Manual for simulating isothermal dendrites of a binary alloy using a Parabolic Grand Potential solver implemented on AMReX

Abhishek Kalokhe and Nasir Attar

December 8, 2022

1 Introduction

This solver aims to simulate the phase separation of a binary alloy in the spinodal region of the phase diagram. The absence of thermodynamic barrier in spinodal region forces the decomposition to only be driven by diffusion. Finite difference discretization is implemented which is second order accurate in space while forward Euler time marching scheme is used for temporal discretization.

2 Infile

The input parameters required for the solver are derived from an infile. This contains information about the domain geometry, the thermodynamic functions(free energy), material properties such as the interfacial energies and their anisotropies as well as boundary conditions. It could also contain special flags related to the running of the solver. The following is a basic description of the keys in the infile. **Each key must end with a semicolon**. Additionally, all lines beginning with "#" will be treated as comments in the infile.

```
##Geometrical dimensions of the simulation domain
DIMENSION = 2;
MESH_X = 1000;
MESH_Y = 1000;
MESH_Z = 1;
##Discretization, space and time
DELTA_X = 5e-8;
DELTA_Y = 5e-8;
DELTA_Z = 5e-8;
DELTA_t = 6e-7;
##Number of phases and composition
NUMPHASES = 2;
NUMCOMPONENTS = 2;
#Running and saving information
NTIMESTEPS = 2000000;
NSMOOTH = 10;
SAVET = 10000;
```

```
STARTTIME = 0;
RESTART = 0;
numworkers = 16;
## Component and Phase names
# COMPONENTS = {Al,Cu,B};
COMPONENTS = {Al, Zn};
PHASES = {alpha, liquid};
##Material properties
##GAMMA={12, 13, 14, 23, 24...}
GAMMA = {0.1};
# Diffusivity = {Diagonal:0/1, phase, 11,22,33, 12, 13, 23...};
DIFFUSIVITY = {1, 0, 0};
DIFFUSIVITY = {1, 1, 1e-9};
##Gas constant and molar volume
R = 8.314;
V = 10e-6;
##Elasticity related parameters
EIGEN_STRAIN = {0, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
EIGEN_STRAIN = {1, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
VOIGT_ISOTROPIC = {0, 270, 187.5, 125.0};
VOIGT_ISOTROPIC = {1, 270, 187.5, 125.0};
#VOIGT_CUBIC = {phase, c11, c12, c44};
#VOIGT_TETRAGONAL = {phase, c11, c12, c13, c33, c44, c66};
##Boundary conditions
#0: Free, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex
#Boundary = {phase, X+, X-, Y+, Y-, Z+, Z-}
BOUNDARY = {phi, 1, 1, 1, 1, 0, 0};
BOUNDARY = {mu, 1, 1, 1, 1, 0, 0};
BOUNDARY = {c, 1, 1, 1, 1, 0, 0};
BOUNDARY = {T, 1, 1, 1, 1, 0, 0};
# Boundary = {phi, 1, 1, 0};
# Boundary = {"u", 3, 3, 2, 2};
#Boundary_value = {Value X+, Value X-, Value Y+, Value Y-, Value
    ↪ Z+, Value Z-}
BOUNDARY_VALUE = {phi, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {mu, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {c, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {T, 0, 0, 0, 0, 0, 0};
##Type of simulation
ISOTHERMAL = 1;
BINARY = 1;
#TERNARY
DILUTE = 0;
T = 866;
##FILEWRITING and OUTPUTTING TO SCREEN
## WRITEFORMAT ASCII/BINARY/HDF5(Only for MPI)
##TRACK_PROGRESS: interval of writing out the progress of the
    ↪ simulation to stdout.
WRITEFORMAT = ASCII;
WRITEHDF5 = 1;
```

```
TRACK_PROGRESS = 1000;
##Model-specific parameters: Grand-potential model
##Phase-field parameters; epsilon:interface width; it is not the
    ↪ gradient energy coefficient
epsilon = 2e-7;
tau = 1.31;
Tau = {0.28};
##Anisotropy functions
##Anisotropy mode, FUNCTION_ANISOTROPY=0 is isotropic
Function_anisotropy = 1;
Anisotropy_type = 4;
dab = {0.01};
#Rotation_matrix = {0, 1, Euler_x(ang), Euler_y(ang), Euler_z(ang
    ↪ )};
Rotation_matrix = {0, 1, 0, 0, 45};
##Potential function
Function_W = 1;
Gamma_abc = {};
#Shifting of domain for infinite domain simulations
Shift = 0;
Shiftj = 30;
#Writing of composition fields along with the chemical potential
    ↪ fields
Writecomposition = 1;
#Noise
Noise_phasefield = 1;
Amp_Noise_Phase = 0.1;
##Temperature
Equilibrium_temperature = 870;
Filling_temperature = 866;
#TEMPGRADY={BASETEMP, DELTAT, DISTANCE, OFFSET, VELOCITY}
Tempgrady = {0.96, 0.06, 800.0, 0, 0.016};
##Function_F
Function_F = 4;
A = {0, 1};
A = {1, 1};
#ceq = {0, 0, 0.83};
#ceq = {0, 1, 0.55};
#ceq = {1, 1, 0.55};
#ceq = {1, 0, 0.55};
#cfill = {0, 0, 0.83};
#cfill = {0, 1, 0.55};
#cfill = {1, 1, 0.55};
#cfill = {1, 0, 0.55};
#slopes = {0, 0, 645};
#slopes = {0, 1, 238};
#slopes = {1, 0, 238};
#slopes = {1, 1, 238};
ceq = {0, 0, 0.926};
ceq = {0, 1, 0.817};
```

```
ceq = {1, 1, 0.817};
ceq = {1, 0, 0.817};
cfill = {0, 0, 0.926};
cfill = {0, 1, 0.817};
cfill = {1, 1, 0.817};
cfill = {1, 0, 0.817};
c_guess = {0, 0, 0.92133};
c_guess = {0, 1, 0.80354};
c_guess = {1, 1, 0.80354};
c_guess = {1, 0, 0.80354};
slopes = {0, 0, 666};
slopes = {0, 1, 333};
slopes = {1, 0, 333};
slopes = {1, 1, 333};
num_thermo_phases = 2;
tdbfname = alzn_mey.tdb;
tdb_phases = {FCC_A1, LIQUID};
phase_map = {FCC_A1, LIQUID};
```

2.1 Simulation geometry, spatial and temporal discretization

```
##Geometrical dimensions of the simulation domain
DIMENSION = 2;
MESH_X = 100;
MESH_Y = 100;
MESH_Z = 1;
```

- DIMENSION: This can take values 2,3 for 2D and 3D simulations respectively. This is a required key in the solver and not mentioning this key might lead to unexpected results
- MESH_X,MESH_Y,MESH_Z: These are the number of grid points in the domain(and not the physical lengths) in the respective X, Y, Z directions. When DIMENSION is 2, the value of MESH_Z will redundant and will be taken as 1.

```
##Discretization, space and time
DELTA_X = 2.0;
DELTA_Y = 2.0;
DELTA_Z = 2.0;
DELTA_t = 0.08;
```

- The values DELTA_X, DELTA_Y, DELTA_Z correspond to the grid resolution in the X, Y, Z directions respectively. Similarly, DELTA_t corresponds to the temporal discretization(time-step).

2.2 Phases and Components information

```
##Number of phases and composition
NUMPHASES = 2;
NUMCOMPONENTS = 2;
```

- The keys are self-explanatory. NUMPHASES corresponds to the number of phases in the domain, while NUMCOMPONENTS corresponds to the number of components(2, for binary, 3 for ternary etc.)
- These keys are absolutely necessary, please fill carefully for successful running of your codes.

```
COMPONENTS = {Al, Cu};
PHASES = {alpha, beta};
```

- COMPONENTS refers to the tuple that consists of the names of the components. The names are separated by commas and the entire tuple needs to placed within {} followed by a semicolon.
- Similarly, PHASES refers to the names of the phases in the domain.

2.3 Boundary conditions

```
##0:FREE, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex
##BOUNDARY = {TYPE, X_LEFT, X_RIGHT, Y_FRONT, Y_BACK, Z_TOP,
    ↪ Z_BOTTOM}
BOUNDARY = {phi, 1, 1, 1, 1, 0, 0};
BOUNDARY = {mu, 1, 1, 1, 1, 0, 0};
BOUNDARY = {c, 1, 1, 1, 1, 0, 0};
BOUNDARY = {T, 1, 1, 1, 1, 0, 0};
```

- Any of the solvers in repository will consist of the following scalar default types. Type "phi" will represent the phase-field order parameters whose number is specified by the key, "NUMPHASES". Depending on the solver whether it is the grand-potential based solver, where "mu" will be the independent variable or if it is the Cahn-Hilliard or Kim-Kim-Suzuki based solvers, where "c" is the independent variable, Type "mu" or "c" will refer to the components in the key "COMPONENTS". Similarly, type "T" will refer to the temperature field. The BOUNDARY key will refer to the boundary condition for the respective Type of field. The following numbers in a given tuple refer to the boundary at the respective (X_LEFT, X_RIGHT, Y_FRONT, Y_BACK, Z_TOP, Z_BOTTOM) boundaries, that refer to the X, Y, Z boundaries in order. The boundary condition is specified by numbers, 0:FREE, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex, where for the DIRICHLET boundary condition the respective boundary can also take in a specified value which can be specified in the following key: BOUNDARY_VALUE. If the key is not present, the default

remains the one that is initialized at the start of the simulation and is left unchanged. Further, if for any direction, for eg: X, one of the extremeties is specified as a PERIODIC boundary, then the other X-boundary will also be initialized as PERIODIC irrespective of the entry in the tuple.

```
#BOUNDARY_VALUE = {Type, Value X+, Value X-, Value Y+, Value Y-,
    ↪ Value Z+, Value Z-}
BOUNDARY_VALUE = {phi, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {mu, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {c, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {T, 0, 0, 0, 0, 0, 0};
```

- This key corresponds to a specific value that needs to be specified on a boundary which will be held constant during the duration of the simulation. The definition of this key should follow the BOUNDARY specification and follows the same type of definition, except here the tuple Value X_LEFT, Value X_RIGHT, Value Y_FRONT, Value Y_BACK, Value Z_TOP, Value Z_BOTTOM, refer to values on the respective boundaries. The values in this tuple will only be utilized if the respective boundary condition on that boundary is DIRICHLET. By default, the value will be treated as the same as the one that is initialised at the start of the simulation, which will be utilized if this key is not present.

2.4 Number of iterations, smoothing time-steps and writing interval

```
#Running and saving information
NTIMESTEPS = 10000;
NSMOOTH = 10;
SAVET = 1000;
STARTTIME = 6000;
RESTART = 1;
numworkers = 4;
```

- NTIMESTEPS: Total number of iterations that you wish the solver to run. This is not the total time
- NSMOOTH: The number of pre-conditioning steps for smoothening sharp phase-field profiles that are initialised at the start of the simulation
- SAVET: Writing interval, i.e, frequency of writing files of the respective fields
- STARTTIME: The iteration number from which the simulation will start.
- RESTART : This key tells the solver to restart by reading in the files corresponding to STARTTIME. If either of the keys STARTTIME/RESTART are non-zero the code will restart after reading in the files corresponding to the value STARTTIME. If STARTTIME = 0 and RESTART = 1, it gives the possibility to start from an already initialized file. This is useful when the filling operations for initialization are time-consuming.

- Number of workers with which the simulation was executed previously. This is required only for the Grand-potential based MPI solvers.

2.5 Material parameters

```
##Gas constant and molar volume
R = 1.0;
V = 1.0;
```

- The values "R" and "V" will refer to the gas constant and the molar volume respectively

```
#DIFFUSIVITY={Diagonal:0/1, phase, 11,22,33..(K-1) diagonal
    ↪ elements, 12, 13, 23...(rest of the elements; rowwise)}
DIFFUSIVITY = {1, 0, 1};
```

- The DIFFUSIVITY key refers to the inter-diffusivity matrix which has the tuple in the following form. The first element can taken in values 0/1, "1" refers to as a diagonal matrix and "0" is a full matrix.
- The following element refers to the phase number referring to the phases in the list PHASES. This can take values from 0 to NUMPHASES-1.
- The following elements are the values in the inter-diffusivity matrix. The first elements are the diagonal terms in the matrix, while the following elements correspond rowwise to the off-diagonal terms in the diffusivity matrix.
- If the first element in the tuple is "1", i.e. the matrix is diagonal irrespective of the number of entries in the tuple only the entries corresponding to the diagonal elements in the matrix will be read in.

```
##GAMMA = {12, 13, 14, 23, 24...}
GAMMA = {1.0};
```

- The GAMMA key refers to the interfacial energy $\gamma_{\alpha\beta}$ between the phases $\alpha\beta$. The elements in tuple correspond to all combination of phases forming the interfaces from the list of phases in PHASES numbered from 0 to NUMPHASES-1.
- The elements are numbered in the order $12, 13, 14, 23, 24 \dots N(N-1)$, $N = NUMPHASES$ where "12" corresponds to the value of the interfacial energy between phase 1 and 2; γ_{12}.
- In the tuple only combinations $\alpha\beta$ are included where $\alpha < \beta$ as the value of the interfacial energy of the $\alpha\beta$ interface is $\gamma_{\alpha\beta}$ which is the same as $\gamma_{\beta\alpha}$.
- Therefore, the total number of elements in the tuple is $\frac{N(N-1)}{2}$.

```
#EIGEN_STRAIN = {phase, exx, eyy, ezz, eyz, exz, exy};
EIGEN_STRAIN = {0, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
EIGEN_STRAIN = {1, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
```

- The EIGEN_STRAIN key refers to the eigen-strain tensor in a given phase. The information about the elements of the eigen-strain are derived from the elements in the tuple.
- The first element refers to the phase number in the list PHASES and can take in values from 0 to NUMPHASES-1.
- The following elements in tuple are mapped to the eigen-strain matrix in the following order $exx, eyy, ezz, eyz, exz, exy$.
- This is an important key required for problems where there are coherent interfaces and the phase transformation is coupled with the stress distribution arising due to coherency strains at the interface.

```
#VOIGT_ISOTROPIC = {phase, c11, c12, c44};
VOIGT_ISOTROPIC = {0, 270, 187.5, 125.0};
VOIGT_ISOTROPIC = {1, 270, 187.5, 125.0};
```

- VOIGT_ISOTROPIC is the key that refers to the elastic stiffness properties in the Voigt notation for an isotropic material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1.
- The following elements are the values $C11, C12, C44$ in order.

```
#VOIGT_CUBIC = {phase, c11, c12, c44};
VOIGT_CUBIC = {0, 270, 187.5, 125.0};
VOIGT_CUBIC = {1, 270, 187.5, 125.0};
```

- VOIGT_CUBIC is the key that refers to the elastic stiffness properties in the Voigt notation for a cubic material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1
- The following elements are the values $C11, C12, C44$ in order

```
#VOIGT_TETRAGONAL = {phase, c11, c12, c13, c33, c44, c66};
```

- VOIGT_TETRAGONAL is the key that refers to the elastic stiffness properties in the Voigt notation for a tetragonal material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1.

- The following elements are the values $C11, C12, C13, C33, C44, C66$ in order.

```
BINARY = 1;
DILUTE = 0;
```

- The keys "BINARY", "TERNARY", "DILUTE" are special flags to the solver allowing for simpler routines in the update of the chemical potential or composition fields.

```
##FILEWRITING and OUTPUTTING TO SCREEN
## WRITEFORMAT ASCII/BINARY
##TRACK_PROGRESS: interval of writing out the progress of the
    ↪ simulation to stdout.
WRITEFORMAT = ASCII;
WRITEHDF5 = 0/1;
TRACK_PROGRESS = 10;
```

- The key WRITEFORMAT mentions the type of the output files to be written. The possible values are ASCII or BINARY that are self-explanatory. When running in parallelized mode with MPI, there is a third possibility for writing files using the key WRITEHDF5 key. For MPI runs, if the WRITEHDF5 key is non-zero, then the value of the WRITEFORMAT key is ignored and files in hdf5 format are written for each time-step (files with extension .h5). Since, the hdf5 file library is linked to the solver by default, the solver needs to be compiled with h5pcc for parallel runs instead of mpicc. This is irrespective of the value of the key WRITEHDF5.
- For the case when the type chosen is BINARY, the format is BIG-ENDIAN such that it matches the BINARY type required for PARAVIEW.
- For MPI runs, in the event WRITEHDF5=0, the WRITEFORMAT key decides the format of the output file, where each processor writes its own file for each time given by the key saveT. In order to view the consolidated file, the tool (./reconstruct) needs to executed in the folder just outside the /DATA folder that is created. The following command needs to be executed for reconstructing,

```
./reconstruct name_of_infile name_of_output_file
    ↪ number_of_workers start_time end_time
```

- For the case when WRITEHDF5=1, all processors write collectively into a single .h5 file. For viewing the file in paraview, it needs to be put in .xml format, which can be performed using the following command, that needs to be executed from just outside the /DATA folder that is created using the run,

```
./write_xdmf name_of_infile name_of_output_file
    ↪ number_of_workers start_time end_time
```

- TRACK_PROGRESS is a key that will inform the solver about the frequency with which the progress of the simulation will be written to stdout.
- The number can be anything other than 0.

2.6 Input file for AMReX

AMReX directly cannot read the inputs obtained from the infile generator, to make the infile compatible for AMReX few changes are required. Two input files are generated during the syntax change process, first input file(say *input1.in*) is generated when the *filling.in* is merged with the *input.in* (*input.in* is generated from the infile generator), and the second input file(say *input2.in*) is generated when required changes(refer to Fig.1.) are made in *input1.in*. The *input2.in* is then read by AMReX, and further calculations are performed. Note that only the syntax of the data is changed when writing the new input files, no data is tampered with during the process.

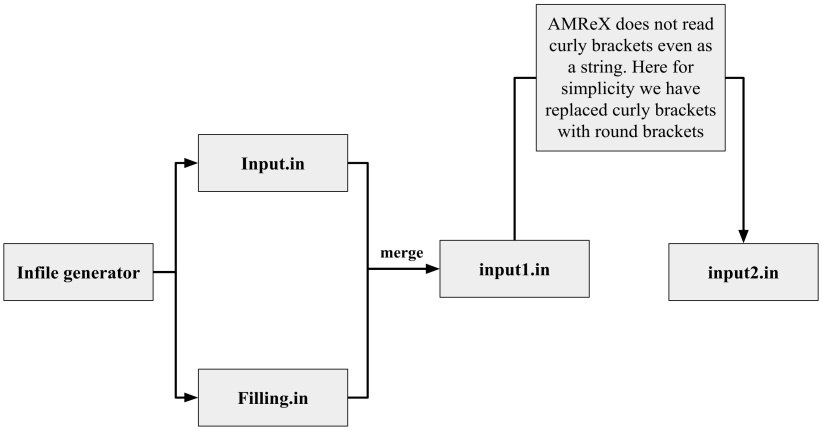

Figure 1: Defining Geometry in AMReX

3 Model formulation

The isothermal, two-phase, binary alloy, phase field solver is based on the Grand Potential model presented in *Phys. Rev. E 85, 021602 (2012).*

4 Diving deeper in AMReX

This section will focus on the methodology that AMReX follows for handling calculations.

4.1 GNUmakefile

The GNUmakefile is necessary to locate the path to AMReX and also provides vital information for compiling. One must make sure the AMREX_HOME

is located properly every time a GNUmakefle is written. GNUmakefile gives freedom to use a debugger(DEBUG), MPI for parallel processing(USE_MPI), choose a compiler(COMP), set precision(PRECISION), use CUDA for GPU (USE_CUDA), set the dimensionality(DIM), etc. Make.package, Make.defs and Make.rules are in the AMReX source folder and must be included. Locally generated Make.package must also be included. VPATH_LOCATIONS and INCLUDE_LOCATIONS are the search path for source and header files. To include additional libraries, write the following before calling Make.defs in the GNUmakefile.

```
INCLUDE_LOCATIONS += path/to/header
LIBRARY_LOCATIONS += path/to/libraries
LIBRARIES += -llib
```

A sample GNUmakefile looks as follows:

```
AMREX_HOME ?= ../../../../../amrex

DEBUG = FALSE
USE_MPI = FALSE
MPI_THREAD_MULTIPLE = FALSE
USE_OMP = FALSE
COMP = gcc
USE_CUDA = FALSE
DIM = 2

USE_CUDA = FALSE
USE_HIP = FALSE
USE_DPCPP = FALSE

include $(AMREX_HOME)/Tools/GNUMake/Make.defs

include ../Source/Make.package
VPATH_LOCATIONS += ../Source
INCLUDE_LOCATIONS += ../Source

include $(AMREX_HOME)/Src/Base/Make.package

include $(AMREX_HOME)/Tools/GNUMake/Make.rules
```

4.2 Reading inputs

The input file arguments are read in AMReX using the ParmParse class present in AMReX_ParmParse. When amrex::Initialize is called, the first command-line argument is taken to be the input file, and the information written in the input file is used to update the ParmParse database [1]. The data from 'input2.in' is read and processed for further use (Refer to Readinput.H).

4.3 Datatype in AMReX

AMReX supports all C++ datatypes. As we saw in 4.1, AMReX can be set to single or double precision(the default is double precision). Accordingly, the 'Real' datatype defined in AMReX_REAL.H changes to float or double, respectively.

4.4 Initialize and Finalize

amrex::Initialize and amrex::Finalize are the bounds of the code. The code must be written between amrex::Initilize and amrex::Finalize as many AMReX classes and functions will not function properly after amrex::Finalize.

4.5 Geometry, Box and DistributionMapping

```
BoxArray ba;
Geometry geom;
{   //Lower Boundary of the domain
    IntVect dom_lo(AMREX_D_DECL(0,0,0));
    //Upper Boundary of the domain
    IntVect dom_high(AMREX_D_DECL(ncellx-1,ncelly-1,ncellz-1));
    //Initialise the domain
    Box domain(dom_lo,dom_high);
    //Define the Box Array
    ba.define(domain);
    //Define partitions of the Box Array
    ba.maxSize(maxgrid);
    //Real size of the box
    RealBox real_box({AMREX_D_DECL(Real(0.0),Real(0.0),Real(0.0))},{AMREX_D_DECL(Real(ncellx*dx), Real(ncelly*dy), Real(ncellz*dz))});
    //Define the doamin, box, coordinate sytem and boundary condition of the geometry
    geom.define(domain, &real_box, CoordSys::cartesian, is_periodic.data());
}
```

Figure 2: Defining Geometry in AMReX

Geometry class is responsible for creating a domain and the coordinate system for the problem. Fig. 2 highlights the requirements for defining the geometry.

Given the problem's dimensionality and the lower and upper bounds of the domain, Box creates a discrete region within the bounds. Typically a box can be made cell-centred or node-centred in one or all directions. A new box with different indexing can also be defined using a present box. Multiple boxes in the domain can intersect, and a new box which shows the intersection space can also be created. The box is defined in the declaration of geometry. The lower can upper bounds of a box can be accessed using Dim3 or XDim3 structs.

BoxArray stores the collection of boxes (boxes obtained from domain decomposition) on a single level. The box is broken into multiple parts using the variable maxgrid. The boxes are divided so that no dimension of the divided box exceeds the maxgrid (refer to Fig. 3a). Boxes can also be partitioned in a non-uniform manner in both X and Y directions(refer to Fig. 3b). The grid size need not be an integral multiple of the maxgrid.

RealBox defines the real size of the physical box. Concerning fig. 2, the number of cells multiplied by the spatial discretization will give the real physical boundary.

DistributionMapping class distributes the partition boxes between processors. AMReX tries to distribute the load on each processor as equally as possible.

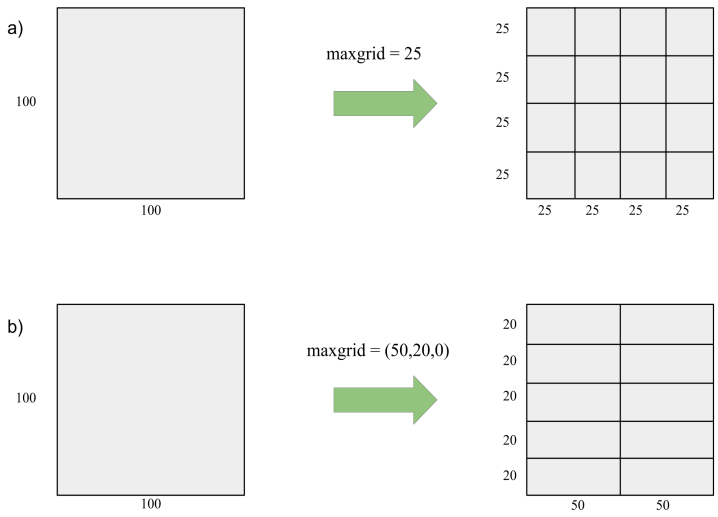

Figure 3: (a) Partitioning the box with less than or equal to 25 grid points in X and Y directions (b) Partitioning the box with less than or equal to 50 grid points in the X direction and less than or equal to 20 grid points in the Y direction

AMReX allows the user to distribute the workload according to their requirement (e.g. below). This can be achieved by passing a vector with a processor rank in the exact order in which AMReX stores the boxes (refer to fig. 4).

```
DistributionMapping dm; //empty object
Vector<int> pmap {...}; // Vector specifying process rank
dm.define(pmap); // dm defined based on pmap
```

4.6 BaseFab, FArrayBox, FabArray and MultiFabs

BaseFab is a class template for multi-dimensional array-like data structure on the Box. The dimensionality of the array is AMREX_SPACEDIM plus one. The additional dimension is for the number of components. Most applications do not use BaseFab directly but utilize the specialised classes derived from BaseFab.

FArrayBox(FAB) is a derived class of BaseFab. FArrayBox(FAB) is associated with the partitioned boxes. FabArray<FAB> is a class template in AMReX_FabArray.H for a collection of FABs on the same AMR level associated with a BoxArray. The grey box(fig. 5) is the FabArray that contains data of all the FABs on the same level. FabArray is distributed among parallel processes. Each process can operate only on the FAB objects owned by it, as the FabArray for each process contains only the local FAB data.

MultiFab is a commonly used FabArray class. A MultiFab contains several FABs defined on the Boxes and grown by the number of ghost cells. Ghost cells are essential for communicating with a neighbouring processor. As MultiFab contains ghost cells, its size is not the actual size of the Box. The original Box existing in the MultiFab is called a valid box. Ghost cell is a crucial feature of

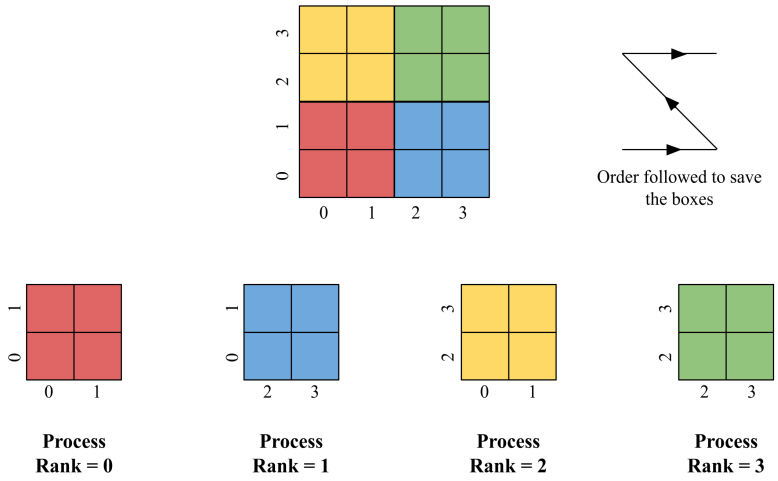

Figure 4: (a) Distribution mapping

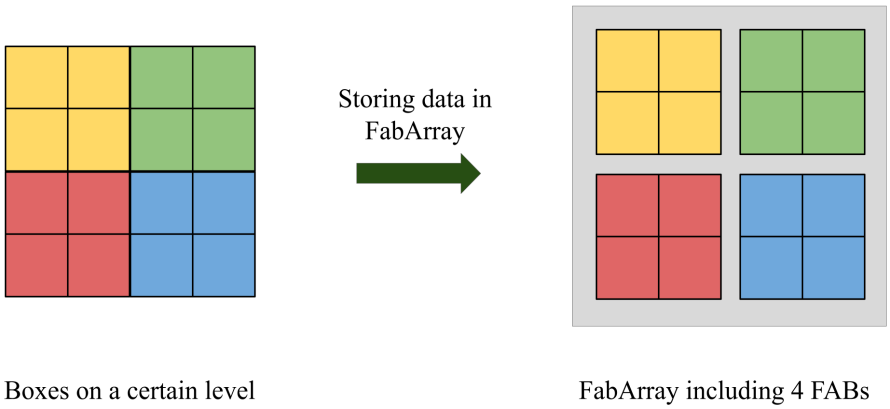

Figure 5: (a) FabArray and FABs

MultiFab, as FABs don't have a concept of ghost cells. In fig. 6a, the green box is the valid box, and the white cells are the ghost cells. The figure also points out the possibility of having a multi-component MultiFab, and one can access a specific component to operate on. Fig 6b shows the MFIter iteration space. In MFIter space, the valid box is coloured red and the neighbouring boxes till the blue box are the ghost cells. The number of times an MFIter loop runs is totally determined by the domain decomposition. A MultiFab can have multiple ghost layers. In this example, only one layer of ghost cells is shown. When two ghost layers are selected, the red box will remain the same in size, but the blue box will contain an additional layer of ghost cells.

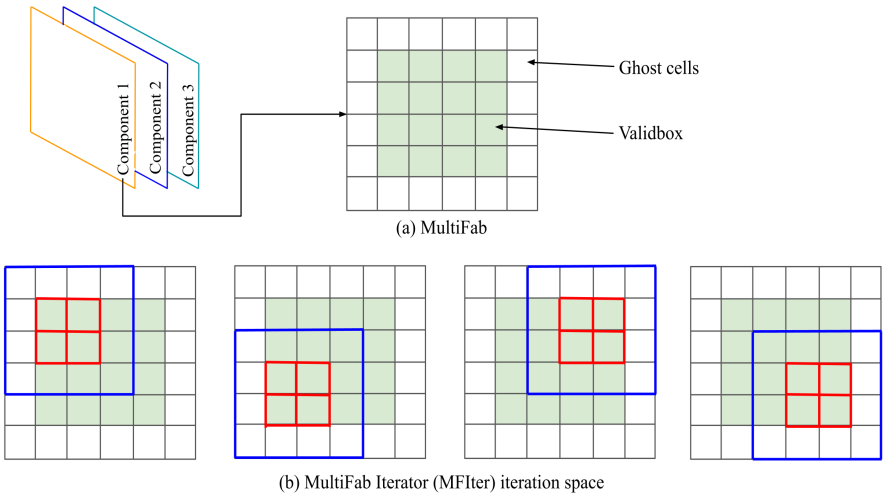

Figure 6: (a)MultiFab and (b)MFIter

4.7 Boundary condition

The ghost cell that is outside the domain can be thought of as either interior(Periodic) or physical (Neumann and Dirichlet conditions). The idea behind physical boundary is to create a BCRec object, which is essentially a multidimensional integer array of 2*DIM components. Each component defines a boundary condition type for the lo/hi side of the domain, for each direction. The amrex::BCType found in BCRec is used to set the type of the boundary condition. BCType::int_dir(interior), BCType::ext_dir(External Dirichlet, the user is responsible for filling the ghost cells for this condition), BCType::foextrap(First order extrapolation, Neumann condition), BCType::reflect_even (Reflection from interior cells with sign unchanged), BCType::reflect_odd (Reflection from interior cells with the sign changed). Refer to boundary_condition.H to see the implementation of the boundary condition. The implementation of the boundary condition on a MultiFab is shown in fig.7.

4.8 ParallelFor

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

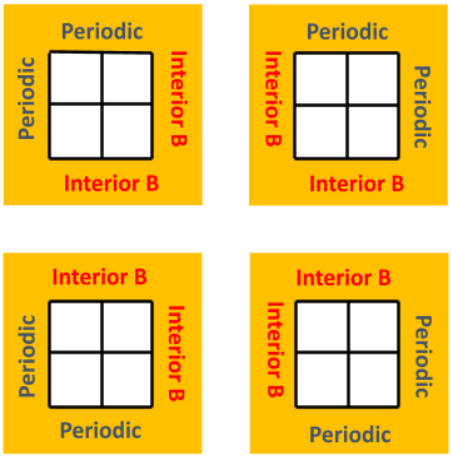

Figure 7: Boundary condition on MultiFab

ParallelFor is a for loop to iterate over a Box. In the example mentioned above, ParallelFor takes two arguments. The first argument is a Box specifying the iteration space, i.e. the valid box. The second argument takes the lambda function with indexes i,j and k that operate on the cell with the position index (i,j,k). AMREX_GPU_DEVICE is a macro that launches GPU kernel when built with USE_CUDA = TRUE. When USE_CUDA = FALSE, the GPU kernel is overlooked, and the loop runs on the CPU. ParallelFor can take up to four indices, one index for a 1D loop, two indices for a 2D loop, three indices for a 3D loop and four indices for a 4D loop. The fourth index in a 4D loop is the number of components.

5 Running AMReX codes

Refer to the Readme.md for installation and running of the codes.