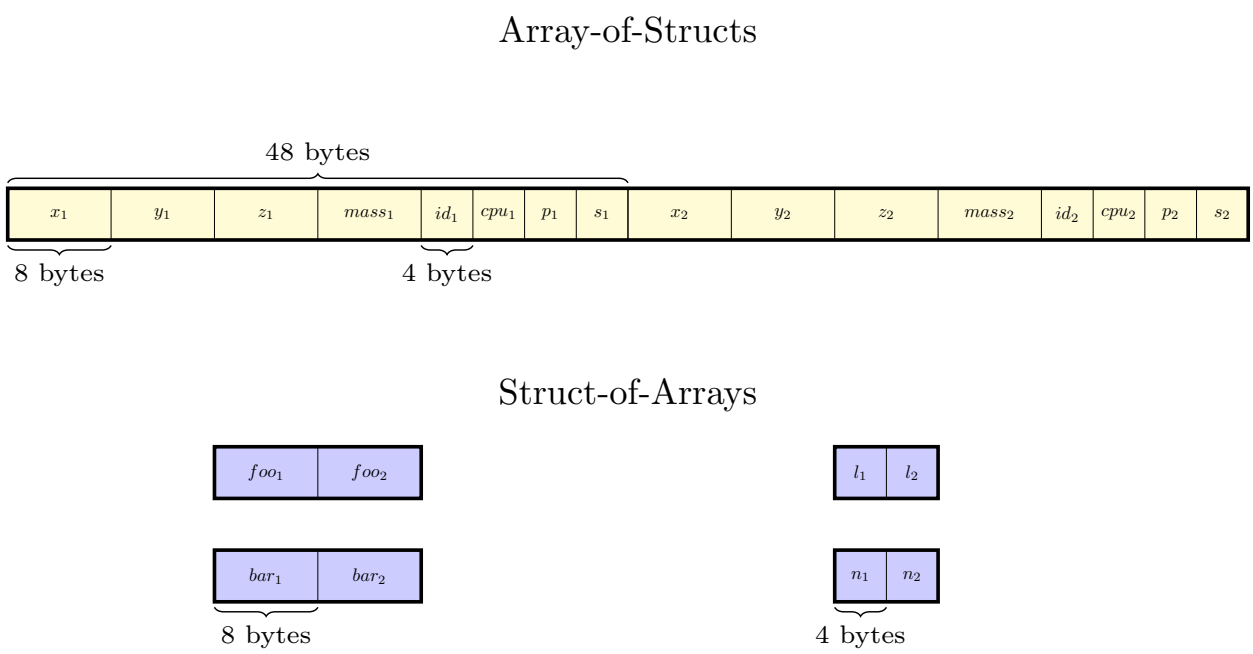

Figure 7.1: An illustration of how the particle data for a single tile is arranged in memory. This particle container has been defined with `NStructReal` = 1, `NStructInt` = 2, `NArrayReal` = 2, and `NArrayInt` = 2. In this case, each tile in the particle container has five arrays: one with the particle 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.

7.2.1 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 Figure 7.1 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 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.

7.2.2 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 Array<Geometry>            & geom,
                   const Array<DistributionMapping> & dmap,
                   const Array<BoxArray>            & ba,
                   const Array<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 Chapter 5), 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 Array < BoxArray > and Array < 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.

7.3 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).

7.4 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) {
    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();
    Array<Real>& real_comp0 = particle_attributes.GetRealData(0);
    Array<int>&  int_comp1  = particle_attributes.GetIntData(1);
    for (int i = 0; i < pti.numParticles; ++i) {
        // do stuff with your SoA data...
    }
}
```

7.5 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 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.

7.6 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());
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 `MultiFab`s that contain the electric field data. These `MultiFab`s 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` has been defined on the same grids as the electric field `MultiFabs`, so that we use the `ParIter` to index into the `MultiFab`s 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`. The FArrayBox's 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.

7.7 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 Figure 7.2 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. Computing the forces on the valid particles can then proceed via direct summation.

For a complete example of a `ParticleContainer` that does this neighbor finding, please see amrex/Tutorials/Particles/ShortRangeParticles. This `ParticleContainer` has additional methods called `fillNeighbors`() and `clearNeighbors`() that fill the `neighbors` data structure with copies of the proper particles. These are then passed into the Fortran routine that computes the forces as follows:

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

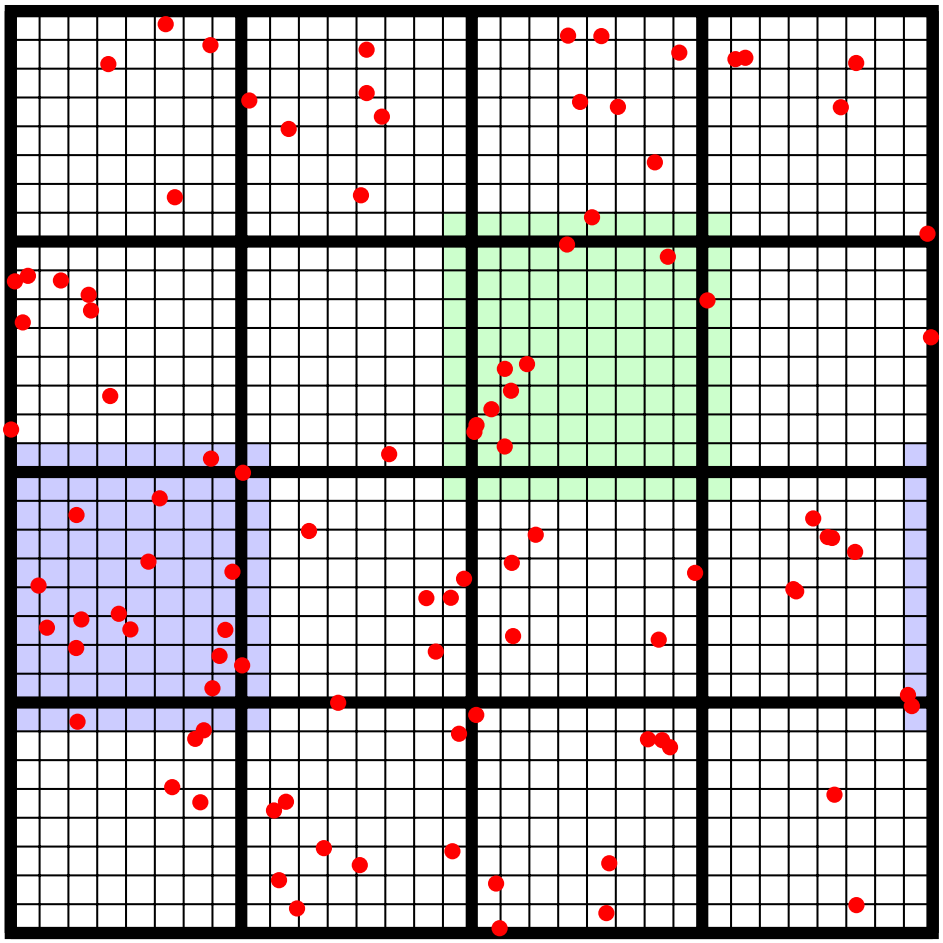

Figure 7.2: An illustration of filling neighbor particles for short-range force calculations. Here, we have a domain consisting of one 32-by-32 grid, broken up into 8-by-8 tiles. The number of ghost cells is taken to be 1. For the tile in green, particles on other tiles in the entire shaded region will 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.

7.8 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, GetArrOfConstPtrs(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 Chapter 9 for more information on visualizing `AMReX` datasets, including those with particles.

CHAPTER 8

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 (Chapter 6). and particles (Chapter 7) We should not confuse the Fortran interface in this chapter with the Fortran kernel functions called inside `MFIter` loops in C++codes (Section 4.15). 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 User's Guide except for Chapters 6 & 7.

8.1 Getting Started

We have discussed AMReX's build systems in Chapter 3. 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 `AmrCore` class (Chapter 5), and the `Octree` adds support for octree type of AMR grids. Each directory has a `Make.package` file that can be included in make files (see `Tutorials/Basic/HelloWorld_F` and `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 `Tutorials/Basic/HelloWorld_F/`. The source code is shown below in it entirety.

```
subroutine amrex_fmain () bind(c)
  use amrex_base_module
  implicit none
  if (amrex_parallel_ioprocessor()) then
     print *, "Hello world!"
```

```
  end if
end subroutine amrex_fmain
```

Note that there is no `program main` in the Fortran code. The `main` function is actually in `Src/F_Interfaces/Base/AMReX_fi_main.cpp`. That `main` function initializes and finalizes AMReX, and calls `amrex_fmain` function in between.

To access AMReX Fortran interfaces, we can use these three modules, `amrex_base_module` for the basics functionalities (Section 8.2), `amrex_amrcore_module` for AMR support (Section 8.3) and `amrex_octree_module` for octree style AMR (Section 8.4).

8.2 The Basics

Module `amrex_base_module` is a collection of various Fortran modules providing interfaces to most of the basics of AMReX C++ library Chapter 4). 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 coordinate direction index starts with 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.

Module `amrex_parallel_module` (`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).

Module `amrex_parmparse_module` (`Src/Base/AMReX_parmparse_mod.F90`) provides interface to `ParmParse` (Section 4.6. 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 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 (Src/F_Interfaces/Base/AMReX_boxarray_mod.F90) is a wrapper for the BoxArray class, and amrex_distromap (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
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 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 `amrexMultiFab`'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 AMReX (e.g., `amrex_multifab`, `amrex_boxarray`, `amrex_distromap`, `amrex_mfiter`, and `amrex_geometry`) contain a `type(c_ptr)` that points a C++ object. They also contain a logical type indicating whether or not this object owns the underlying object (i.e., responsible for deleting the object). Due to the semantics of Fortran, one should not return these types with functions. Instead we should pass them as arguments to procedures (preferably with `intent` specified). These five types all have `assignment(=)` operator that performs a shallow copy. After the assignment, the original objects still owns the data and the copy is just an alias. For example,

```
type(amrex_multifab) :: mf1, mf2
call amrex_multifab_build(mf1, ...)
call amrex_multifab_build(mf2, ...)
! At this point, both mf1 and mf2 are data owners
mf2 = mf1    ! This will destroy the original data in mf2.
             ! Then mf2 becomes a shallow copy of mf1.
             ! mf1 is still the owner of the data.
call amrex_multifab_destroy(mf1)
! mf2 no longer contains a valid pointer because mf1 has been destroyed.
call amrex_multifab_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.

8.3 Amr Core Infrastructure

Module `amrex_amr_module` provides interfaces to AMR core infrastructure. With AMR, subroutine `amrex_fmain` should look like below,

```
subroutine amrex_fmain () bind(c)
  use amrex_amr_module
  implicit none
  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()
end subroutine amrex_fmain
```

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 C++ TagBoxArray object. We need to convert this into a Fortran amrex_tagboxarray object.

```
type(amrex_tagboxarray) :: tag
tag = tags
```

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` (Section 5.2.4). Its usage is demonstrated in the tutorial at `Tutorials/Amr/Advection_F/`.

8.4 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 AMReX grids are more general than octree grids. A tutorial example using `amrex_octree_module` (`Src/F_Interfaces/Octree/AMReX_octree_mod.f90`) is available at `Tutorials/Amr/Advection_octree_F/`. Procedures `amrex_octree_init` and `amrex_octree_finalize` must be called as follows,

```
subroutine amrex_fmain () bind(c)
  use amrex_amrcore_module
  use amrex_octree_module
  implicit none
  call amrex_octree_int()  ! This should be called first.
  call amrex_amrcore_init()
  call my_amr_init()       ! user's own code, not part of AMReX
  ! ...
  call my_amr_finalize()   ! user's own code, not part of AMReX
  call amrex_amrcore_finalize()
  call amrex_octree_finalize()
end subroutine amrex_fmain
```

By default, the grid size is 8^3, and this can be changed via `ParmParse` parameter `amr.max_grid_size`. 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 9

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.

9.1 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. Here are some instructions and tips for using Amrvis:

1. Download and build Amrvis:

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

   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 Figure 9.1). 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.

9.2 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 Section 2.3).

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".

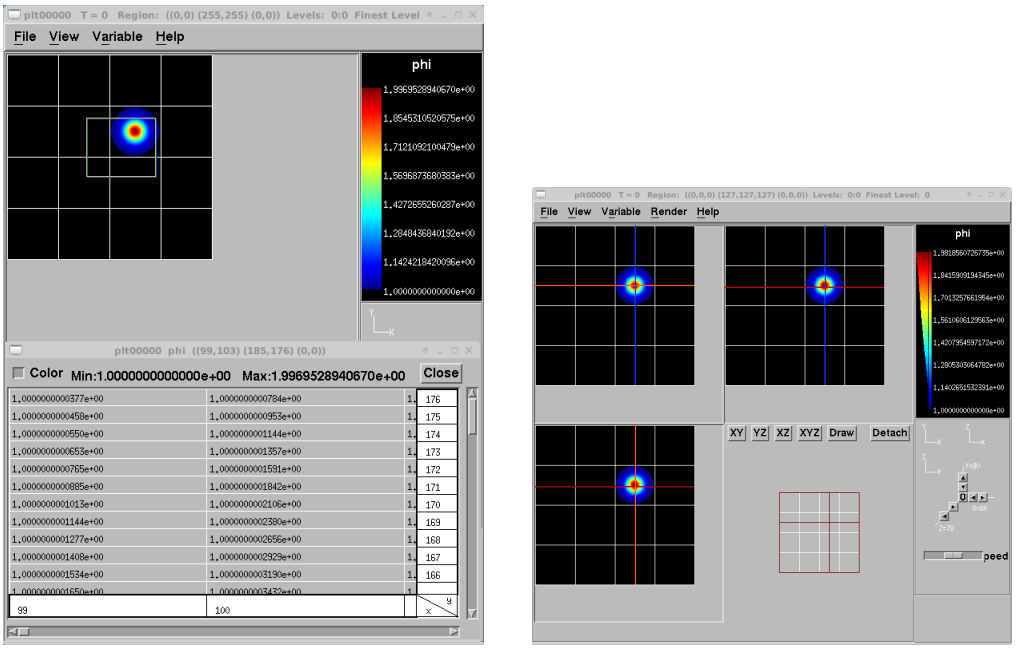

Figure 9.1: 2D and 3D images generated with Amrvis

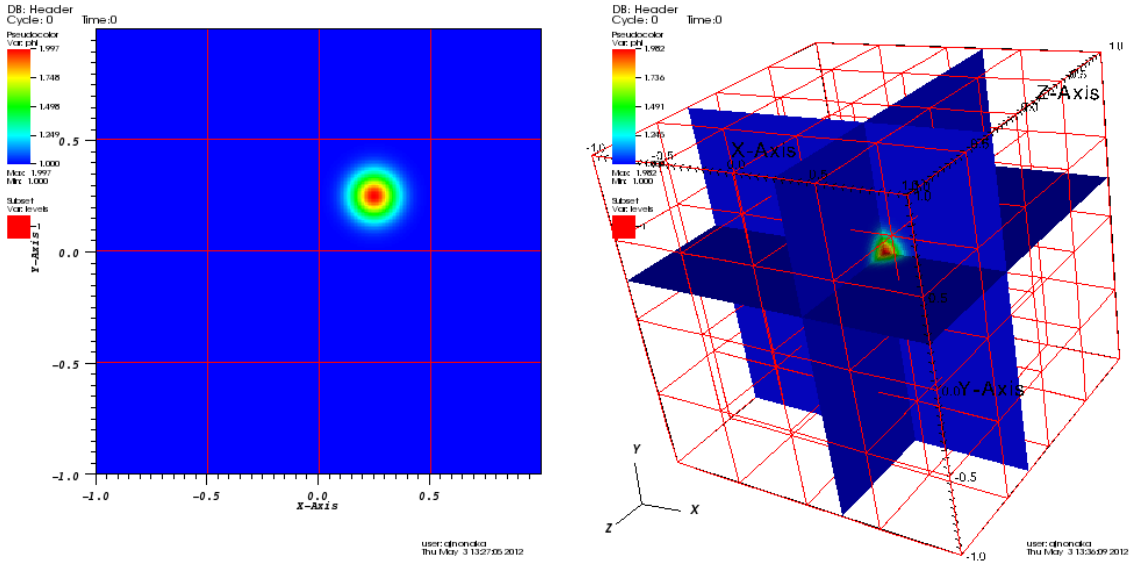

Figure 9.2: (Left) 2D image generated with VisIt. (Right) 3D image generated with VisIt.

- 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 Figure 9.2.

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 Figure 9.2.

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.

9.3 ParaView

The open source visualization package ParaView v5.3.0 can be used to view 3D plotfiles, and v5.4.0 can be used to view 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 v5.3.0, then select "File" → "Open".
2. Navigate to 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 Figure 9.3.

To visualize particles (for example, you could run the `ShortRangeParticles` example:

1. First, we have to convert the AMReX particle data to a format ParaView can read. In the run directory, there will be a sequence of particle files (`particles00000`, `particles00001`, ⋯, `particles01000`).

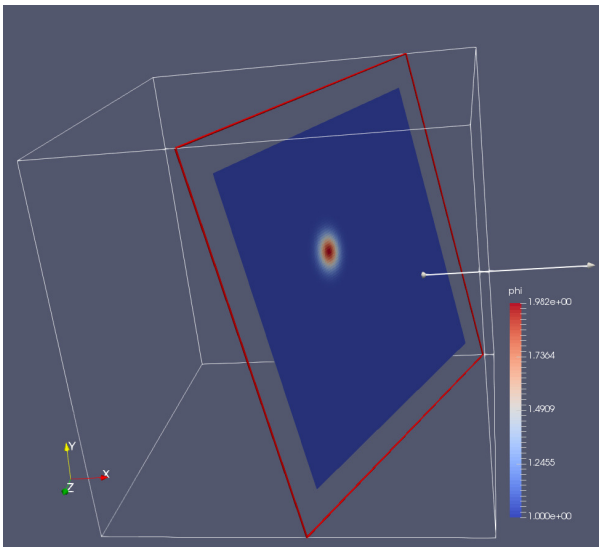

Figure 9.3: Plotfile image generated with ParaView

2. Run the script, `amrex/Tools/Py_util/amrex_particles_to_vtp/amrex_particles_to_vtp.py` as follows, e.g., `python amrex_particles_to_vtp.py 0 1000 particles`. You will generate a sequence of `.vtp` files.
3. Run ParaViewv5.4.0, and select "File" → "Open". You will see a combined "particles..vtp" file grouping the files. Select that and click OK.
4. Click "Apply" and under "Representation" select "Point Gaussian".
5. Change the Gaussian Radius if you like. You can scroll through the frames with the VCR-like controls at the top, as shown in Figure 9.4.

9.4 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 yt on both a local workstation, as well as at the NERSC HPC facility for high-throughput visualization of large data sets.

9.4.1 Using yt 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__)
```

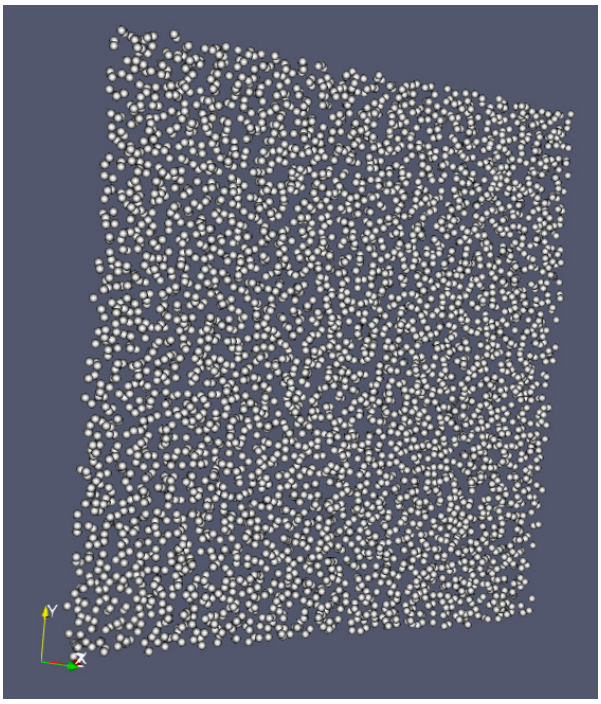

Figure 9.4: Particle image generated with ParaView

```
3.4-dev
```

Next, load a plot file; in this example we use a plot file from the Nyx cosmology application:

```
In [3]: ds = yt.load("plt00401")
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: current_time              =
     0.00605694344696544
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_dimensions         =
     [128 128 128]
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_left_edge          =
     [ 0.  0.  0.]
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'),
```

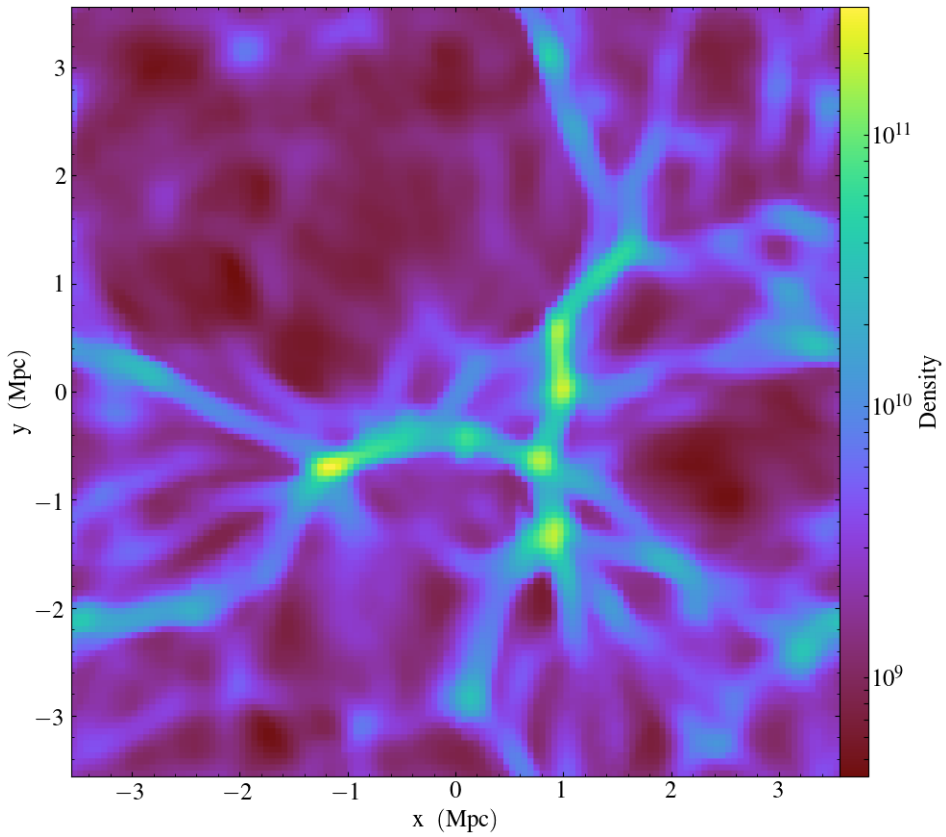

Figure 9.5: Slice plot of 128^3 Nyx simulation using yt.

```
 ('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 Figure 9.5. One can also make volume renderings with yt; 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:
    <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 Figure 9.6.

9.4.2 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.

9.4.2.1 Interactive yt with Jupyter notebooks

Unlike VisIt (§9.2), 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.

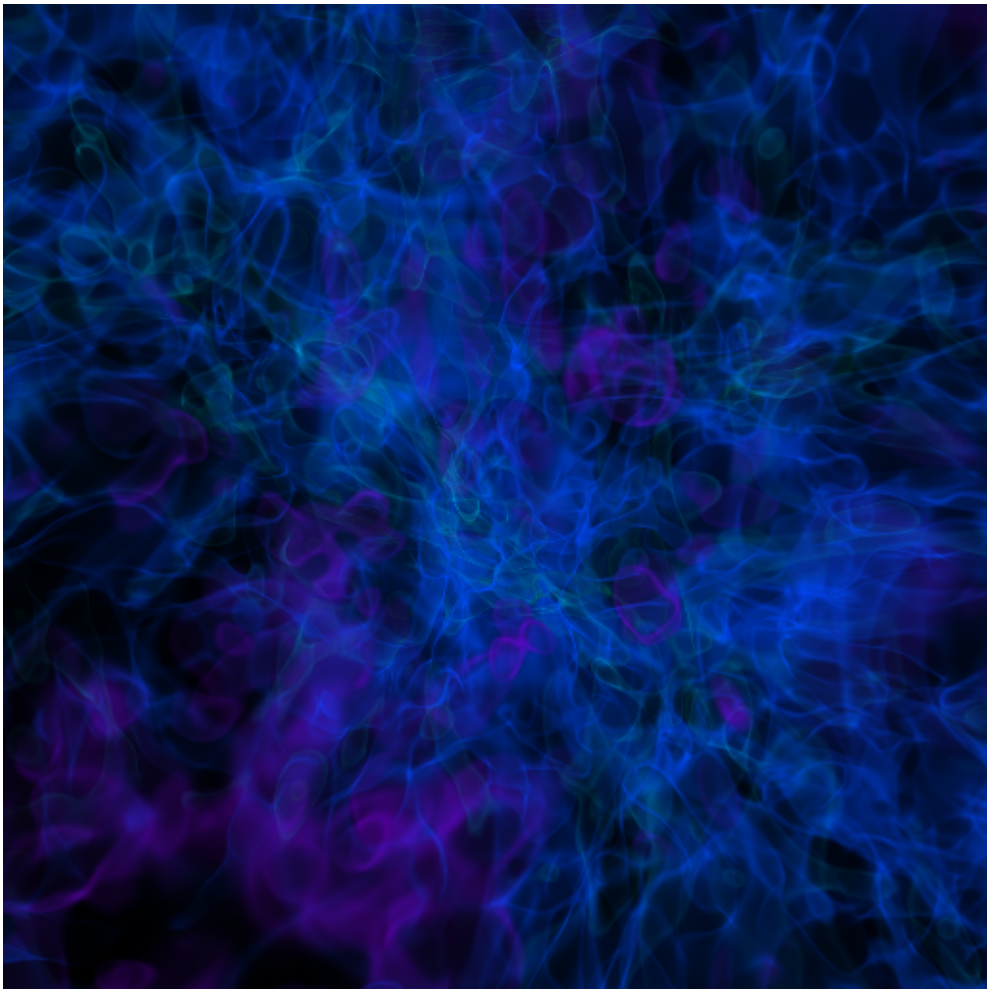

Figure 9.6: Volume rendering of 128^3 Nyx simulation using yt. This corresponds to the same plot file used to generate the slice plot in Figure 9.5.

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 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]

9.4.2.2 Parallel yt

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 on 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

[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.

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'

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.]
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.

CHAPTER 10

Profiling

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. The broad-brush C++ instrumentation is as follows:

10.1 Instrumenting the Code

10.1.1 C++

```
void YourClass::YourFunction()
{
  BL_PROFILE("YourClass::YourFunction()");  // this name can be any string

  // your function code
}
```

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);
```

10.1.2 Fortran90

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.

10.2 Types of Profiling

Currently you have two options for AMReX-specific profiling. If you set `TINY_PROFILE = TRUE` in your GNUMakefile then at the end of the run, a summary of exclusive and inclusive function times will be written to stdout.

If you set `PROFILE = TRUE` then a `bl_prof` directory will be written that contains detailed per-task timings of the code. An exclusive-only set of function timings will be written to stdout

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.

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.

10.3 Sample Output

Sample output from `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%
```

10.4 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 11

CVODE

AMReX supports 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 (`https://computation.llnl.gov/sites/default/files/public/cv_guide.pdf`).

The second, newer Fortran interface to CVODE uses the `iso_c_binding` feature of the Fortran 2003 standard to implement a "thinner," more direct interface to the C functions in CVODE. In contrast to the FCVODE interface, the Fortran 2003 interface calls C functions directly, with identical arguments. When compiling CVODE, one need not build the Fortran interface to CVODE at all to use this new interface. The examples provided in AMReX use this new interface.

To use CVODE in an AMReX application, follow these steps:

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 \
   ```

[1] `https://computation.llnl.gov/projects/sundials/cvode`
[2] `https://computation.llnl.gov/projects/sundials`

```
-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
```

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