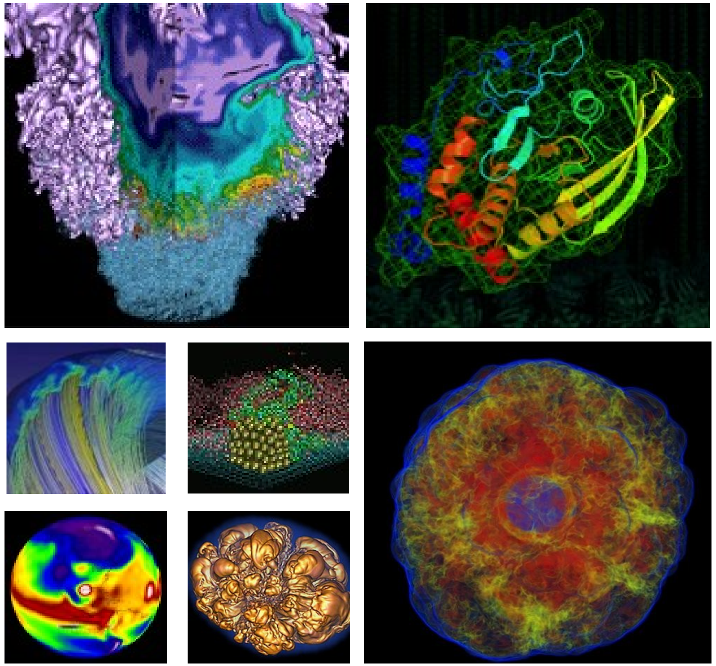
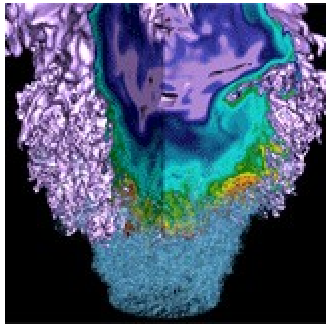
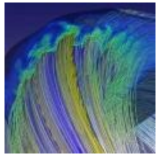
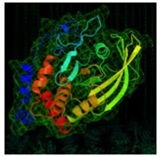
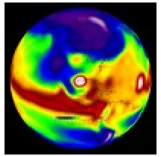
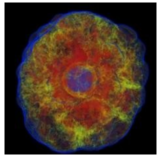
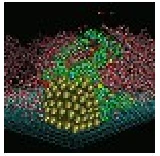
AMReX: Block Structured Adaptive Mesh Refinement



Don Willcox
Center for Computational Sciences and Engineering
CRD, LBL

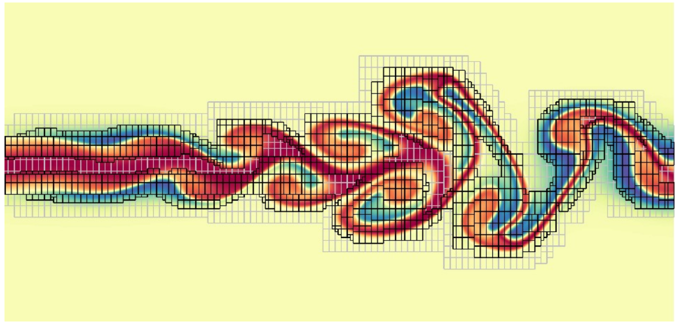
What is AMReX?

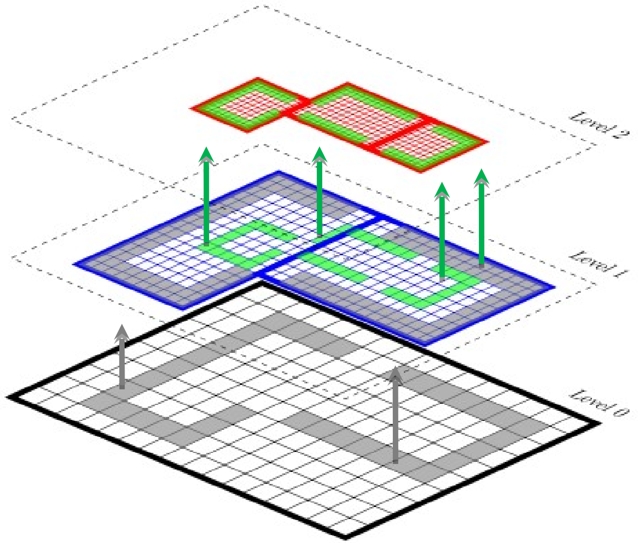
AMReX

AMReX is the (block-structured) AMR software framework being developed in the Co-Design Center.

Originally designed for solution of time-dependent PDEs but is not constrained to PDEs.



Much of the algorithmic methodology embedded in AMReX was developed as part of the DOE Applied Mathematics Program.



AMR appears in multiple applications

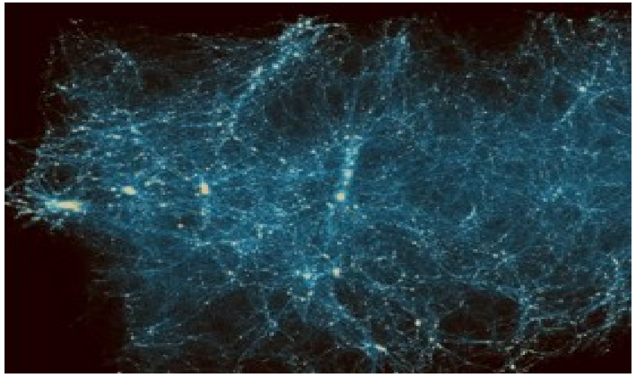
Some AMReX ECP Projects

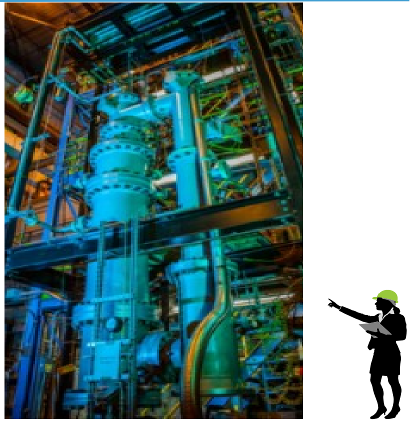
WarpX: Accelerator design

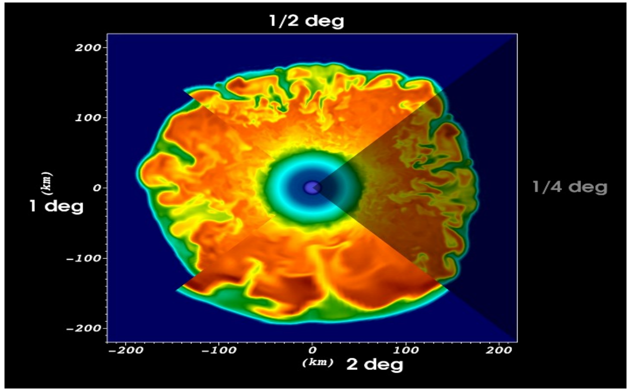
PeleLM: Combustion

FLASH5: Astrophysics

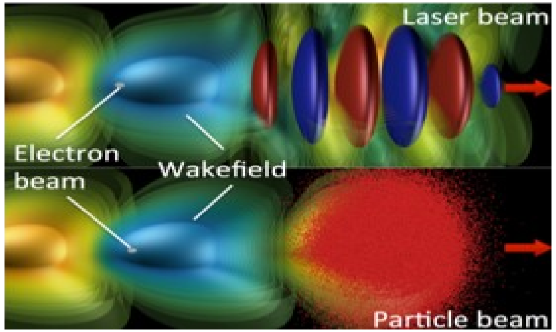
MFIX-Exa: Multiphase flow


Cosmology

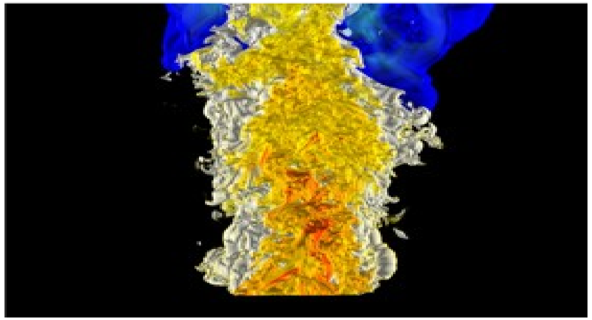

Multiphase flow



Astrophysics



Accelerators


Combustion

Overview of AMReX

- Written in a mix of **C++14/Fortran** (plus an option for using Fortran interfaces)
- Supports **parallelism through MPI, MPI+X** (where X can be OpenMP, OpenACC, and/or CUDA) and UPC++ is in progress.
- **Explicit & implicit** single- and multi-level mesh operations
- **Multilevel synchronization** operations
- **Particle and particle/mesh** algorithms
- Solution of parabolic and elliptic systems using **geometric multigrid solvers**
- **Embedded boundary** (cut-cell) representation of geometry
- Support for multiple **load balancing** strategies
- **Native I/O format** – supported by Visit, Paraview, yt

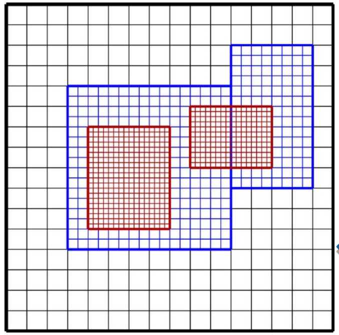
Flavors of (Adaptive) Mesh Refinement

Adaptive Mesh Refinement:

- refines mesh in regions of interest
- allows local regularity – accuracy, ease of discretization, easy data access
- naturally allows hierarchical parallelism
- uses special discretizations only at coarse/fine interfaces (co-dimension 1)
- requires only a small fraction of the book-keeping cost of unstructured grids

Example: AMReX

Example: FLASH

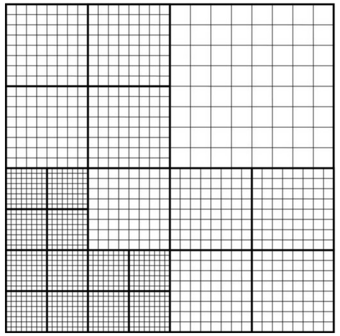


Grid sizes

May differ Same

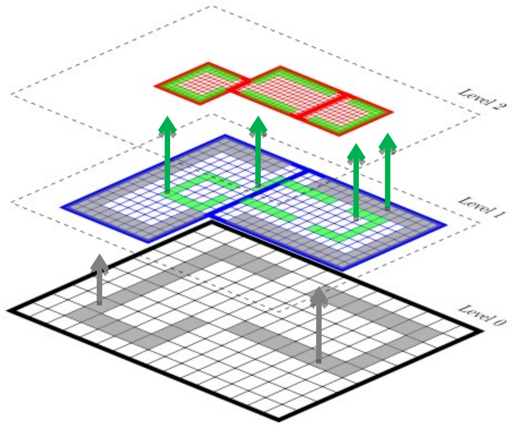
Child grid have unique parent?

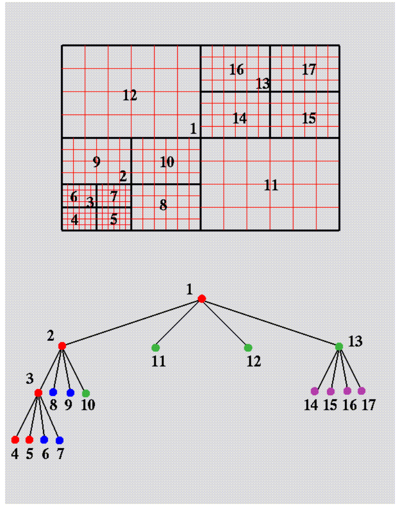
No Yes



https://iopscience.iop.org/article/10.1088/0067-0049/186/2/457)

Patch-Based vs OctTree





http://cucis.ece.northwestern.edu/ projects/DAMSEL/

Both styles of block-structured AMR break the domain into logically rectangular grids/patches. Level-based AMR organizes the grids by levels; quadtree/octree organizes the grids as leaves on the tree.

CPU Parallelism Strategy

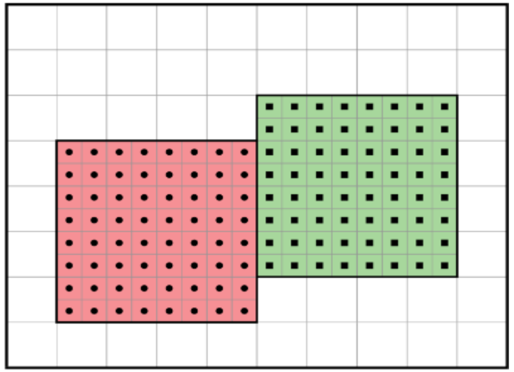
"MPI over grids, OpenMP over logical tiles"

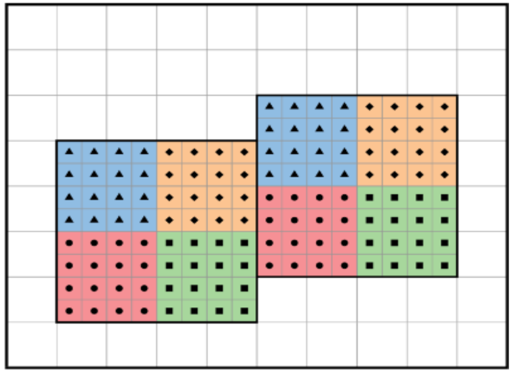
Done using MultiFAB iterators called **MFIter**:

- Handles proper looping **over local grids**.
- Stores relationship of grids across MPI ranks.
- Coordinates **tiling**.

Typical usage is for **OpenMP to loop over all the tiles** (potentially from multiple FABs) on a single MPI rank. Also includes:

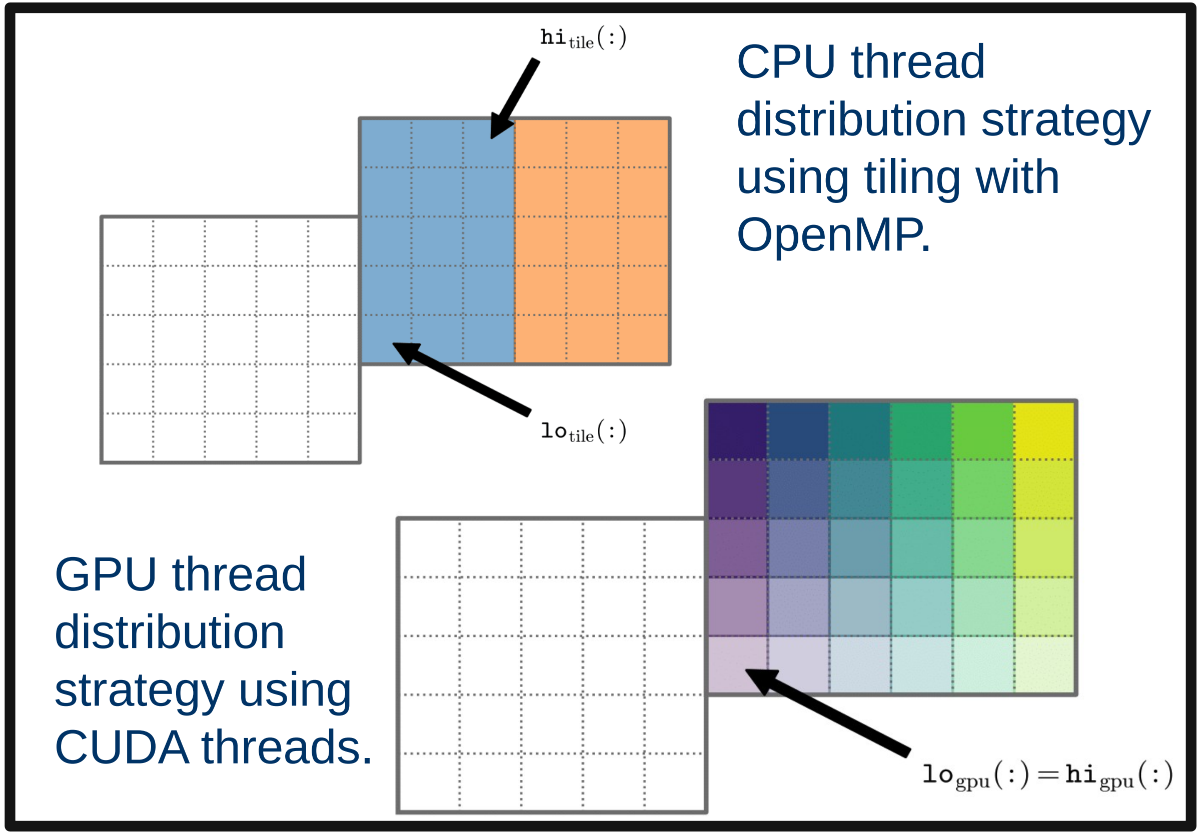
- Static vs. dynamic scheduling
- Synchronous vs. asynchronous
  (overlapping communication and computation)





GPU Parallelism is fine-grained

- **OpenMP threads** were on the order of tiles across **local boxes** (~10-100).
- GPU threads are on the order of local number of cells (~thousands).
- **GPU Parallelization** strategy is shifted to a finer-grained implementation **over cells**.


$\texttt{hi}_{\text{tile}}(:)$
$\texttt{lo}_{\text{tile}}(:)$
CPU thread distribution strategy using tiling with OpenMP.
GPU thread distribution strategy using CUDA threads.
$\texttt{lo}_{\text{gpu}}(:) = \texttt{hi}_{\text{gpu}}(:)$

Single Level Operations

Parallel Copy
– The most general parallel communication routine for mesh data
– Copies between MultiFabs that can have different BoxArrays and DistributionMappings
– Can take general “copy” operator - copy or add

Ghost cell operations
– FillBoundary - fills ghost cells from corresponding valid cells
– SumBoundary - adds from ghosts to corresponding valid

Neighbor particles / lists
– Each grid can grab copies of particles from other grids within a certain distance
– Can precompute list of potential collision partners over next N steps

Particle-mesh deposition / interpolation
– General version that can take user-defined lambda function specifying the kernel

Communication Between Levels

Interpolation:

- Filling ghost cells
- existing interpolaters for cell-, face- and node-centered data
- Default = factor of 2 but more general refinement ratios easy to add

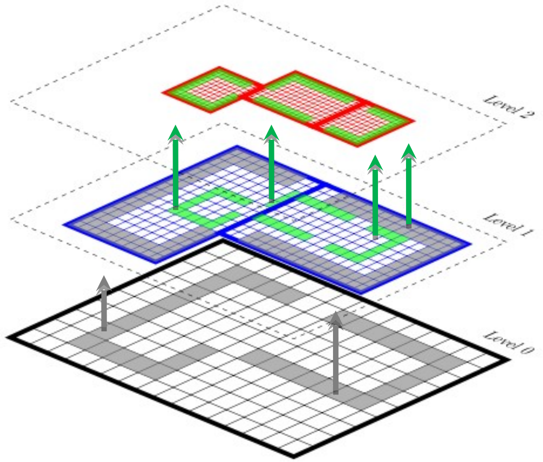
Restriction:

- Averaging fine onto coarse for synchronization
- existing restriction schemes for cell-, face- and node-centered data
- Default = factor of 2 but more general refinement ratios easy to add

Flux registers:

- Available in AmrCore/AmrLevel based applications – stores fluxes at coarse-fine interfaces for easy

Multi Level Operations



Regridding
– Tagging, grid construction, data filling …

Interpolation:
– Ghost cell filling and regridding

Restriction:
– Average fine onto coarse for synchronization

Flux registers:
– Available in AmrCore/AmrLevel based applications – stores fluxes at coarse-fine interfaces for easy refluxing

Virtual and Ghost Particle Construction
– For representing effect of particles at coarser / finer levels

Particle Redistribute
– Puts particles back on the right level / grid / tile after t**hey have moved**

Time-Stepping

AMR doesn’t dictate the spatial or temporal discretization on a single patch, but we need to make sure the data at all levels gets to the same time.

Subcycling in time means taking multiple time steps on finer levels relative to coarser levels.

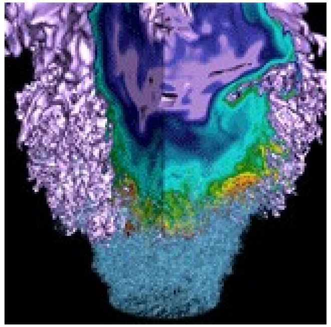
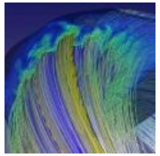
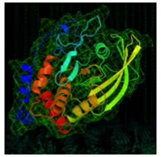
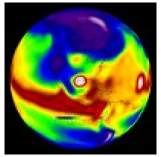
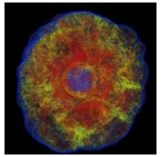
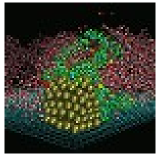
Non-subcycling:

- Same dt on every grid at every level
- Every operation can be done as a multi-level operation before proceeding to the next operation, e.g. if solving advection-diffusion-reaction system, we can complete the advection step on all grids at all levels before computing diffusion

Subycling:

- dt / dx usually kept constant
- Requires separation of “level advance” from “synchronization operations”
- Can make algorithms substantially more complicated

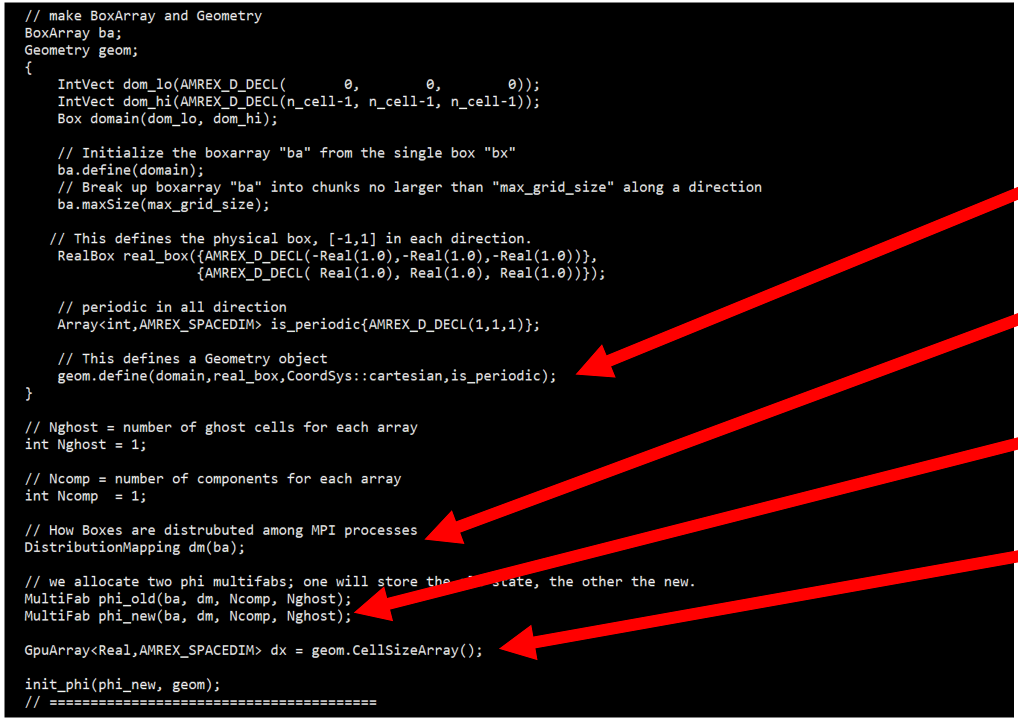
What Does a Simple AMReX Code Look Like?

AMReX Core Mesh Data Hierarchy

- **IntVect**
  - Dimension length array for indexing.
- **Box**
  - Rectilinear region of index space (using IntVects)
- **BoxArray**
  - Union of Boxes at a given level
- **FArrayBox (FAB)**
  - Data defined on a box (double, integer, complex, etc.)
  - Stored in column-major order (Fortran)
- **MultiFAB**
  - Collection of FArrayBoxes at a single level
  - Contains a Distribution Map defining the relationship across MPI Ranks.
  - Primary Data structure for AMReX mesh based work.

AMReX Core Mesh Data Hierarchy



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

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

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

    // periodic in all direction
    Array<int,AMREX_SPACEDIM> is_periodic{AMREX_D_DECL(1,1,1)};

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

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

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

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

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

GpuArray<Real,AMREX_SPACEDIM> dx = geom.CellSizeArray();

init_phi(phi_new, geom);
// ========================================
```

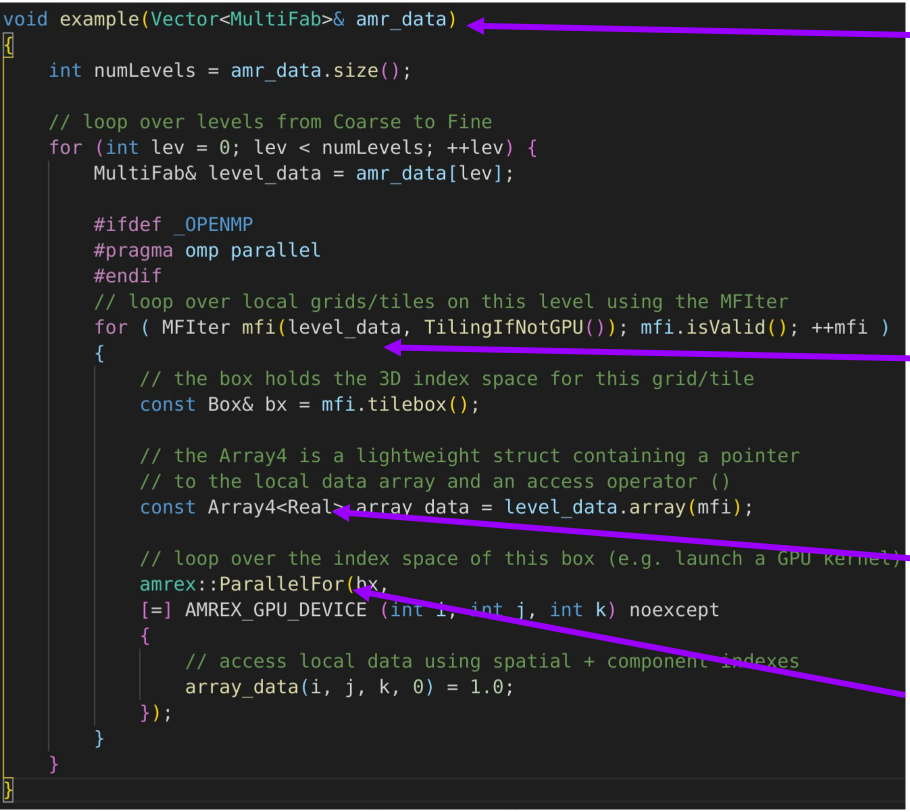
“Geometry” object carries information about domain size, periodicity, dx, etc

“DistributionMapping” = mapping of grids to MPI ranks

“MultiFab” holds solution data

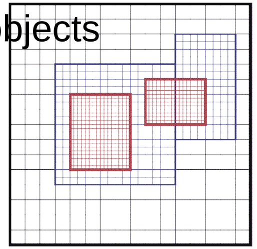
“dx” is array defined by “real_box[dir] / n_cell[dir]
no requirement that dx = dy = dz

How AMReX Loops over Mesh Data

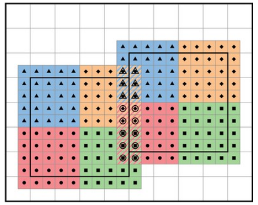


```
void example(Vector<MultiFab>& amr_data)
{
    int numLevels = amr_data.size();

    // loop over levels from Coarse to Fine
    for (int lev = 0; lev < numLevels; ++lev) {
        MultiFab& level_data = amr_data[lev];

        #ifdef _OPENMP
        #pragma omp parallel
        #endif
        // loop over local grids/tiles on this level using the MFIter
        for ( MFIter mfi(level_data, TilingIfNotGPU()); mfi.isValid(); ++mfi )
        {
            // the box holds the 3D index space for this grid/tile
            const Box& bx = mfi.tilebox();

            // the Array4 is a lightweight struct containing a pointer
            // to the local data array and an access operator ()
            const Array4<Real> array_data = level_data.array(mfi);

            // loop over the index space of this box (e.g. launch a GPU kernel)
            amrex::ParallelFor(bx,
            [=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
            {
                // access local data using spatial + component indexes
                array_data(i, j, k, 0) = 1.0;
            });
        }
    }
}
```

- MPI-distributed data at all levels stored as a vector of AMReX MultiFab objects

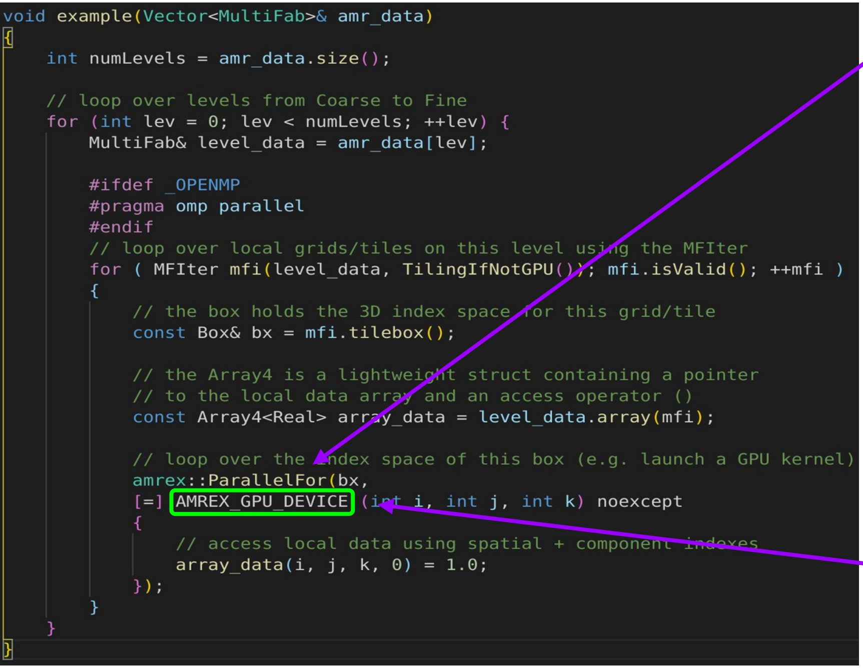


- Each **MultiFab** contains **pointers to local grid data** for one MPI rank + **grid distribution** metadata



- Loop over local data with the **M**ulti**F**ab **Iter**ator. For OpenMP, generate logical tiles for local grids.
- The **Array4** contains a pointer and access operator(). The lambda captures it by value.
- The **ParallelFor** takes index space in a box and a C++ lambda function to call on each 3D index

How AMReX Loops over Mesh Data (GPUs)



```
void example(Vector<MultiFab>& amr_data)
{
    int numLevels = amr_data.size();

    // loop over levels from Coarse to Fine
    for (int lev = 0; lev < numLevels; ++lev) {
        MultiFab& level_data = amr_data[lev];

        #ifdef _OPENMP
        #pragma omp parallel
        #endif
        // loop over local grids/tiles on this level using the MFIter
        for ( MFIter mfi(level_data, TilingIfNotGPU()); mfi.isValid(); ++mfi )
        {
            // the box holds the 3D index space for this grid/tile
            const Box& bx = mfi.tilebox();

            // the Array4 is a lightweight struct containing a pointer
            // to the local data array and an access operator ()
            const Array4<Real> array_data = level_data.array(mfi);

            // loop over the index space of this box (e.g. launch a GPU kernel)
            amrex::ParallelFor(bx,
            [=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
            {
                // access local data using spatial + component indexes
                array_data(i, j, k, 0) = 1.0;
            });
        }
    }
}
```

- The **ParallelFor** takes index space in a box and a C++ lambda function to call on each 3D index.
- The **Array4** contains a pointer and access operator(). Captured by value in the lambda.
- AMReX memory arena uses **CUDA Unified Memory**
- AMReX **ParallelFor** launches CUDA kernel
- All we had to do was label our “work” lambda function as a GPU function

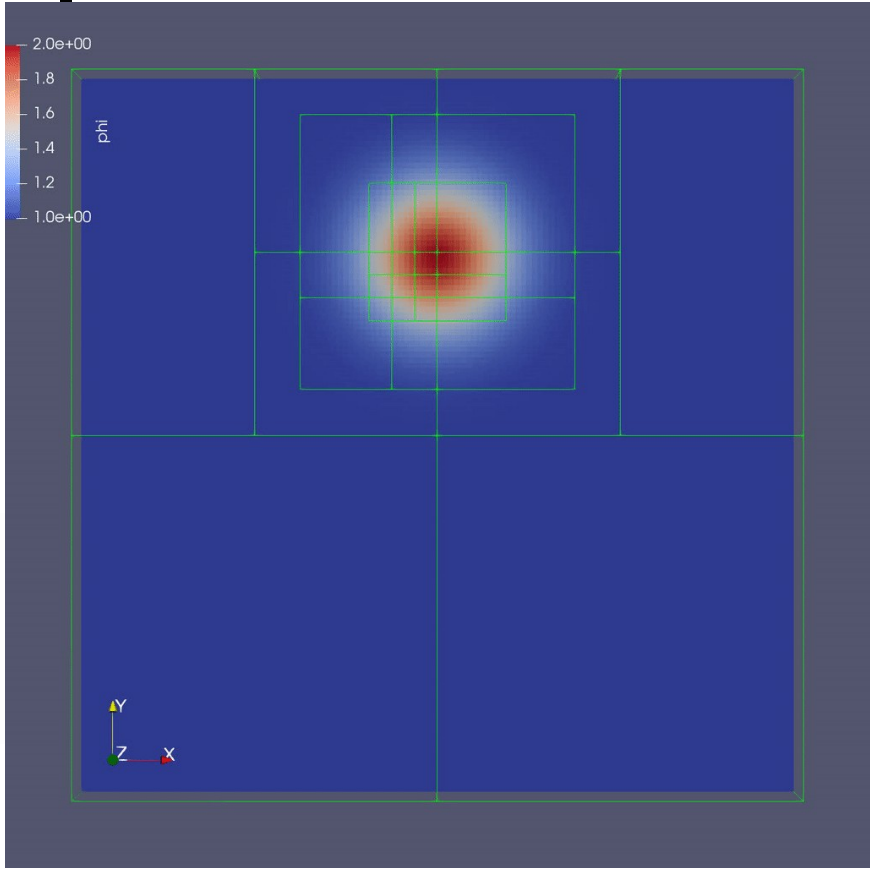
A Simple AMR Example: Scalar Advection

We want to advect a blob of dye in a fluid clockwise, then counterclockwise to return to the starting position.

We also want to refine on the density of the dye in the fluid.

The mesh refinement is adaptive, i.e. we compute a new set of cells to refine and a new set of grids every N timesteps.

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}^{\mathbf{spec}} \phi) = \mathbf{0}$$



Particle-Mesh + GMG Example

We want to advect a blob of dye in a fluid clockwise, but this time we set the velocity field by solving the incompressible flow equation.

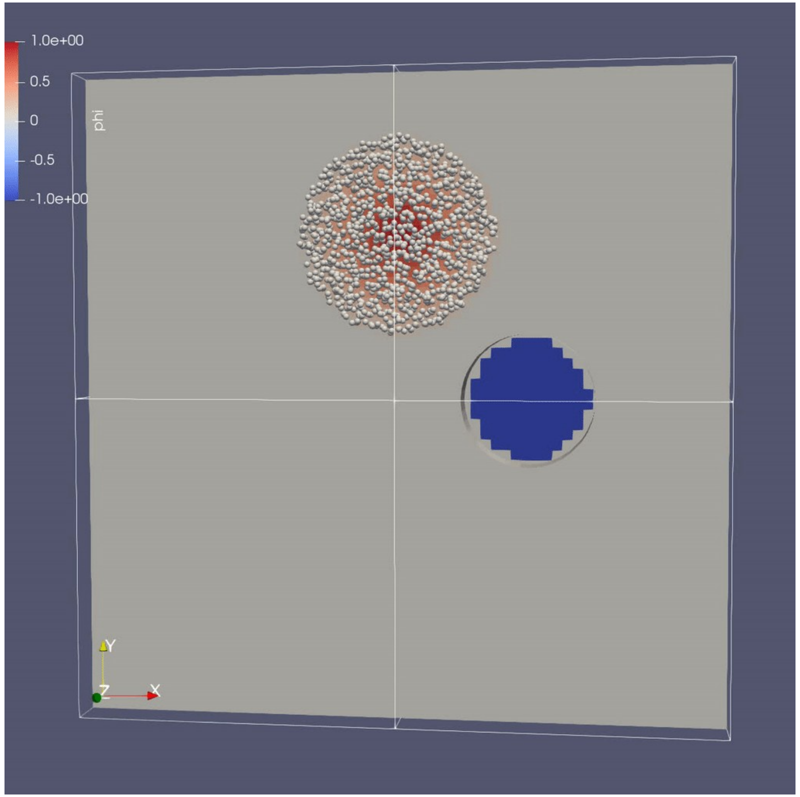
We use a linear solver to solve for the incompressible velocity field.

$$\nabla \cdot (\beta \nabla \xi) = \nabla \cdot \mathbf{u}^{\mathbf{spec}}$$

We represent the density of the dye using particles with computational weights.

$$\phi(i,j,k) = \frac{1}{dxdydz} \cdot \sum_p S_x \cdot S_y \cdot S_z \cdot w_p$$

$$\phi_p = \sum_i \sum_j \sum_k S_x \cdot S_y \cdot S_z \cdot \phi(i,j,k)$$



For More Information About AMReX ...

software: https://www.github.com/AMReX-Codes/amrex

documentation: https://amrex-codes.github.io/amrex

movies based on AMReX codes: https://amrex-codes.github.io/amrex/gallery.html