AMReX: Building a Block-Structured AMR Application

Presented at the Argonne Training Program on Extreme-Scale Computing (ATPESC) 2026

ATPESC Numerical Software Track

Presented by Andrew Myers and Weiqun Zhang, LBNL
based on slides from Ann Almgren

On behalf of the AMReX team (~203 contributors)

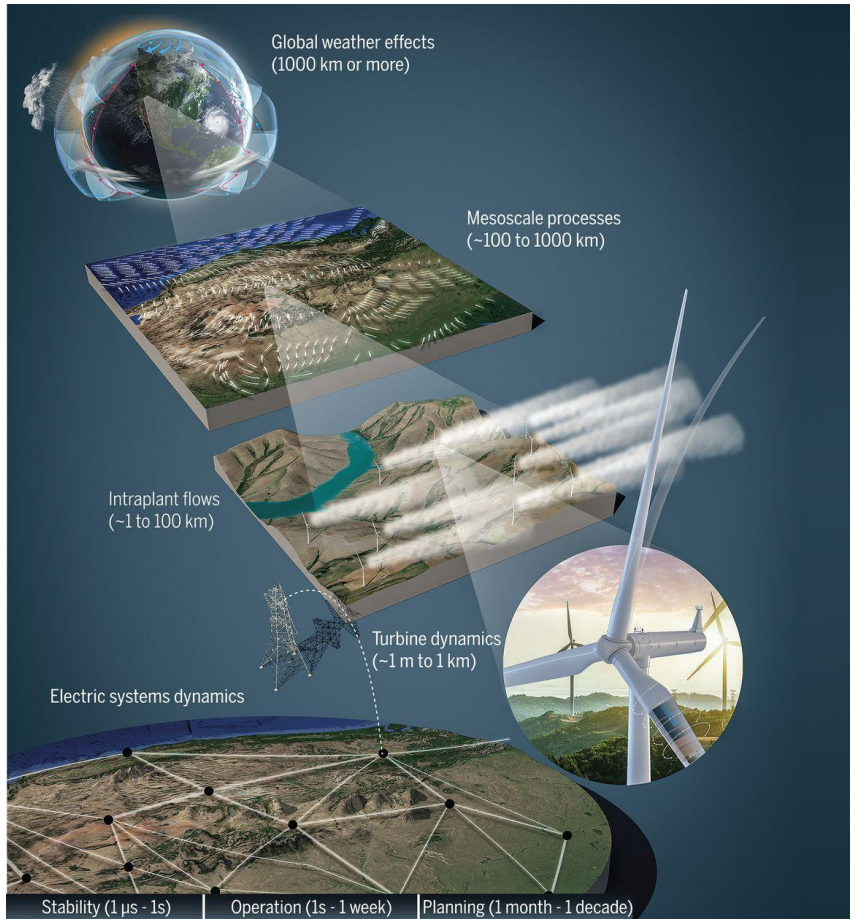
Setting the Stage

Most of the problems we solve today are hard.

Characteristics of these problems are
often that they couple multiple physical processes
across a range of spatial and temporal scales.

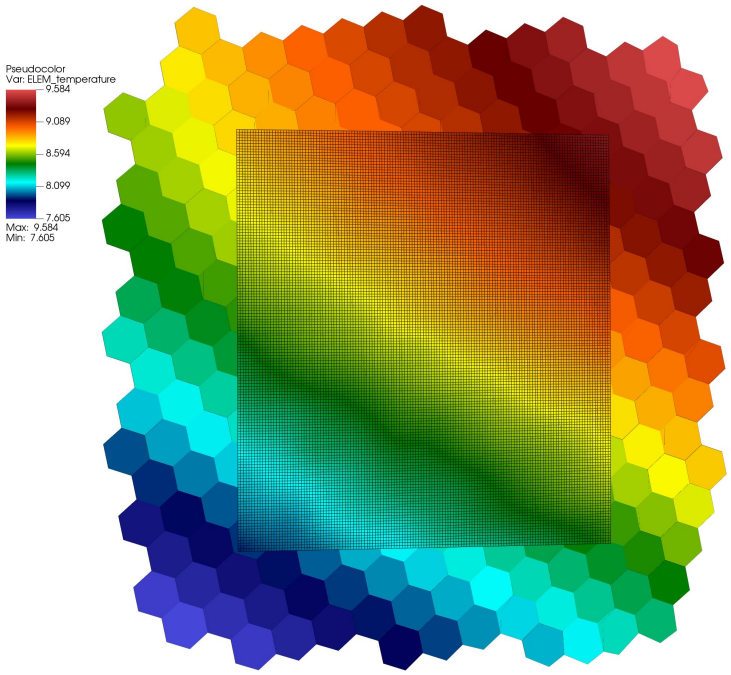
Gone are the days of simple physics, simple geometry, single algorithm, homogeneous architectures … ☹

So how do we build algorithms and software for hard multi-physics multi-scale multi-rate problems without starting over every time?



Grand challenges in the science of wind energy
Veers, P. et al., 2019 *Science,* Vol. 366, Issue 6464
https://science.sciencemag.org/content/366/6464/eaau2027

Setting the Stage (p2)



Not all simulations use a mesh

But for those that do, the choice is usually **structured** vs **unstructured.**

Structured:

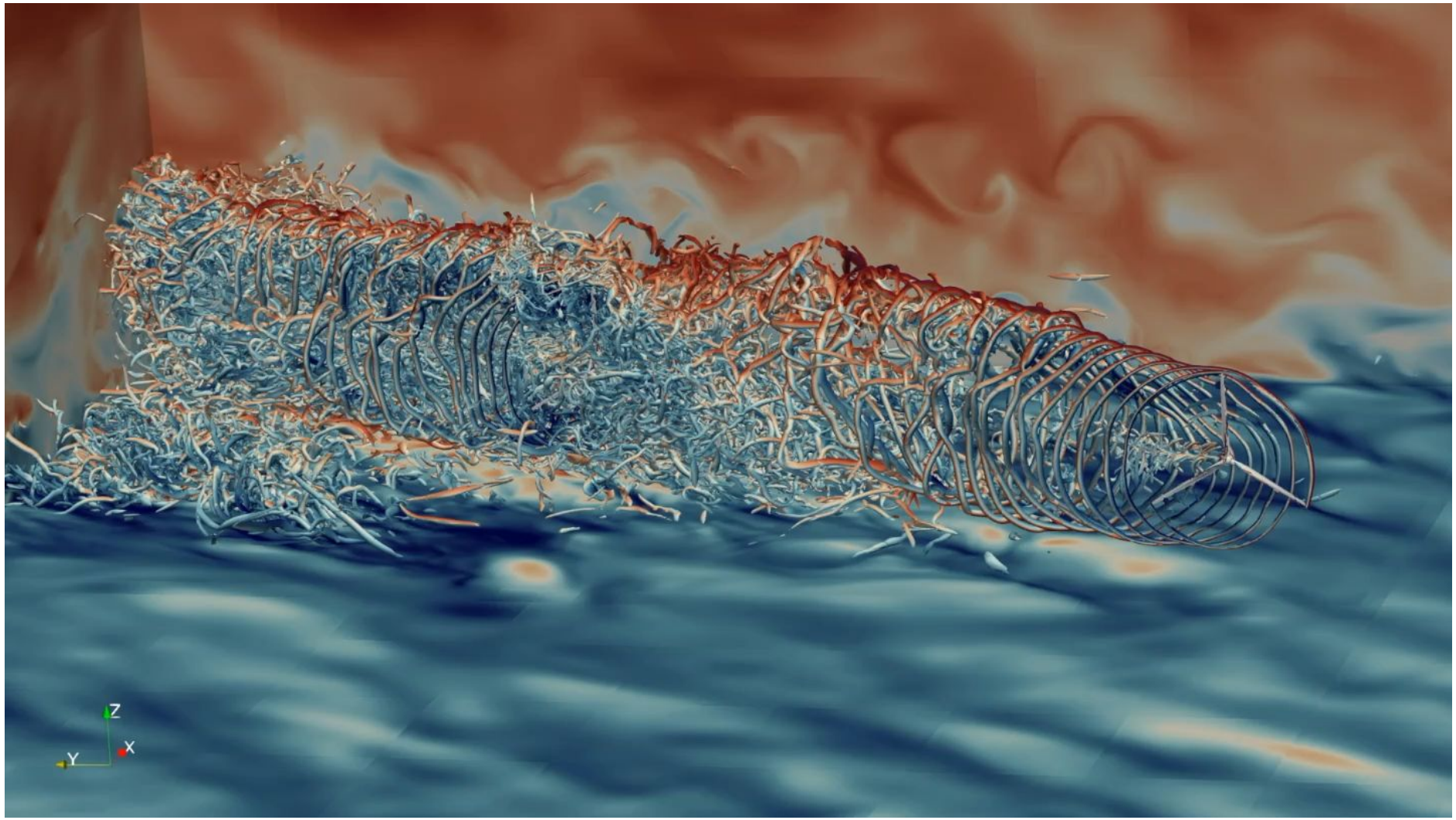
- Easier to write discretizations
- Simple data access patterns
- Extra order of accuracy due to cancellation of error
- Easy to generate complex boundaries through cut cells but hard to maintain accuracy at boundaries

Unstructured:

- Can fit the mesh to any geometry – much more generality
- No loss of accuracy at domain boundaries
- More “book-keeping” for connectivity information, etc
- Geometry generation becomes time-consuming

Both structured and unstructured meshes can be **dynamically adaptive**.

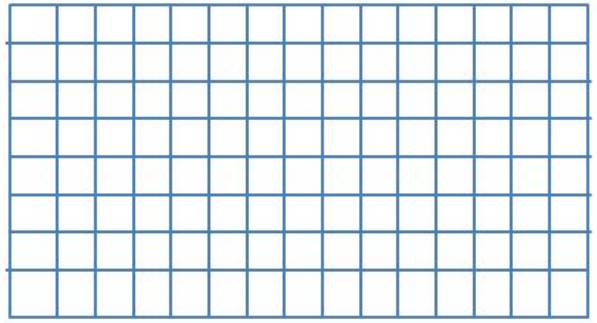
Structured vs Unstructured?



ExaWind simulation of two NREL 5-MW turbines in turbulent atmospheric flow.

Image credit: Brazell, Brunhart-Lupo, Henry de Frahan, Rood, Sharma, Vijayakumar, et al.

Structured Grid Options

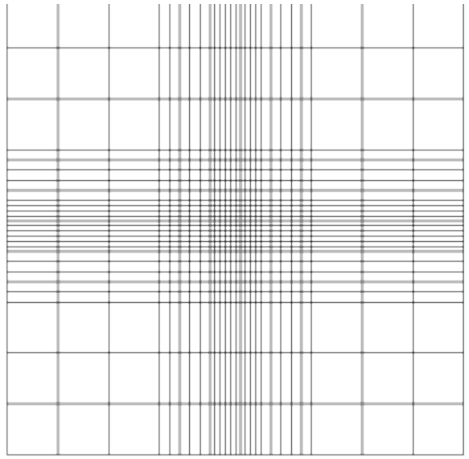


https://commons.wikimedia.org/wiki

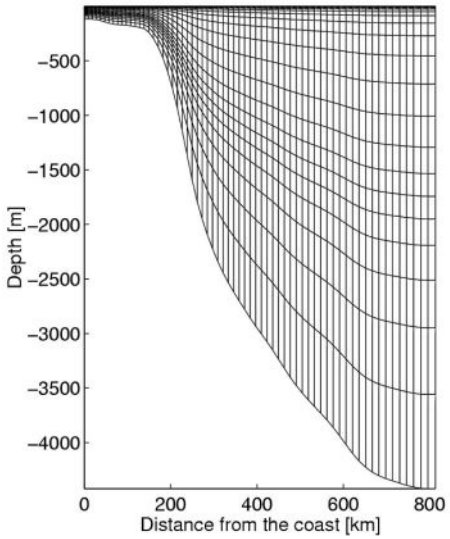
Logically rectangular doesn't mean physically rectangular

Structured with non-constant cells split pros and cons of structured vs unstructured:

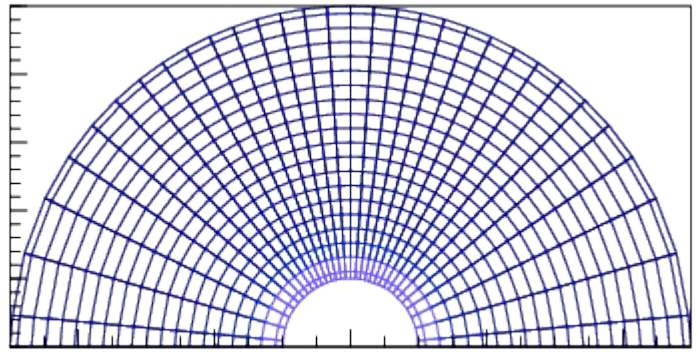
- Can fit (simple) non-rectangular boundaries while still having known connectivity
- Finer in certain regions (mesh refinement)
- Harder to maintain accuracy



http://silas.psfc.mit.edu/22.15/lectures/chap4.xml

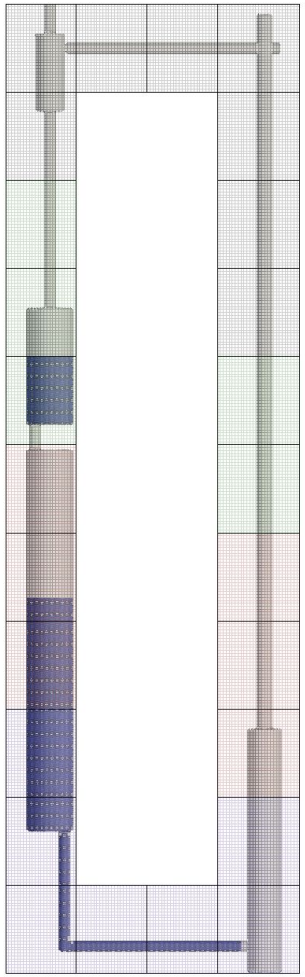


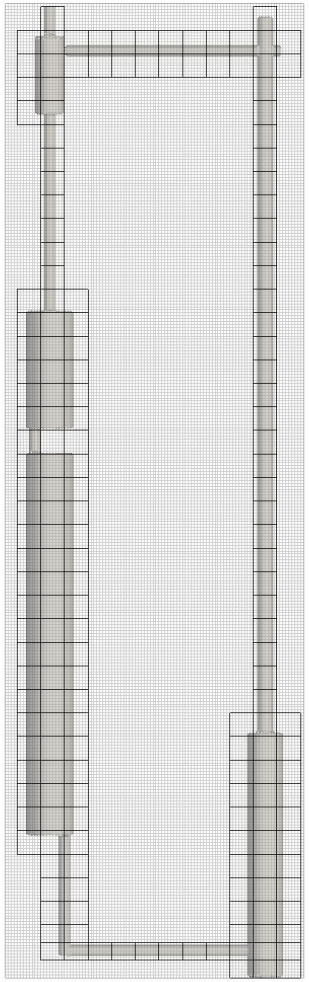
http://www.cfoo.co.za/simocean/modelsroms.php



http://silas.psfc.mit.edu/22.15/lectures/chap4.xml

More Structured Grid Options





Structured grid does not have to mean the entire domain is logically rectangular either.

One can also “prune” the grids so as to not waste memory or MPI ranks – can still use rectangular cells in non-rectangular domain.

Grid pruning can save both memory and work.

Why Is Uniform Cell Size Good?

Numerical Analysis 101:

$$\phi_{i+1} = \phi(x_i) + \Delta x_r \phi_x + 1/2(\Delta x_r)^2 \phi_{xx} + O(\Delta x_r{}^3)$$
$$\phi_{i-1} = \phi(x_i) - \Delta x_l \phi_x + 1/2(\Delta x_l)^2 \phi_{xx} + O(\Delta x_l{}^3)$$

We often use a centered difference as an approximation for a gradient,

$$\frac{\phi_{i+1} - \phi_{i-1}}{\Delta x_l + \Delta x_r} = \phi_x + 1/2(\Delta x_r - \Delta x_l)\phi_{xx} + O(\Delta x^2)$$

Note we only get second-order accuracy if we use constant cell spacing.

Can we confine this error?

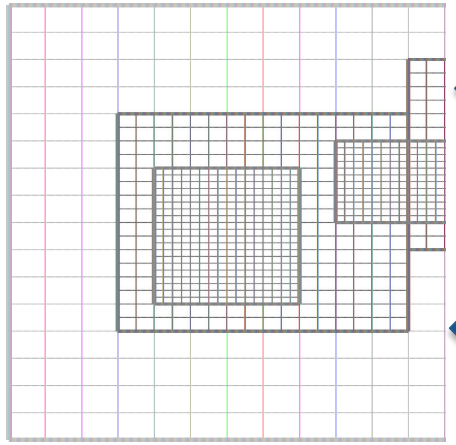
Can We Have the Best Of Both Worlds?

Distorting the mesh is not ideal, but we can't afford uniformly fine grid.

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



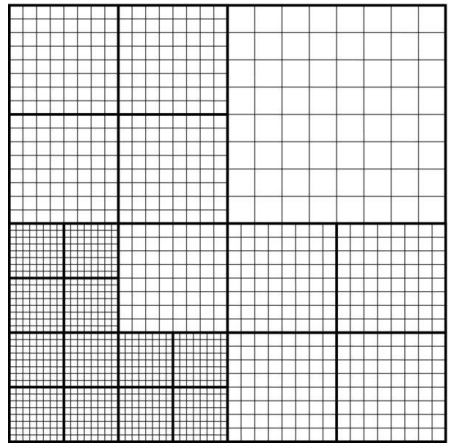
Grid sizes

May differ ← → Same

Child grid have unique parent?

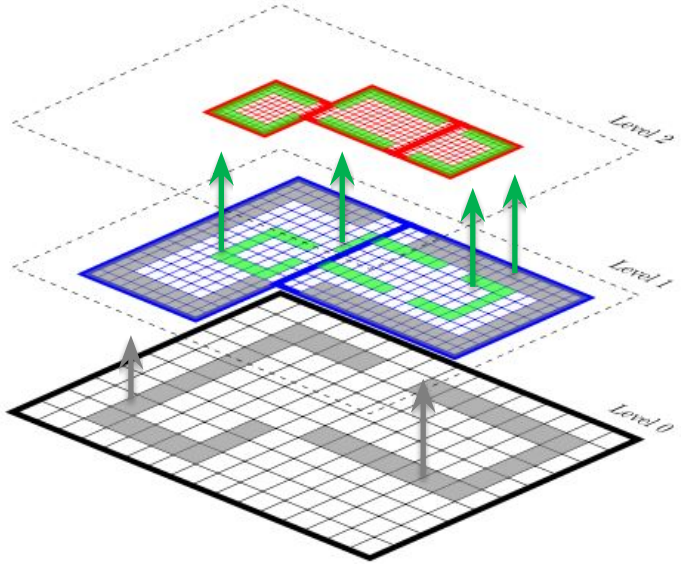
No ← → Yes

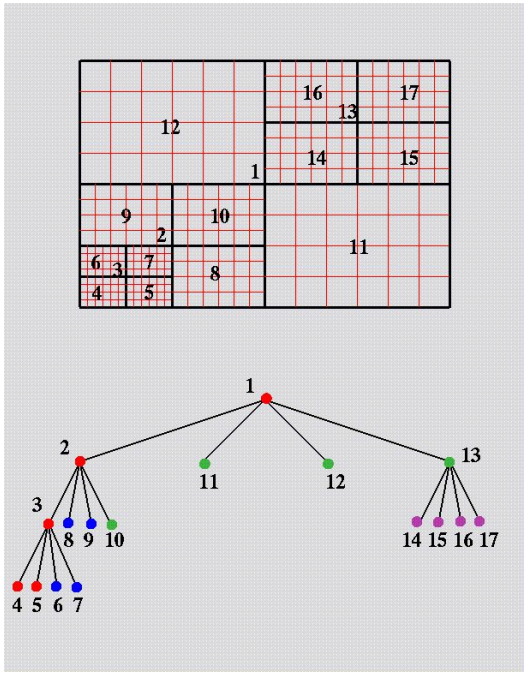
Example:
FLASH



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.

What is an “AMR algorithm”?

Key things you need to know if you want to use AMR:

- How to advance the solution one patch at a time
- What the right matching conditions are at coarse/fine interfaces
    - This depends very much on the type of equation, e.g. hyperbolic vs elliptic, and what features of the solution are important (e.g., is conservation important?)
    - How you solve on the patch (e.g. with implicit vs explicit update) affects how you synchronize between levels

What about 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.

The main question is:

To subcycle or not to subcycle?

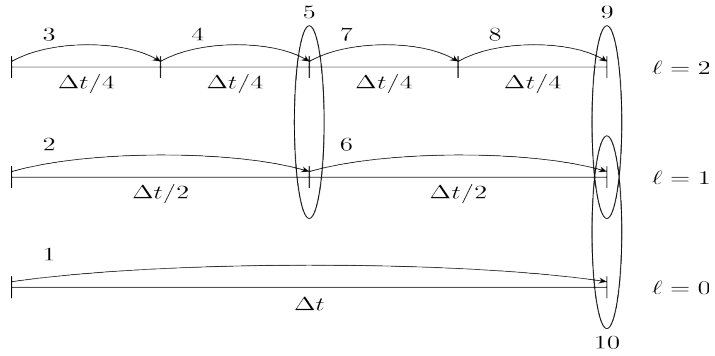
Subcycling in time means taking multiple time steps on finer levels relative to coarser levels.

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" – but this can often feel more intuitive
- Can make algorithms substantially more complicated



AMReX Hands-On Examples

Let's do a few hands-on exercises that demonstrate AMReX capabilities

"**AMR 101**": AMR for scalar advection

- Multilevel mesh data – fluid velocity on faces and tracer on cell centers
- Dynamic AMR
- Strong scaling behavior
- Performance portability (CPU vs GPU)

Instructions for how to access and run the code are on the web page:

https://xsdk-project.github.io/MathPackagesTraining2026/lessons/amrex/

Source code for these examples is here:

https://github.com/AMReX-Codes/ATPESC-codes

Scalar Advection: Governing Equations

We demonstrate building an AMR application with a simple example of a hyperbolic conservation law: the scalar advection equation:

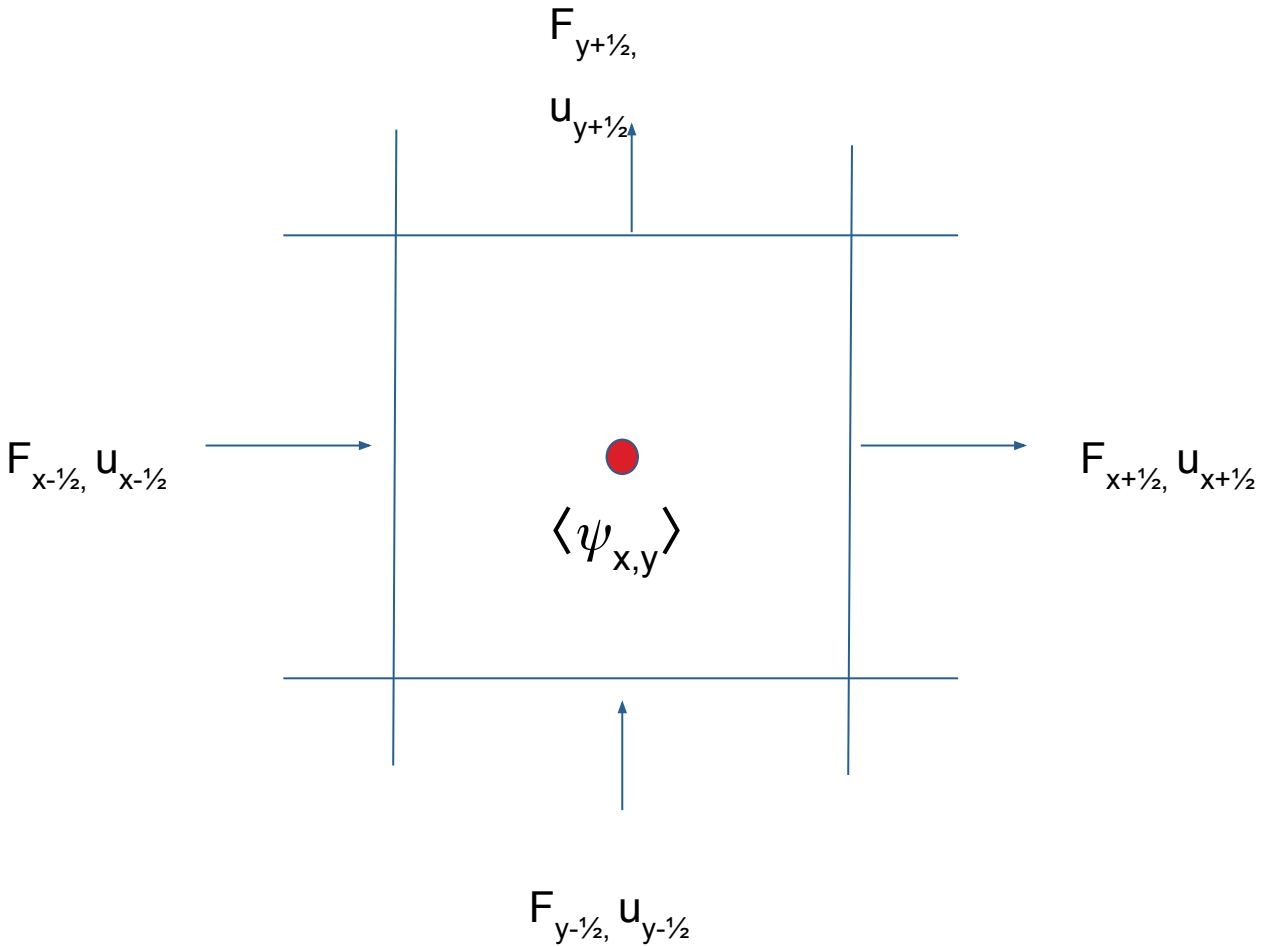
$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}^{\mathbf{spec}} \phi) = \mathbf{0}$$

Here, ϕ represents a passive scalar (think, concentration of a dye in a fluid) and u^{spec} is a specified velocity field. This is a continuity equation that represents mass conservation. If we take u^{spec} to be given by the curl of a scalar stream function ψ, it is automatically divergence free (incompressible):

$$\psi(i,j) = \sin^2(\pi x)\sin^2(\pi y)\cos(\pi t/2)/\pi$$

$$u = -\frac{\partial \psi}{\partial y}, v = \frac{\partial \psi}{\partial x}.$$

Scalar Advection: Numerical Algorithm (single level)



Finite Volume discretization
Godunov-type conservative update:

$$\langle \psi_{x,y} \rangle^{n+1} = \langle \psi_{x,y} \rangle^{n} + \left(F_{x-½,} - F_{x+½,}\right) dt\, dx + \left(F_{y-½,} - F_{y+½,}\right) dt\, dy$$

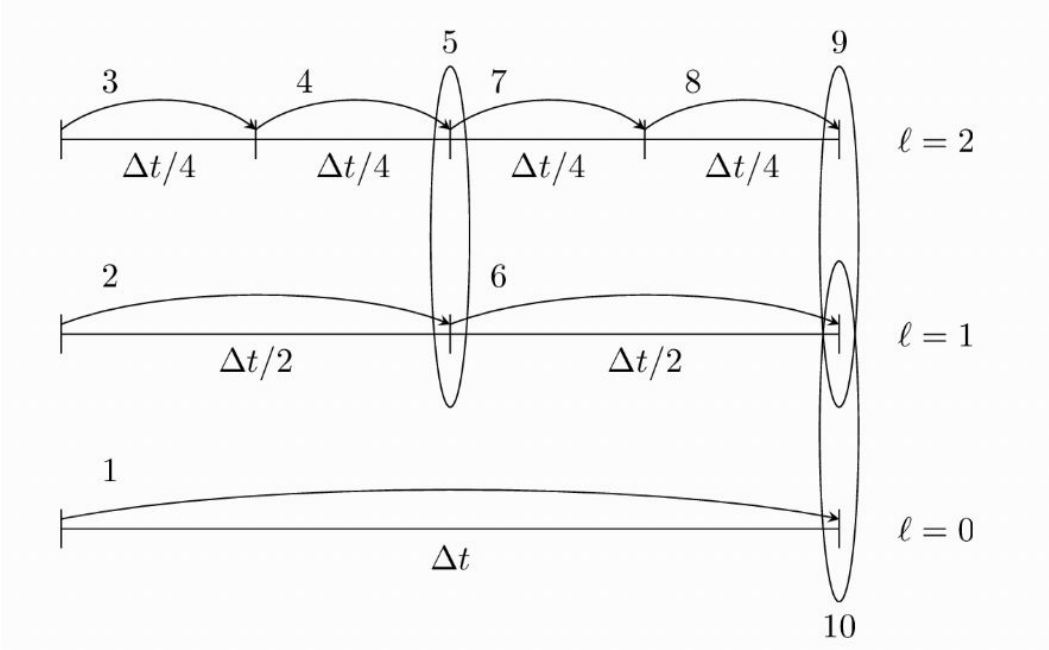
Details of computing F:

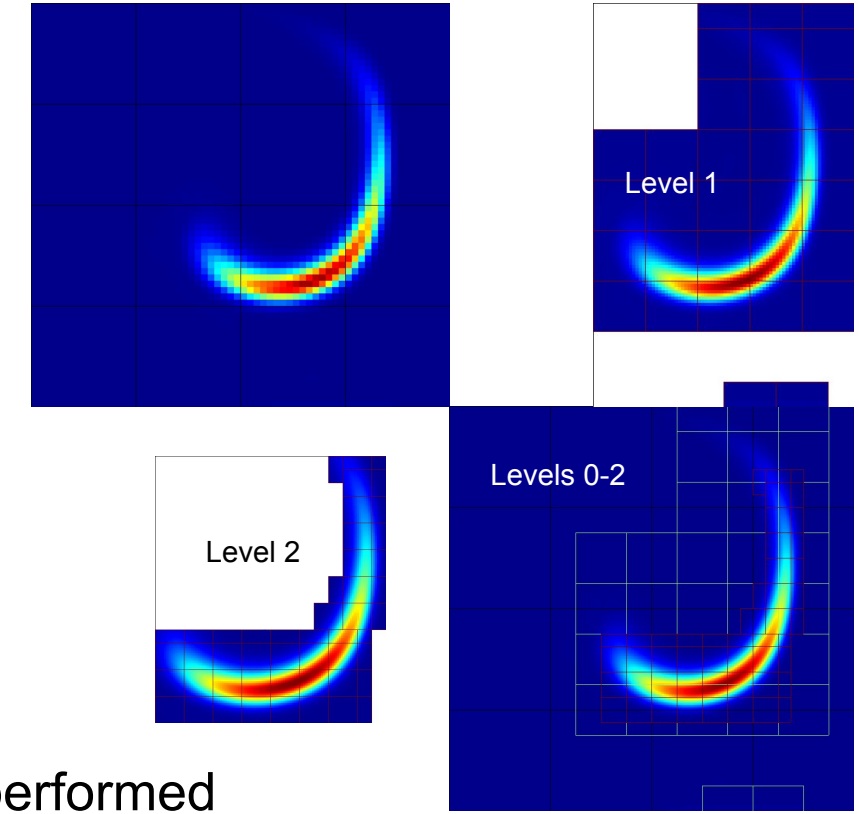
First compute 2nd order, limited slopes with minmod limiter, then 4th order correction

Predict Fx, Fy separately using upwinding

Then, apply transverse corrections, making the overall method unsplit

Mesh refinement (with sub-cycling)





Circles indicate the point at which synchronization needs to be performed

Flux registers store (time- and area- weight) flux mismatches at coarse fine boundaries. The coarse solution needs to be corrected by the divergence of this mismatch. Then, solution can be averaged down.

Inputs

```
stop_time          = 2.0                # the final time (if we have not exceeded number of steps)
max_step           = 100000              # the maximum number of steps (if we have not exceeded stop_time)

amr.n_cell         = 64 64 8             # number of cells at the coarsest AMR level in each coordinate direction

amr.max_grid_size  = 16                  # the maximum number of cells in any direction in a single grid

amr.plot_int       = 10                  # frequency of writing plotfiles

adv.cfl            = 0.9                 # cfl number to be used for computing the time step

adv.phierr = 1.01  1.1  1.5              # regridding criteria  at each level
```

In addition to max_grid_size, there is also the blocking_factor (grids must be divisible by this)

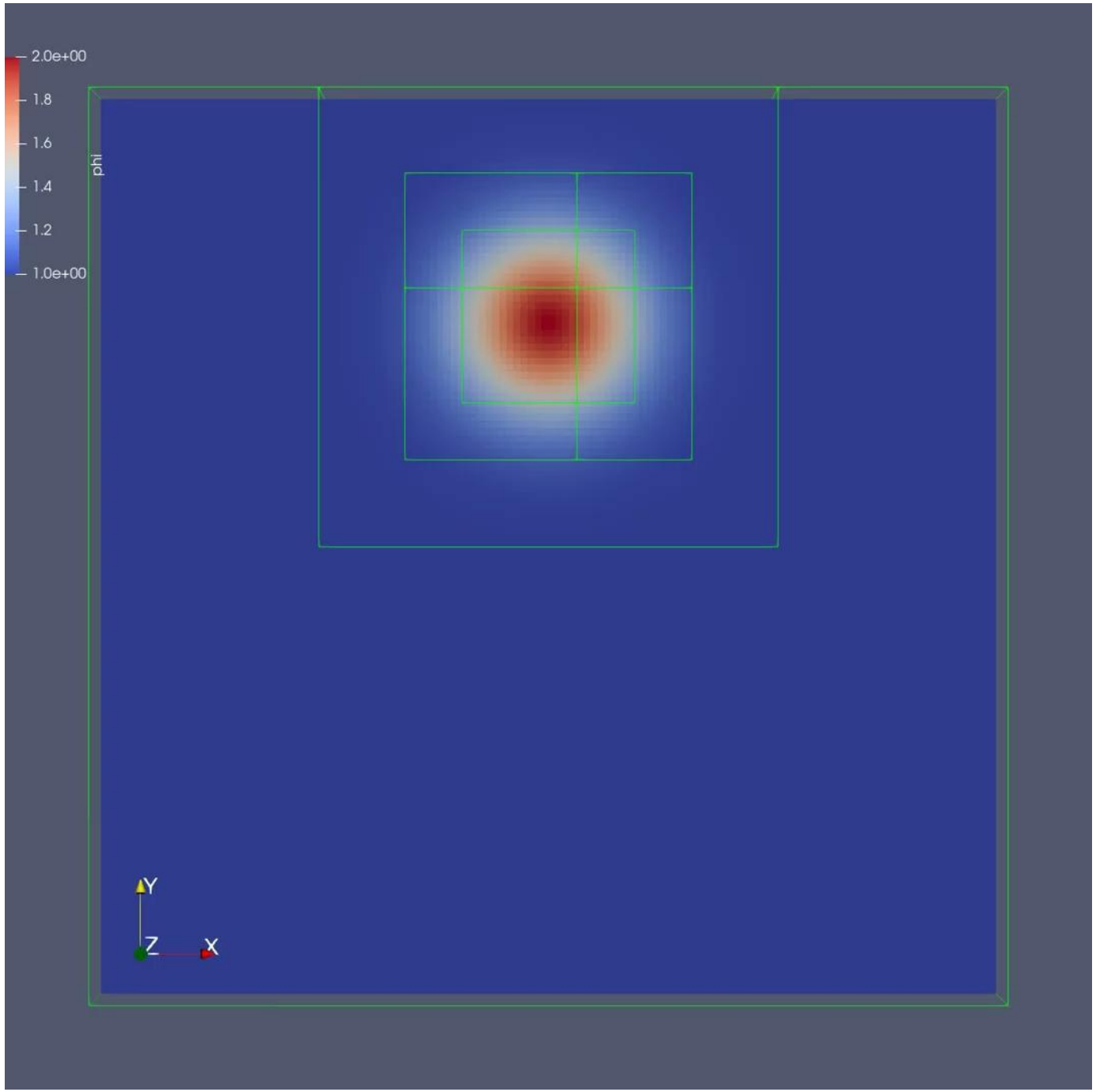
Tagging / regridding

Creating grids at levels > 0 involves tagging cells based on a user provided function.

In this example, just a threshold. But can use gradients, more complicated logical expressions, etc…

Fine grids are created using the Berger-Rigoutsos clustering algorithm with the additional constraints of satisfying the blocking_factor and max_grid_size criteria.

Other parameters: amr.grid_eff, amr.n_error_buf.



Does the code conserve phi as expected?

What happens if you turn off refluxing and/or subcyling?

Add more levels, adjust refinement criteria.

Try the GPU-enabled executable and compare the runtime to the CPU-only code.

Try multiple MPI ranks

Performance Portability / GPUS

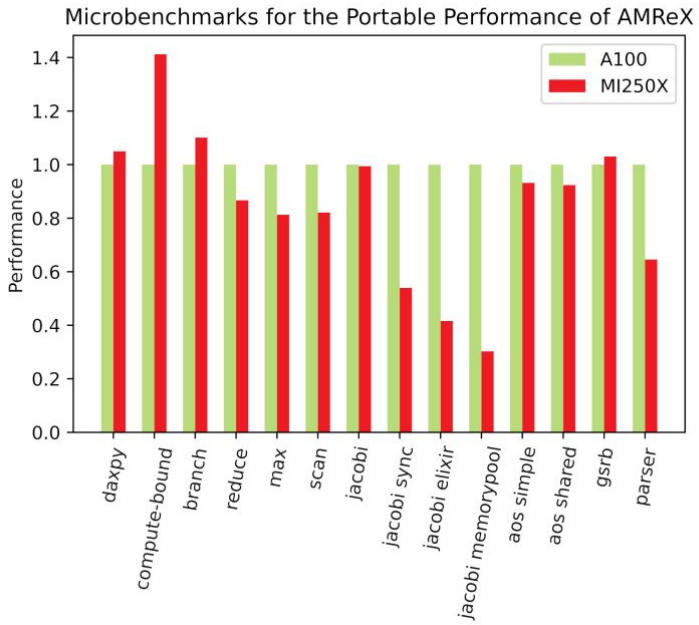
```
#ifdef _OPENMP
#pragma omp parallel if(Gpu::notInLaunchRegion())
#endif
    {

        for (MFIter mfi(state,TilingIfNotGPU()); mfi.isValid(); ++mfi)
        {
            const Box& bx  = mfi.tilebox();
            const auto statefab = state.array(mfi);
            const auto tagfab  = tags.array(mfi);
            Real phierror = phierr[lev];

            amrex::ParallelFor(bx,
            [=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
            {
                state_error(i, j, k, tagfab, statefab, phierror, tagval);
            });
        }
    }
```

Performance portability - ParallelFor looping construct

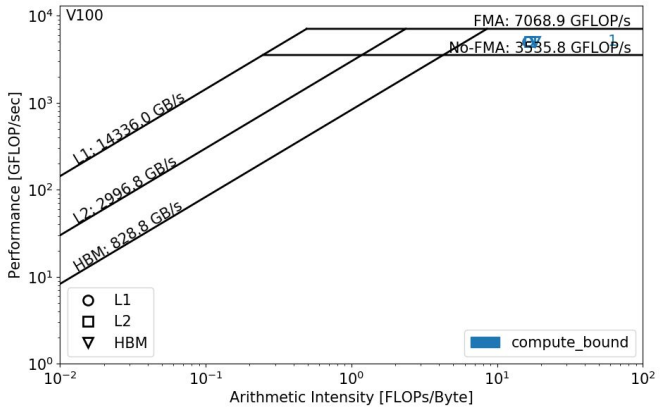
AMReX provides a ParallelFor looping construct based on C++ lambda functions

Similar to performance portability layers like Kokkos / Raja, but tailored to structured grid applications

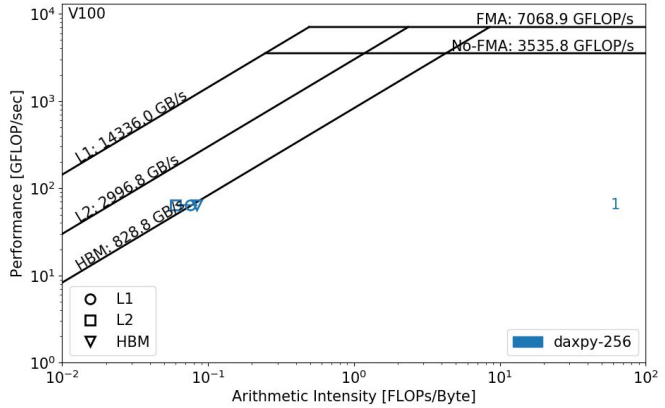
Translates to either a GPU kernel launch (CUDA / HIP / SYCL) or a normal for loop.




```
amrex::ParallelFor(bx,
[=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
{
    Real y = a(i,j,k);
    Real x = 1.0;
    for (int n = 0; n < 20; ++n) {
        Real dx = -(x*x-y) / (2.*x);
        x += dx;
    }
    a(i,j,k) = x;
});
```



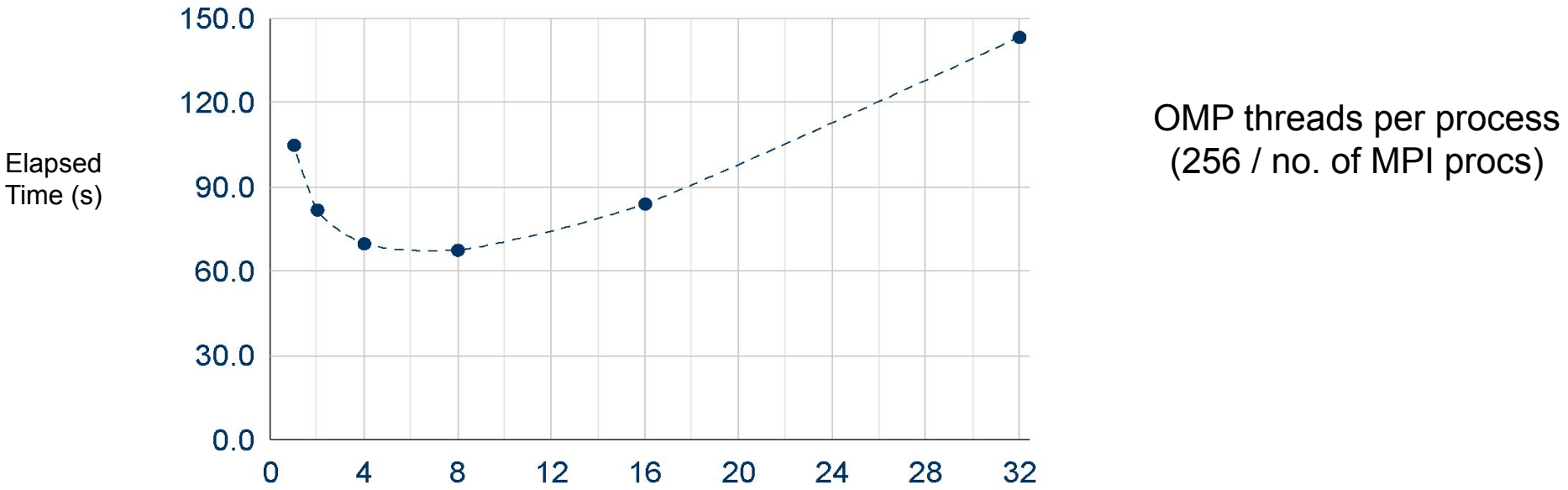

```
amrex::ParallelFor(bx,
[=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
{
    a(i,j,k) = a(i,j,k) + 0.5*b(i,j,k);
});
```



OpenMP on CPUs

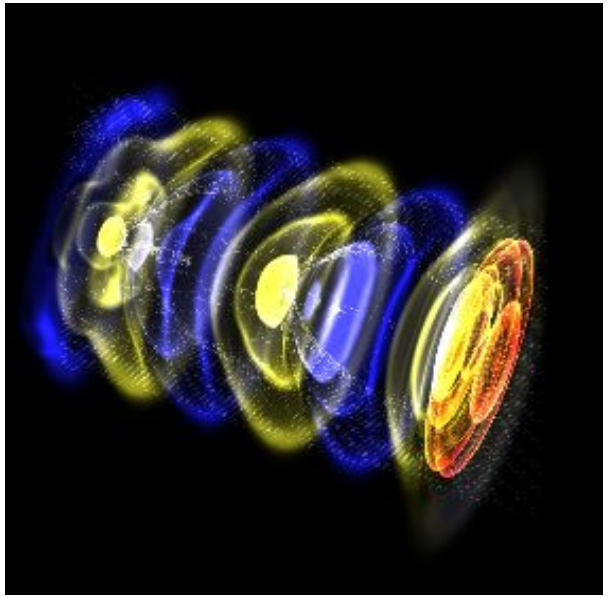
Atmospheric boundary layer simulations using the Energy Research and Forecasting (ERF) code

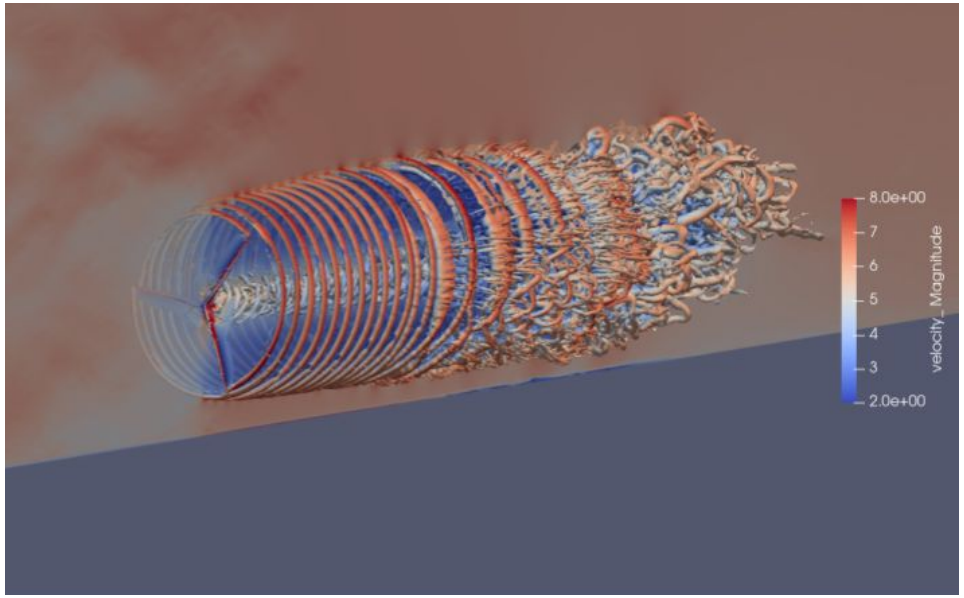
- 100 time steps, I/O and diagnostic calculations are turned off.
- Tested over 2 nodes (256 physical cores) on NERSC's Perlmutter system. The problem size and number of cores are kept constant while varying the balance of MPI processes and OMP threads per process.

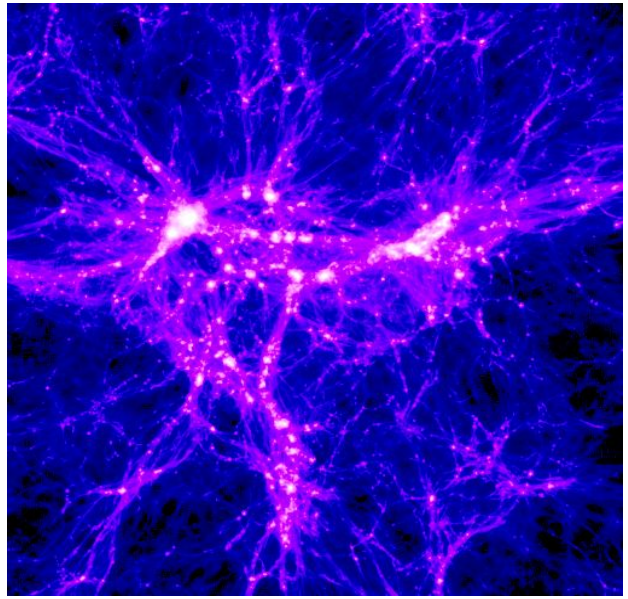


Beyond Linear Advection

This tutorial wasn't actually very complicated, but hopefully it suggests that if you don't want to write a parallel GPU-ready AMR code from scratch, something like this might be a good starting point…







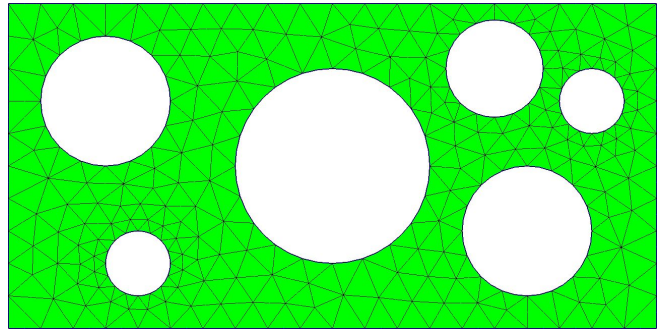
Let’s model the dye a different way …

Let’s imagine instead that we model the dye as a collection of particles. And let’s make the flow more interesting by inserting an obstacle in the flow.

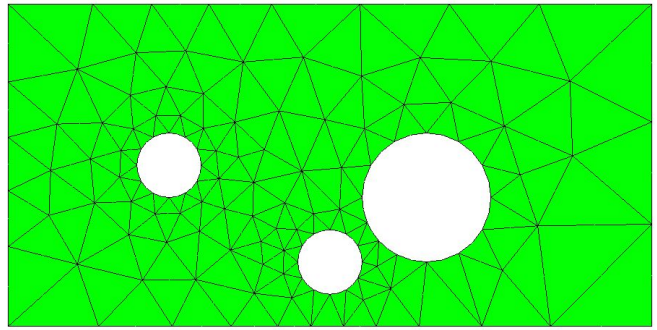
In addition to mesh and particle data, AMReX supports geometric data for solid obstacles/boundaries in the form of “cut cell” quantities (EB = embedded boundary representation)

Note that the cut cell / embedded boundary approach in a structured mesh is very different than an unstructured mesh:

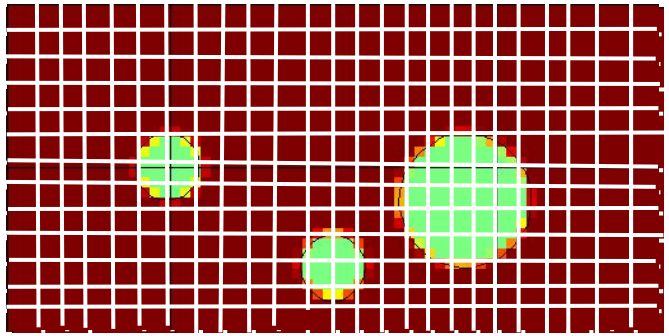
- In a structured mesh, “creating” the geometry means locally intersecting the object with the mesh
- In an unstructured mesh, “creating” the geometry means defining the entire mesh in a way that aligns with the object



Unstructured meshes change with the geometry



Structured meshes don’t … but we need to compute new intersections



AMReX Hands-On Examples

Let's do another hands-on exercise

"**AMR 102**": Particles and Linear Solvers and EB, Oh My!

- Fluid velocity initialized on cell faces
- Obstacle placed in the flow using EB approach
- Velocity field "projected" to ensure divergence-free flow around the object
- Passive particles move with the velocity field, mimicking the presence of the dye

Instructions for how to access and run the code are on the web page:

https://xsdk-project.github.io/MathPackagesTraining2026/lessons/amrex/

Particle-Mesh advection technique

Interpret the scalar ϕ as the number density of physical dye particles. Represent each physical particle by some number N_p of computational particles. Each particle stores a weight w_p equal to the number of physical particles it represents.

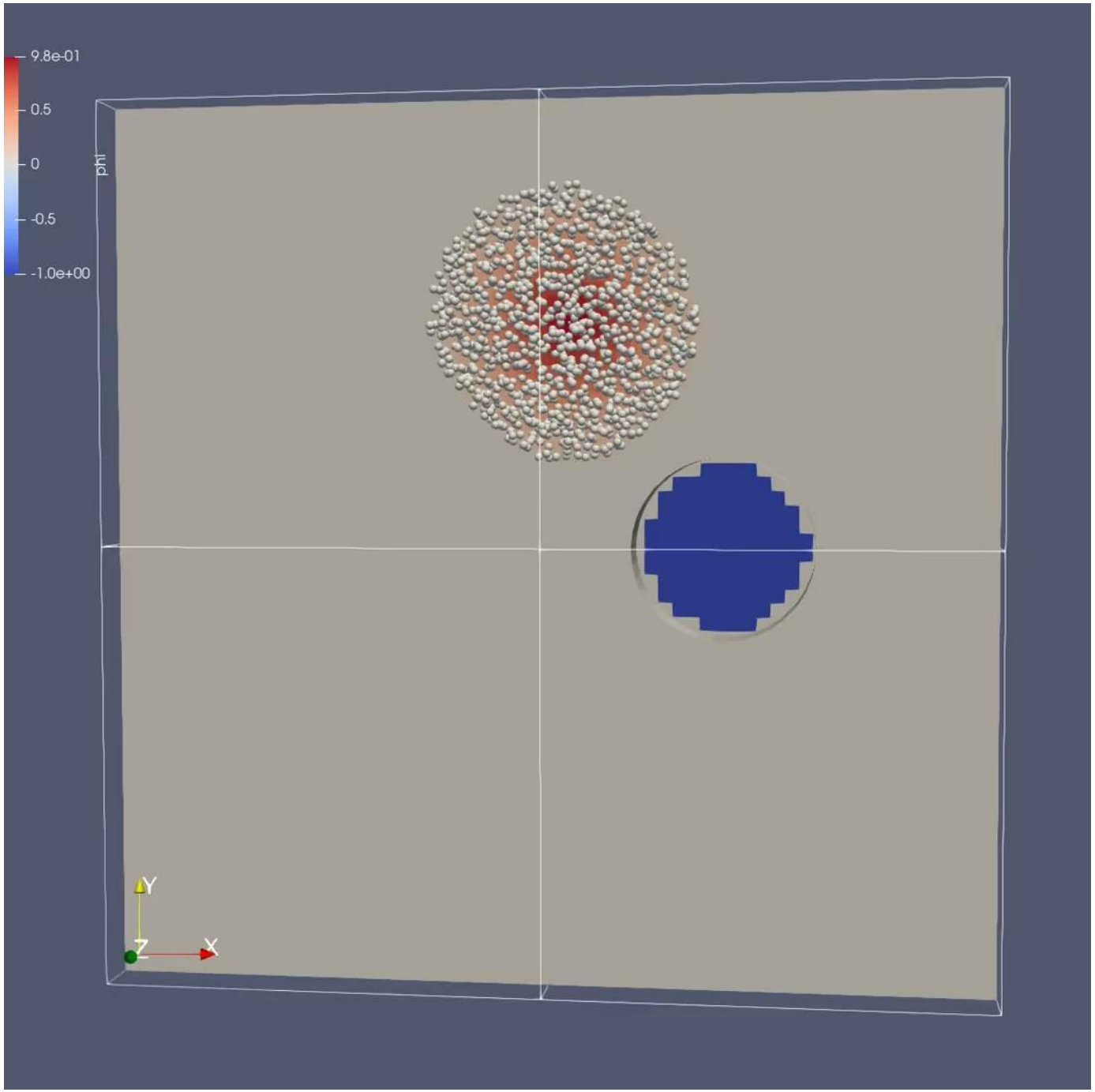
We can compute these weights by interpolating the grid quantity to the particle positions:

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

$$w_p = \phi_p \cdot \frac{dxdydz}{n_{ppc}}$$

Where S is the shape factor, which depends on the type of interpolation we are using. The particles are advanced in time by interpolating the velocity to their positions using the same shape factors. Any time we want to represent ϕ on the grid, we can compute:

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



Things to try with this one:

Change the properties of the cylinder (size, location, orientation)

Try more or fewer particles

Change the order of interpolation used in the PIC scheme

etc…

AMReX Hands-On Examples

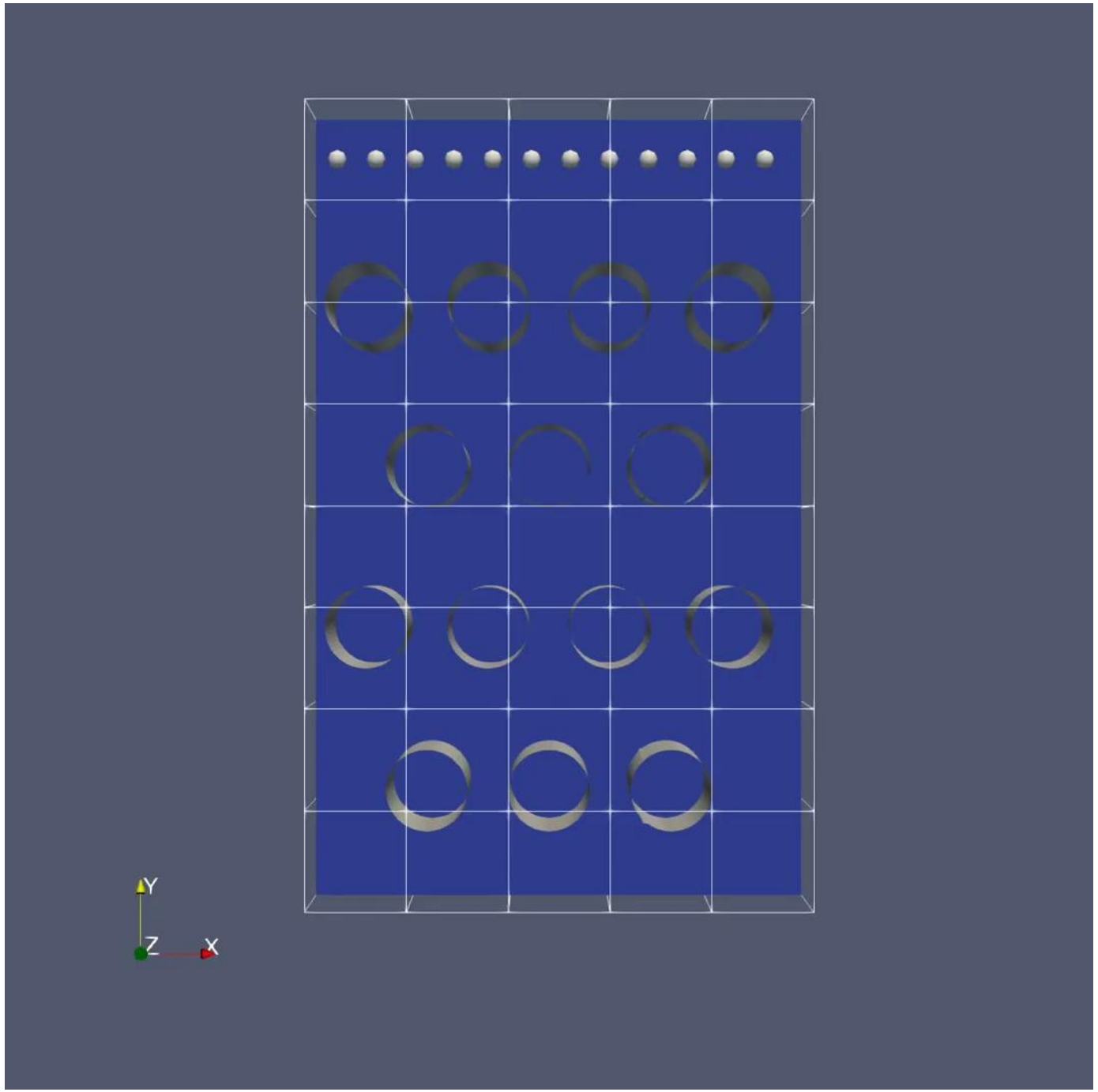
Final exercise:

"**AMReX-Pachinko**": Particles falling under gravity and bouncing off obstacles

- Demonstrates interaction of particles with solid boundaries and with walls

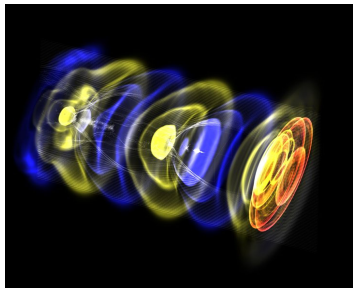
Instructions for how to access and run the code are on the web page:

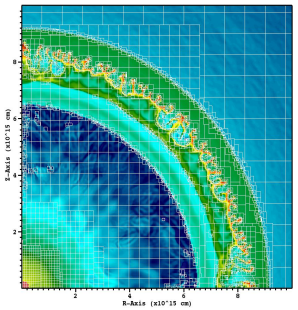
https://xsdk-project.github.io/MathPackagesTraining2026/lessons/amrex/


Y
Z
X

AMReX: software framework for block-structured AMR

Astrophysics:
- Castro (compressible)
- MaestroEx (low-Mach)
- SedonaEx (Monte Carlo radiation transport)
- Emu (neutrino transport)
- Quokka (radiation-hydrodynamics)
- GRTeclyn (numerical relativity)





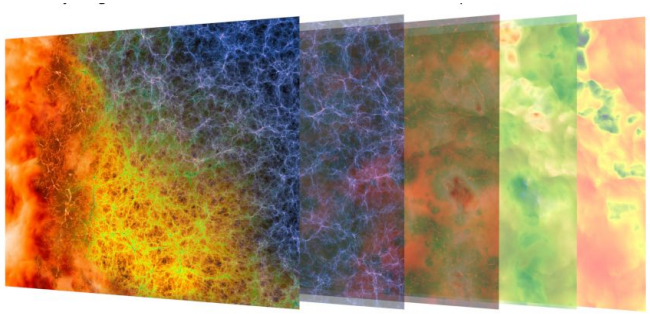
Incompressible Navier-Stokes:
- IAMR
- incflo

Ocean Modeling:
- Remora

Solid Mechanics:
- Alamo

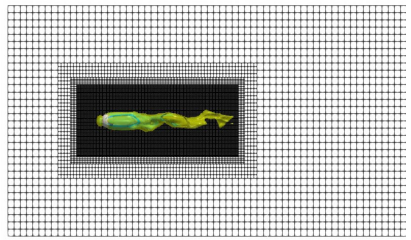
Atmospheric science:
- AMR-wind
- ERF

Biological cell modelling:
- BoltzmanMFX
- CCM

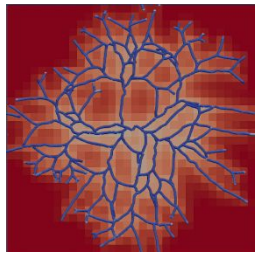


Cosmology:
- Nyx

Combustion:
- PeleC (Compressible)
- PeleLM (Low Mach)



Multi-phase Flow:
- MFIX-Exa

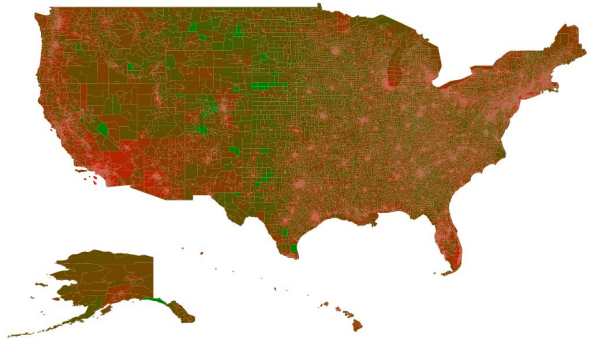


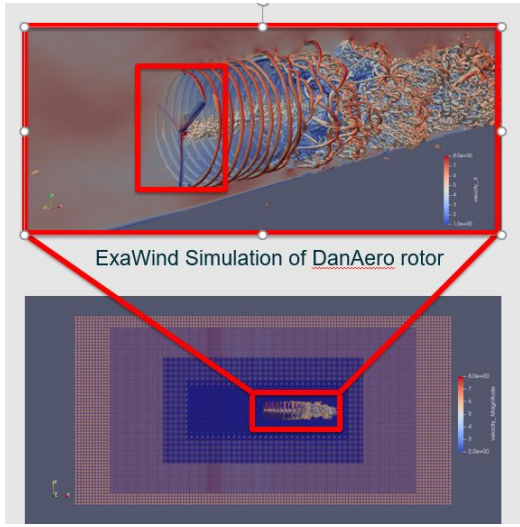
Accelerator Modelling:
- WarpX
- ImpactX
- Hipace++

Multiscale Modelling and Stochastic Systems:
- FHDeX

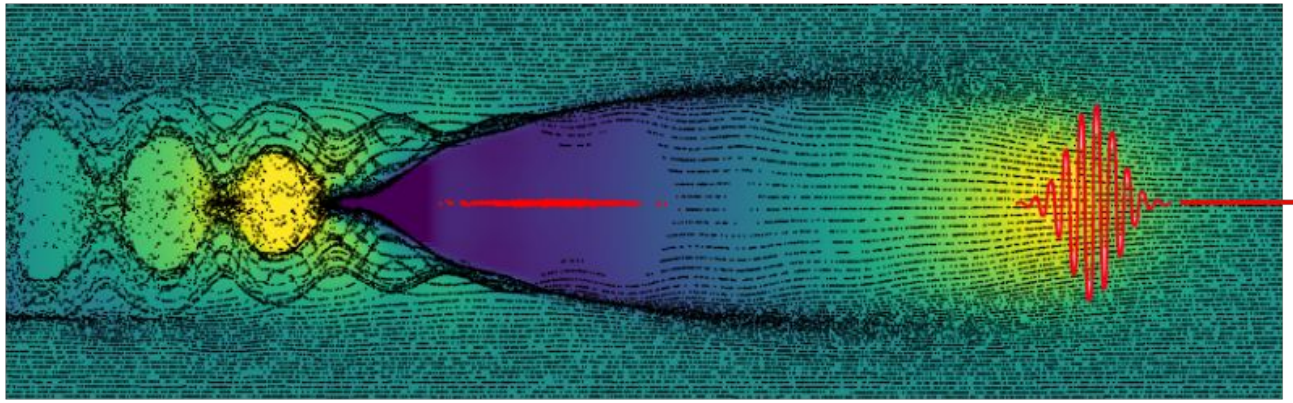
Electromagnetics:
- ARTEMIS

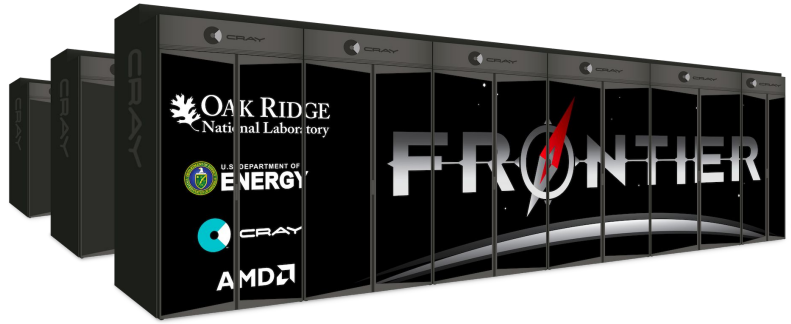
Epidemiology:
- ExaEpi





WarpX: 500x FOM, Gordon Bell Award





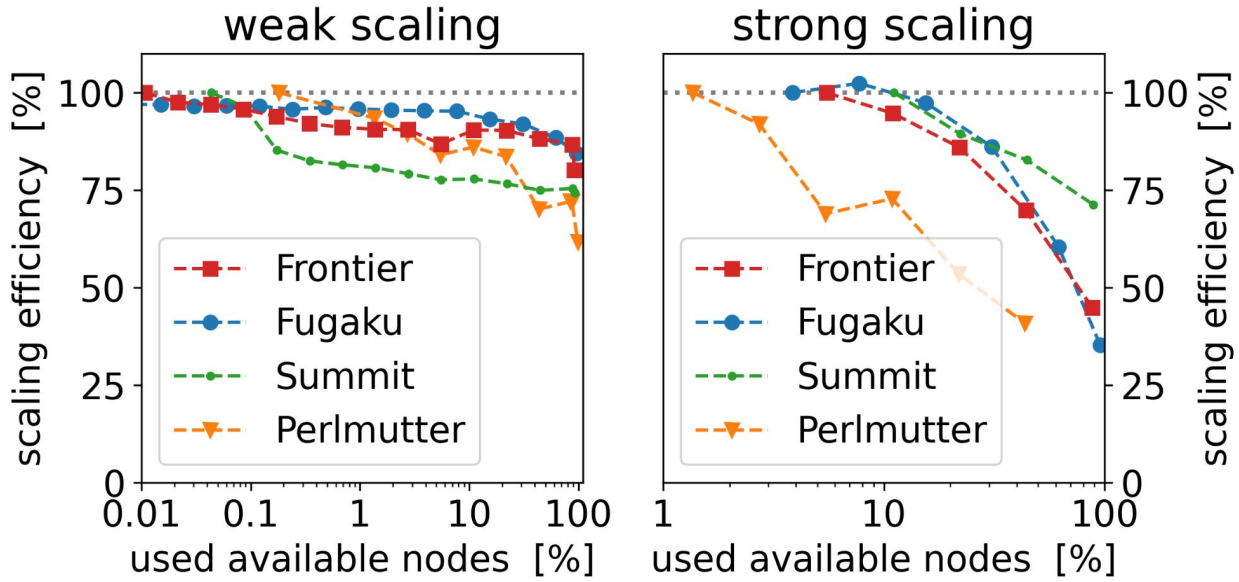
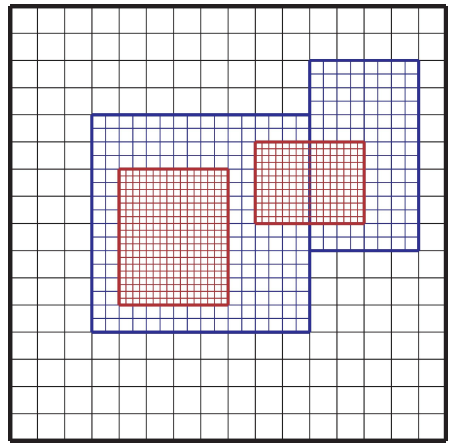


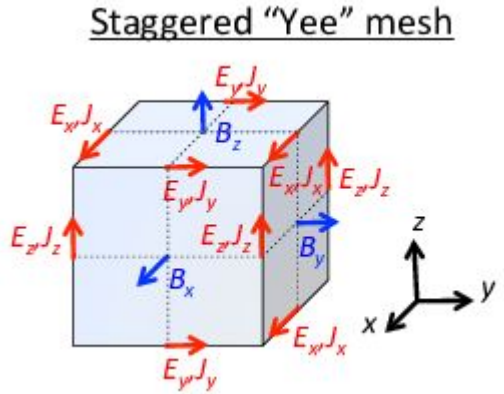
Figure-of-Merit over time

Date	Code	Machine	N_c/Node	Nodes	FOM
3/19	Warp	Cori	0.4e7	6 625	2.2e10
3/19	WarpX	Cori	0.4e7	6 625	1.0e11
6/19	WarpX	Summit	2.8e7	1 000	7.8e11
9/19	WarpX	Summit	2.3e7	2 560	6.8e11
1/20	WarpX	Summit	2.3e7	2 560	1.0e12
2/20	WarpX	Summit	2.5e7	4 263	1.2e12
6/20	WarpX	Summit	2.0e7	4 263	1.4e12
7/20	WarpX	Summit	2.0e8	4 263	2.5e12
3/21	WarpX	Summit	2.0e8	4 263	2.9e12
6/21	WarpX	Summit	2.0e8	4 263	2.7e12
7/21	WarpX	Perlmutter	2.7e8	960	1.1e12
12/21	WarpX	Summit	2.0e8	4 263	3.3e12
4/22	WarpX	Perlmutter	4.0e8	928	1.0e12
4/22	WarpX	Perlmutter†	4.0e8	928	1.4e12
4/22	WarpX	Summit	2.0e8	4 263	3.4e12
4/22	WarpX	Fugaku†	3.1e6	98 304	8.1e12
6/22	WarpX	Perlmutter	4.4e8	1 088	1.0e12
7/22	WarpX	Fugaku	3.1e6	98 304	2.2e12
7/22	WarpX	Fugaku†	3.1e6	152 064	9.3e12
7/22	WarpX	Frontier	8.1e8	8 576	1.1e13

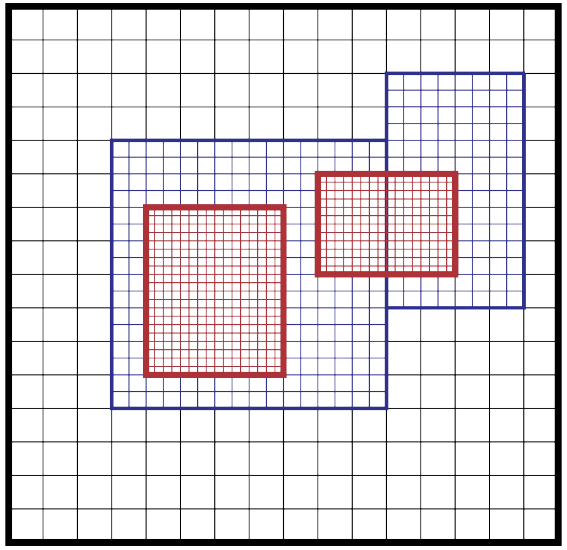
Mesh Data Structures

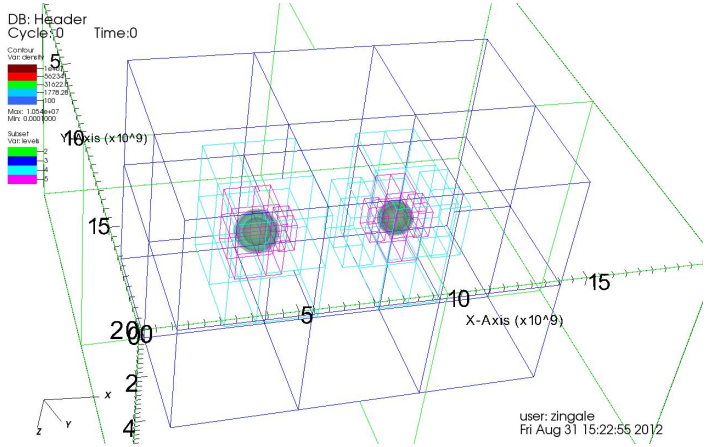
- Most AMReX applications involve mesh data of some kind
- AMReX provides tools for storing distributed mesh data (i.e. multidimensional arrays associated with each patch in the AMR hierarchy).
- Mesh data can be cell, face, edge, or node centered. Useful for, e.g.
  - staggered “Yee” grid for electromagnetics / Arakawa C-grid
  - MAC projection in fluid simulation
  - etc…
- Also provide tools for parallel communication (gpu-aware):
  - Ghost cell exchange (Fillboundary - copy valid to ghost, SumBoundary - add ghost to valid), Nodal Syncing
  - General ParallelCopy - copy / add from one layout to another



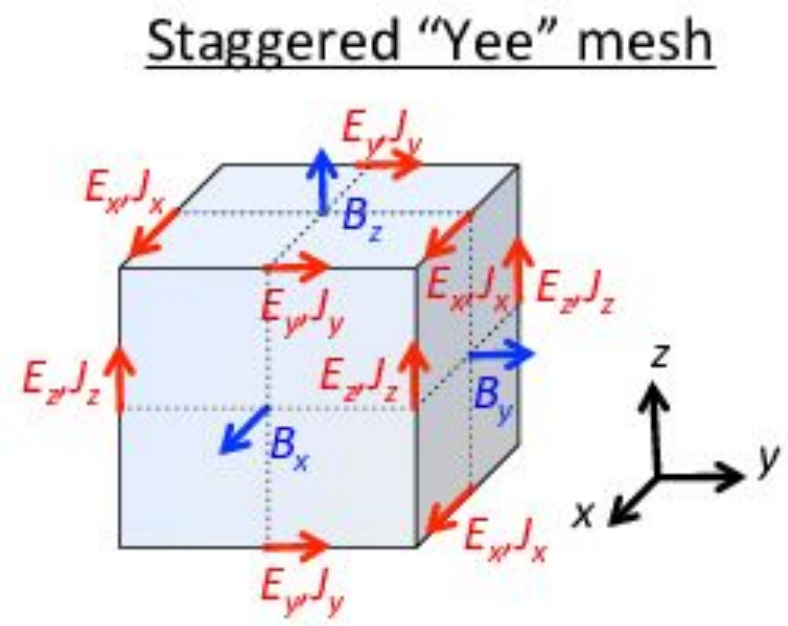


Box, IntVect, & IndexType





- Box represents a (logically) rectangular domain in global integer indexing space.
- IntVect is an integer tuple.
- IndexType: cell or node. For example, Bx on x-face: (node, cell, cell)



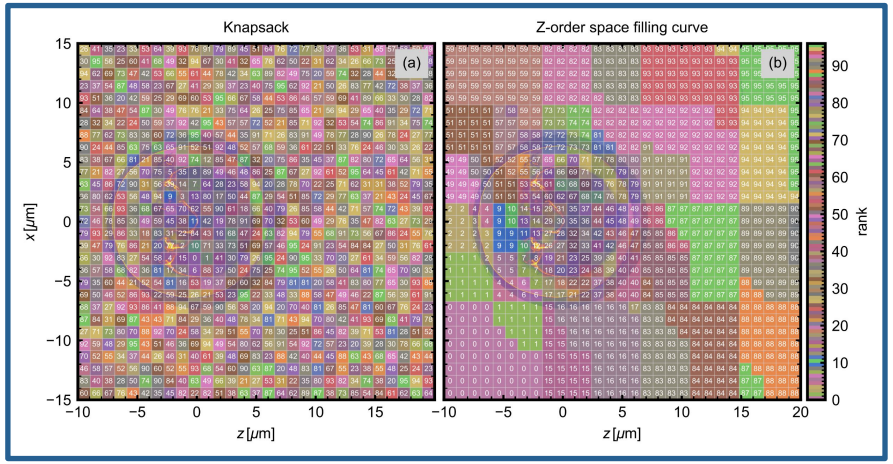
BoxArray & DistributionMapping

```
Box domain(IntVect(0), IntVect(127)); // a box with 128^3 cell
BoxArray ba(domain); // Make a new BoxArray w/ one Box
ba.maxSize(64); // Chop into Boxes of 64^3 cells
Print() << ba << "\n";
```

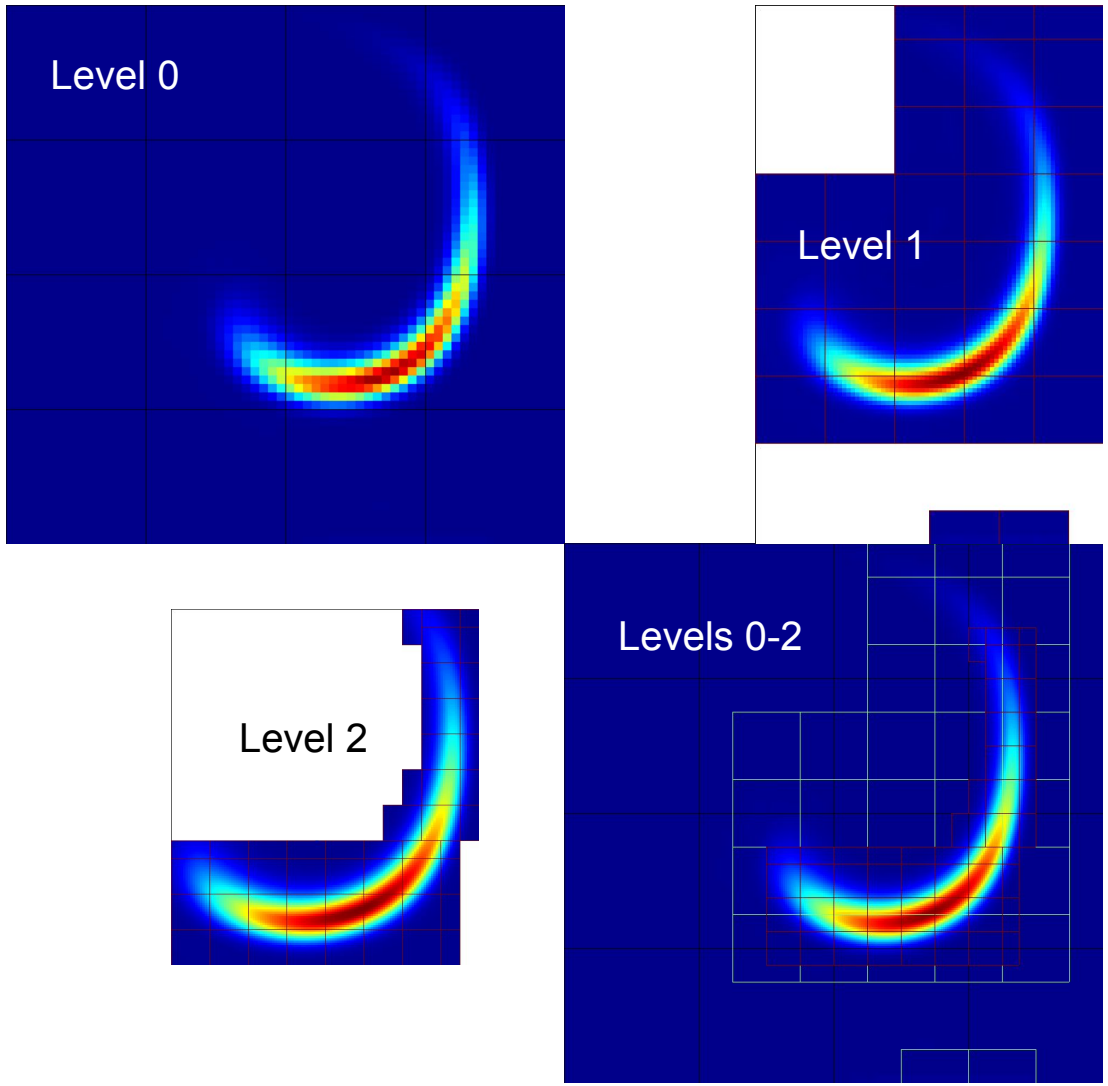
```
(BoxArray maxbox(8)
       m_ref->m_hash_sig(0)
((0,0,0) (63,63,63) (0,0,0)) ((64,0,0) (127,63,63) (0,0,0))
((0,64,0) (63,127,63) (0,0,0)) ((64,64,0) (127,127,63) (0,0,0))
((0,0,64) (63,63,127) (0,0,0)) ((64,0,64) (127,63,127) (0,0,0))
((0,64,64) (63,127,127) (0,0,0)) ((64,64,64) (127,127,127) (0,0,0))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

```
DistributionMapping dm(ba);
```

- BoxArray is an array of Boxes on a single AMR level.
- DistributionMapping describes which MPI process owns the data for Boxes in a BoxArray.
- DistributionMapping strategies: Space filling curve, knapsack, round robin, etc.



MultiFab



- MultiFab is a distributed container for data on a single level.
- Multiple Fab (Fortran Array Box). Fortran multi-D array convention (i.e., column major).
- Can have multiple components. Multiple 4-D arrays.
- Can have ghost cells.
- What memory to use?
- non-copyable, but movable.

```
// ba: BoxArray
// dm: DistributionMapping
int ncomp = 3;
IntVect nghost(1,2,0);
MultiFab mf(ba, dm, ncomp, nghost);
MultiFab hmf(ba, dm, 1, 0,
   MFInfo().SetArena(The_Cpu_Arena()));
MultiFab mf2(std::move(mf));
mf2 = std::move(mf);
std::swap(hmf,mf2);
```

Examples of MultiFab Functions

```
mf = 3.14; // set value

mf.LocalCopy(mf2, …); // Copy data from mf2 to mf. The two have the same layout.
mf.ParallelCopy(mf2, …); // Copy data w/ MPI. The two have different BoxArray
                         //   and DistributionMapping.
mf.FillBoundary(…); // Ghost cell exchange

auto mfmin = mf.min(…); // Return the minimum value

mf.minus(mf2, …); // mf = mf - mf2

// Create an alias MultiFab without deep-copying the data
// Suppose the original MultiFab stores density, x-velocity, y-velocity, and
// z-velocity. We need to call a function expecting x-velocity being the first
// component.
MultiFab mf_alias(mf, amrex::make_alias, xvel_comp, 3);

// We can also take raw pointers and create a MultiFab without deep-copying or
// taking the ownership. Some users use amrex this way in their existing codes.
```

FArrayBox, Array4 & MFIter

- A MultiFab contains multiple FArrayBoxes.
- FArrayBox: container for multi-dimensional (3+1) array. Movable but not copyable.
- Array4: 4D array view similar to C++23 std::mdspan, but with negative indexing support. Fortran array syntax.
- FArrayBox (usually) owns the memory, whereas Array4 never owns the memory making it trivially copyable.
- MFIter: Iterator for MultiFab. It supports logical tiling, which for CPU runs improve cache efficiency and OpenMP performance.

```
auto const lo = amrex::lbound(mfi.validbox());
auto const hi = amrex::ubound(mfi.validbox());
for (int k = lo.z; k <= hi.z; ++k) {
for (int j = lo.y; j <= hi.y; ++j) {
for (int i = lo.x; i <= hi.x; ++i) {
  a(i,j,k) = a2(i,j,k);
}}}
```

→

```
// mf & mf2 have the same layout
for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  FArrayBox& fab = mf[mfi];
  FArrayBox const& fab2 = mf2[mfi];
  fab.template
      copy<RunOn::Device>(fab2);
}

for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  Array4<Real> const& a =
                 mf.array(mfi);
  Array4<Real const> const& a2 =
                 mf.const_array(mfi);
  ParallelFor(mfi.validbox(),
              [=] AMREX_GPU_DEVICE
              (int i, int j, int k) {
      a(i,j,k) = a2(i,j,k);
  });
}
```

FArrayBox, Array4 & MFIter

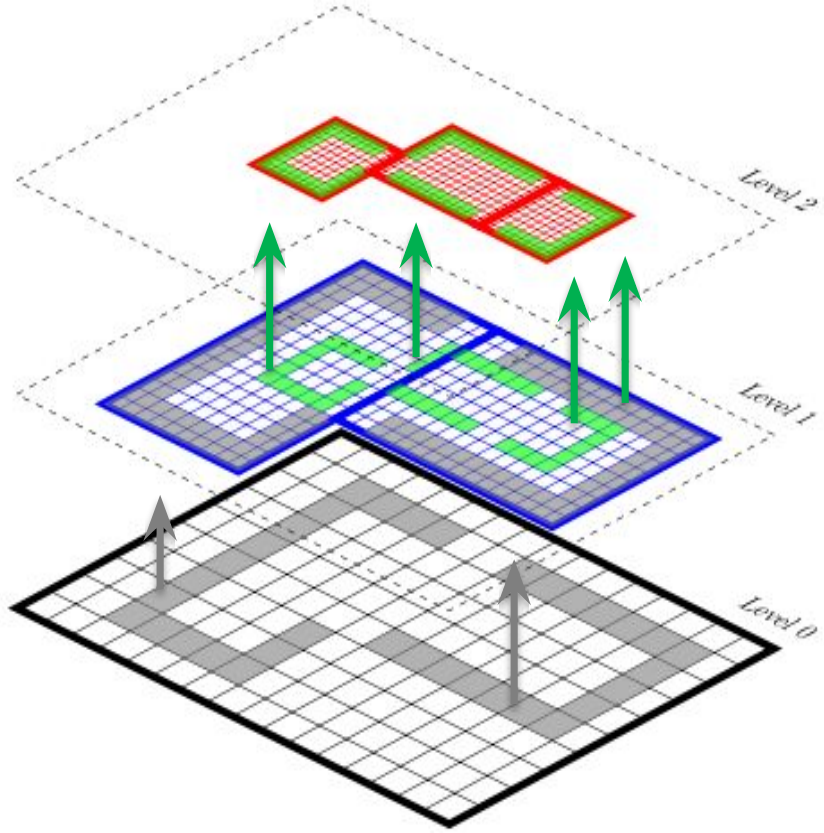
- A MultiFab contains multiple FArrayBoxes.
- FArrayBox: container for multi-dimensional (3+1) array. Movable but not copyable.
- Array4: 4D array view similar to C++23 std::mdspan, but with negative indexing support. Fortran array syntax.
- FArrayBox (usually) owns the memory, whereas Array4 never owns the memory making it trivially copyable.
- MFIter: Iterator for MultiFab. It supports logical tiling, which for CPU runs improve cache efficiency and OpenMP performance.

```
auto const& a = mf.arrays();
auto const& a2 = mf.const_arrays();
// a single kernel for multiple Boxes
ParallelFor(mf, [=] AMREX_GPU_DEVICE
            (int bno, int i, int j, int k)
{
  a[bno](i,j,k) = a2[bno](i,j,k);
});
```

```
// mf & mf2 have the same layout
for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  FArrayBox& fab = mf[mfi];
  FArrayBox const& fab2 = mf2[mfi];
  fab.template
      copy<RunOn::Device>(fab2);
}

for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  Array4<Real> const& a =
                 mf.array(mfi);
  Array4<Real const> const& a2 =
                 mf.const_array(mfi);
  ParallelFor(mfi.validbox(),
              [=] AMREX_GPU_DEVICE
              (int i, int j, int k) {
      a(i,j,k) = a2(i,j,k);
  });
}
```

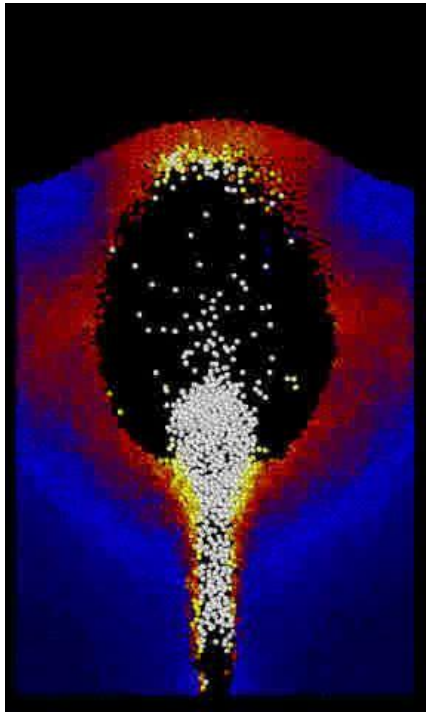
AMR Operations

- Classes: Amr/AmrLevel (more built-in functionalities), AmrCore (more flexibilities)
- Regridding: Tagging cells for refinement, grid generation, data filling, …
- Interpolation: Ghost cell filling
- Restriction: Average fine data down to coarse level
- Flux registers: Refluxing for face fluxes in conservation law systems and edge fluxes in MHD on Yee grids.
- Time stepping
  - Subcycling in time or non-subcycling
  - 2nd order Godunov method
  - Method of lines with high-order Runge-Kutta
  - Spectral deferred corrections
- Multi-level linear system solvers

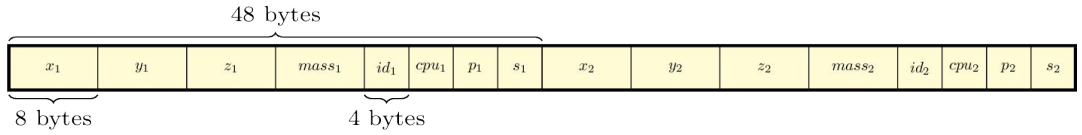


Particle Data Structures

- Most AMReX applications also have particles - usually these live on / interact with an AMR hierarchy of mesh data.
- These could be passive, tracers, charged particles interacting through self-forces, fluid-particle drag, particle-particle collisions.
- Users can specify combination of AoS / SoA data layout.
- Tools for particle mesh interpolation, neighbor list construction + iteration.
- Tools for binning, sorting, stream compaction, partition.
- Parallel Communication patterns:
  - Both local and global redistribution
  - Halo exchange

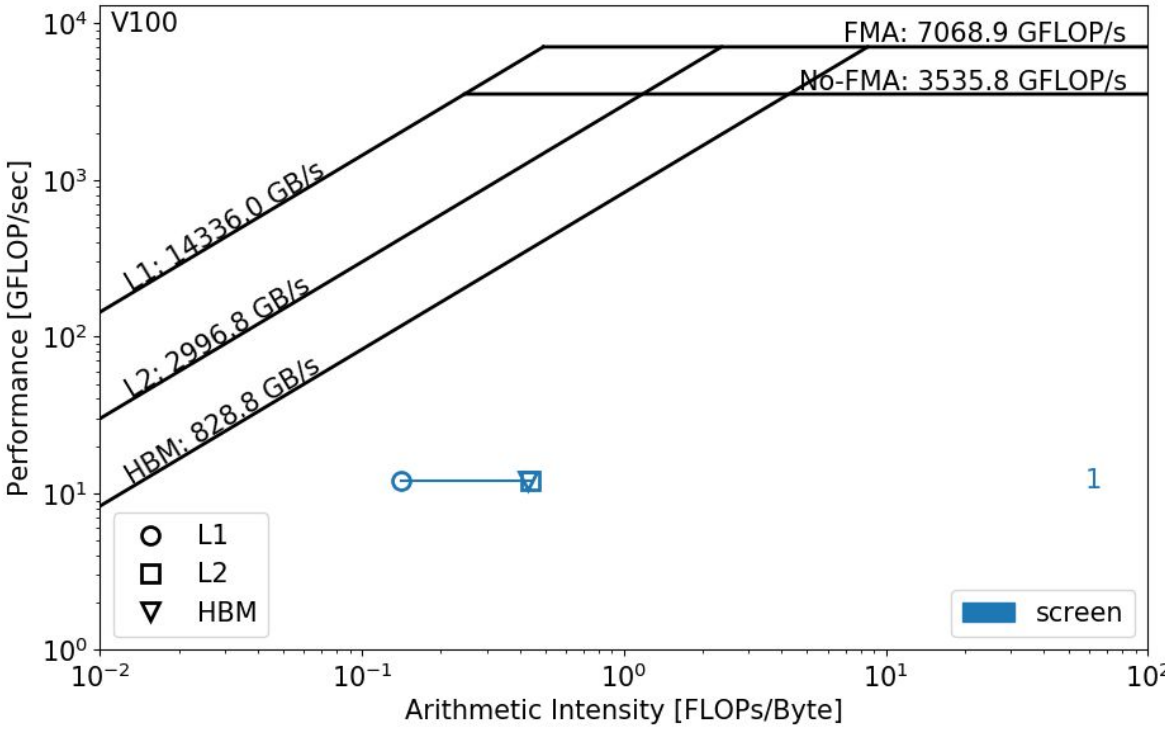


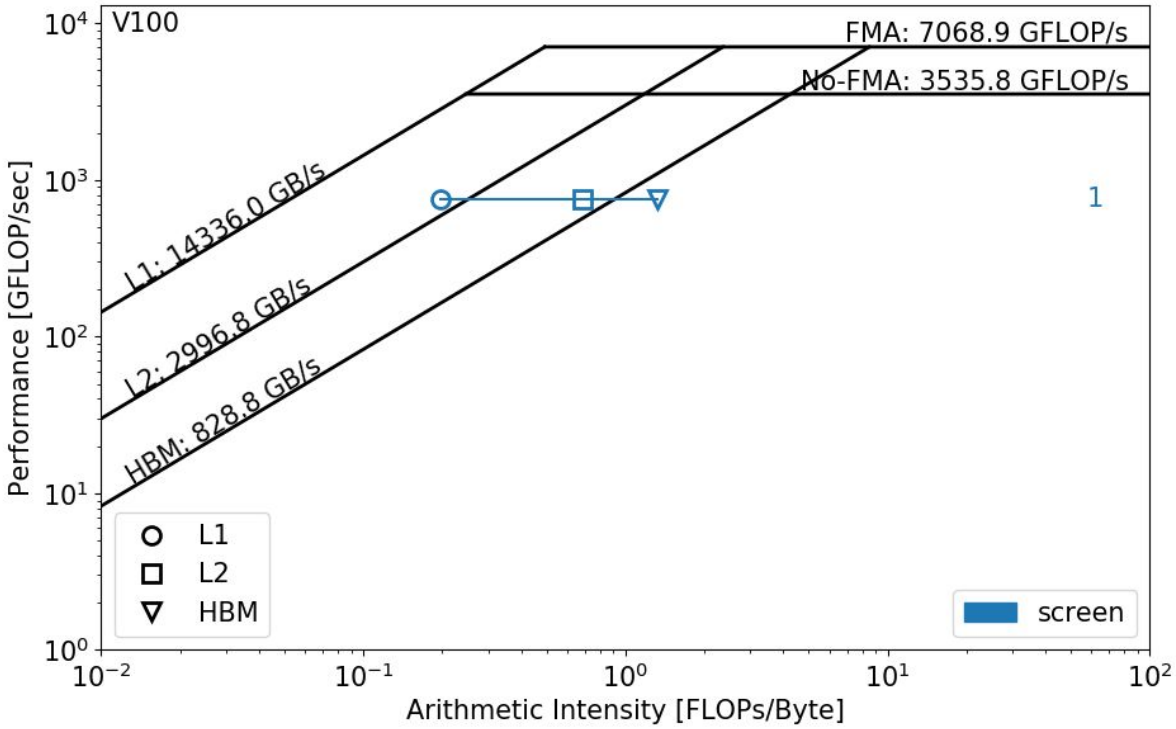
Array-of-Structs



Struct-of-Arrays

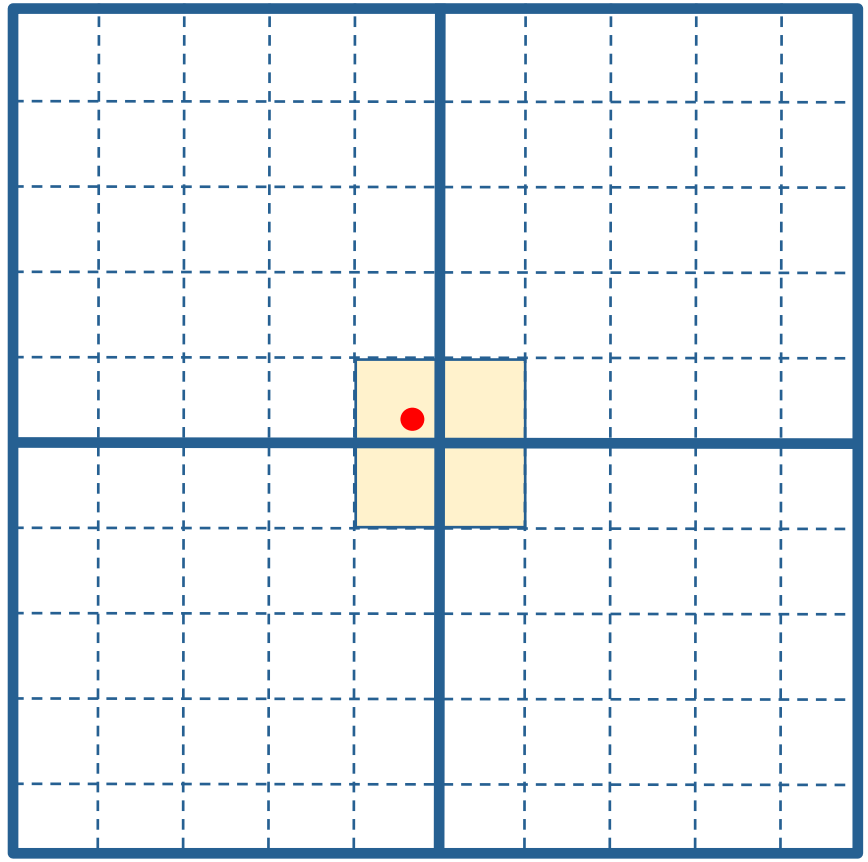
AoS vs. SoA: MFiX-Exa





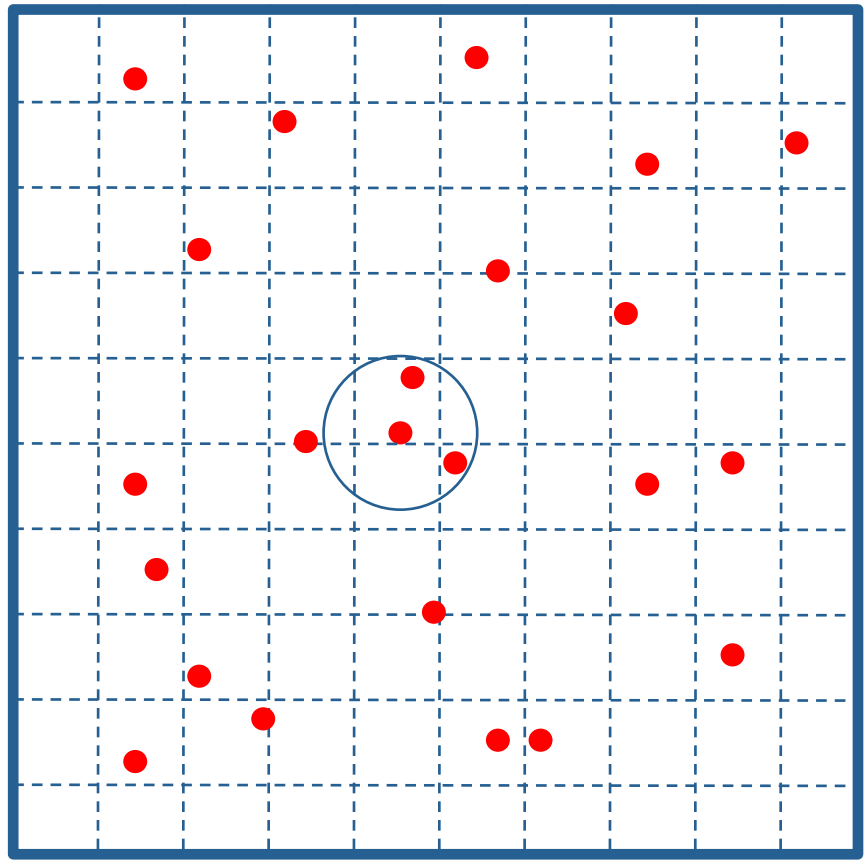
Particle-Mesh Interaction

- AMReX provides a set of tools for handling particle-mesh interaction
- Simple interpolation kernels (NGP, CIC) for mesh-to-particle and particle-to-mesh interpolation are provided
- Lambda function interface for performing user-provided operations interpolations (e.g. high-order particle shapes, Gaussian kernels)
- Performs needed parallel communication “under-the-hood” (SumBoundary)
- Support for dual-grid
- Uses data duplication strategy for CPU threading, per-particle atomics for GPU threading.



Particle-Particle Interaction

- AMReX includes tools for building and iterating over neighbor lists for MD-style particle-particle interactions
- Pre-compute a list of potential interaction partners that can be reused for multiple time steps
- Uses a cell list to avoid direct N^2 search of all possible pairs
- Users can specify operation using lambda function.
- Support for half and full neighbor lists (tradeoff based on amount of contention / atomics performance)



Embedded Boundary (Cut Cell) Support:

Use a cut cell approach for complex geometries

- Special stencils at/near cut cells
- Regular stencils elsewhere
- Connectivity information stored in a single integer.

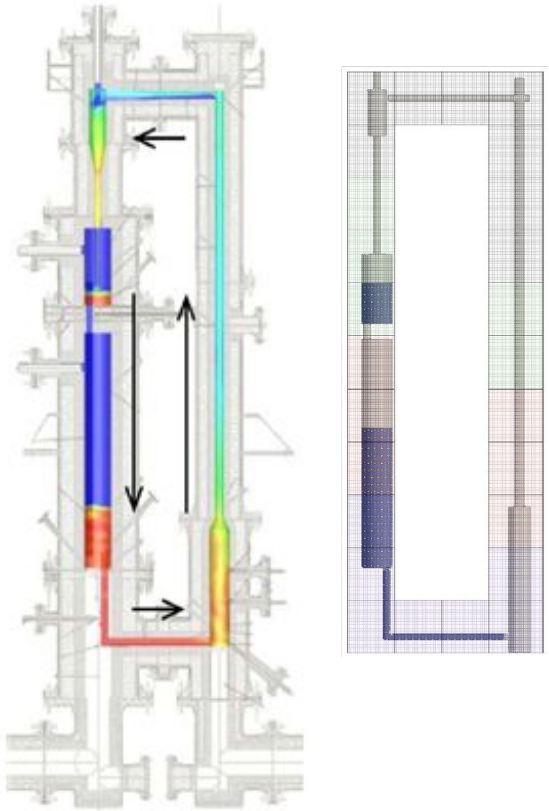
Grids/tiles can be queried as to whether they contain irregular cells

Support for EB-aware

- interpolation, e.g. from cell centers to face centroids
- restriction
- slopes
- linear solvers (cell-centered and nodal)

Option for mesh pruning (for memory savings)

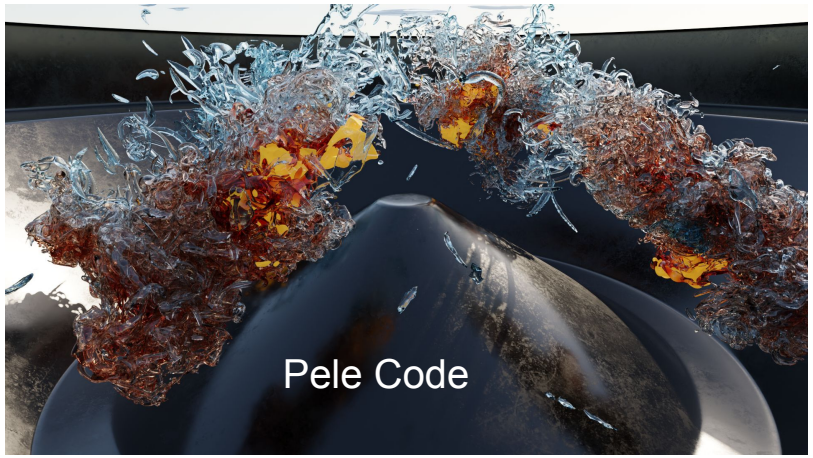
Particles can see boundary through level set function

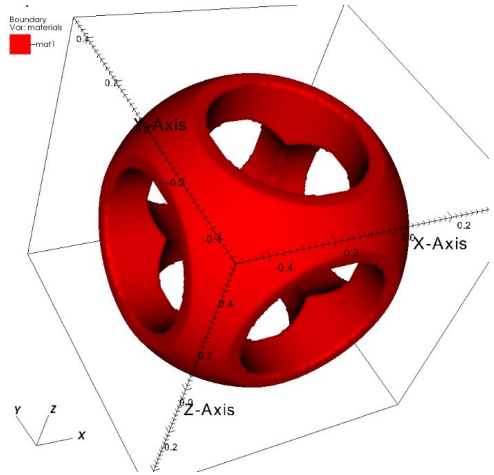


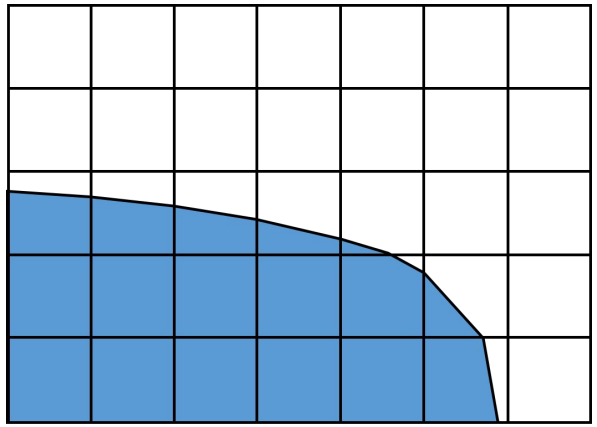
(L) Prototype of Chemical Looping Reactor (CLR) from MFIX-Exa project

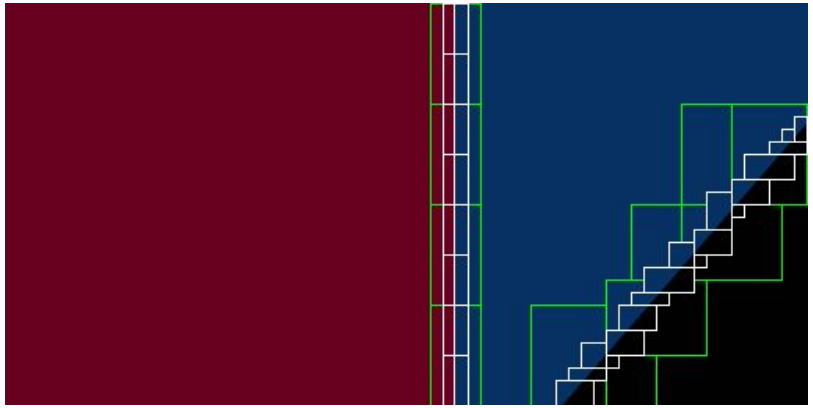
(R) Grid generation using mesh pruning

Embedded Boundary (Cut Cell) Support:









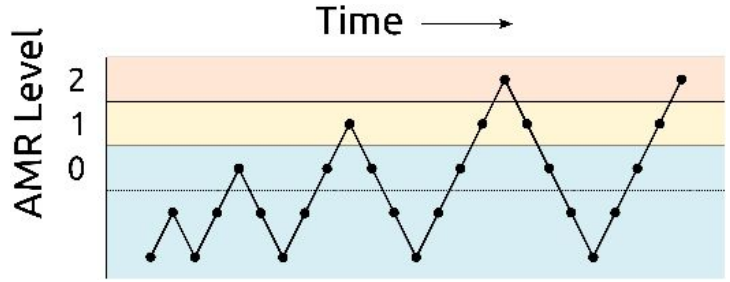
- Geometry can be generated by implicit functions or STL.
- Support for moving EB.
- An example of using CSG (constructive solid geometry).

```
EB2::SphereIF sphere(0.5, {0.0,0.0,0.0}, false);
EB2::BoxIF cube({-0.4,-0.4,-0.4}, {0.4,0.4,0.4}, false);
auto cubesphere = EB2::makeIntersection(sphere, cube);

EB2::CylinderIF cylinder_x(0.25, 0, {0.0,0.0,0.0}, false);
EB2::CylinderIF cylinder_y(0.25, 1, {0.0,0.0,0.0}, false);
EB2::CylinderIF cylinder_z(0.25, 2, {0.0,0.0,0.0}, false);
auto three_cylinders = EB2::makeUnion(cylinder_x, cylinder_y, cylinder_z);

auto csg = EB2::makeDifference(cubesphere, three_cylinders);
```

Geometric Multigrid (GMG) Solvers:

Support for multi-AMR level GMG solvers

- cell vs nodal data
- variable coefficient Poisson or Helmholtz equation
- tensor solve for Navier-Stokes
- single- vs multi-component



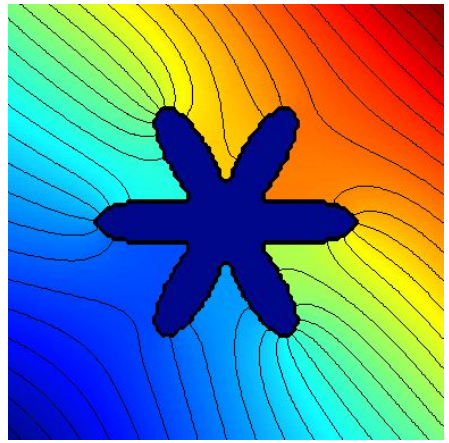
EB-aware options

Option for masking to enable overset mesh algorithms

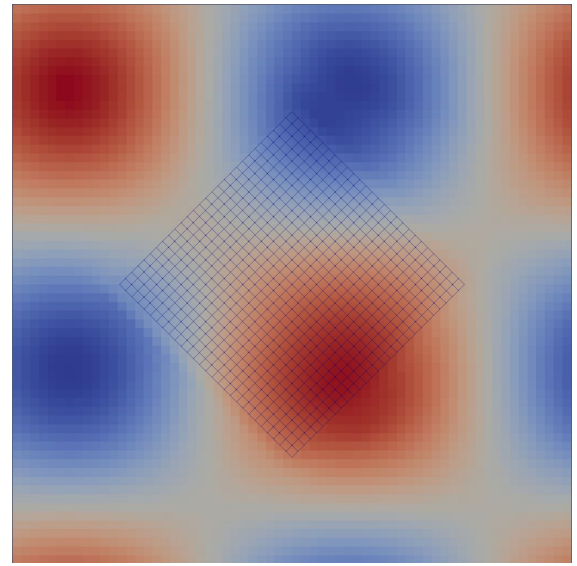
Native “bottom solver” is BiCG (or CG or both)

Interface to call hypre or PETSc as “bottom solver” -- which can be at any multigrid level, including the top

Options for agglomeration and consolidation on coarsening



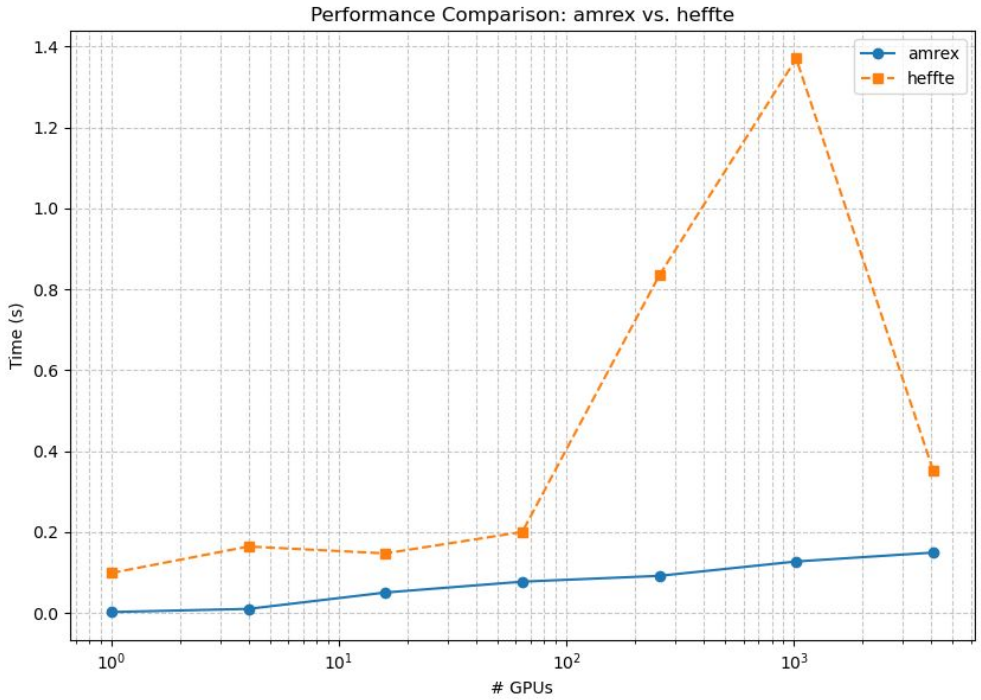
Embedded Boundaries



Overset mesh

Parallel FFTs

- AMReX was not designed to be a FFT library. Due to users' feature requests, we have developed parallel FFT capabilities.
- Use FFTW on CPUs, and cuFFT, rocFFT and Intel MKL on Nvidia, AMD and Intel GPUs.
- Arbitrary domain decomposition, not just pencil or slab, not just one box per process.
- A series of 1D/2D FFTs plus parallel communication and data rotation.



pyAMReX: Python bindings for AMReX

- Bridges the compute in AMReX block-structured codes and data science.
- Provides zero-copy application GPU data access for AI/ML, in situ analysis, application coupling and enables rapid, massively parallel prototyping.
- C++ library using pybind11.

```python
import amrex.space3d as amr
import cupy as cp

mfab = amr.MultiFab(<...>)
for mfi in mfab:
  array4 = mfab.array(mfi)
  cp_arr = cp.array(array4, copy=False)
  …
```

```cpp
// C++
MultiFab mfab(...);
for (MFIter mfi(mfab); mfi.isValid(); ++mfi) {
  auto const& array4 = mfab.array(mfi);
  …
}
```

https://github.com/AMReX-Codes/pyamrex

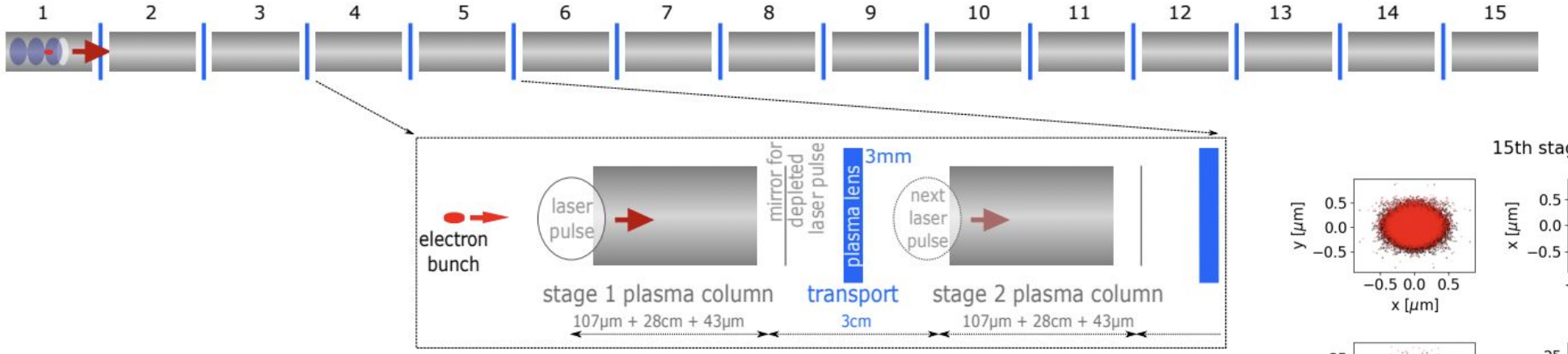
pyAMReX Example: MPMD

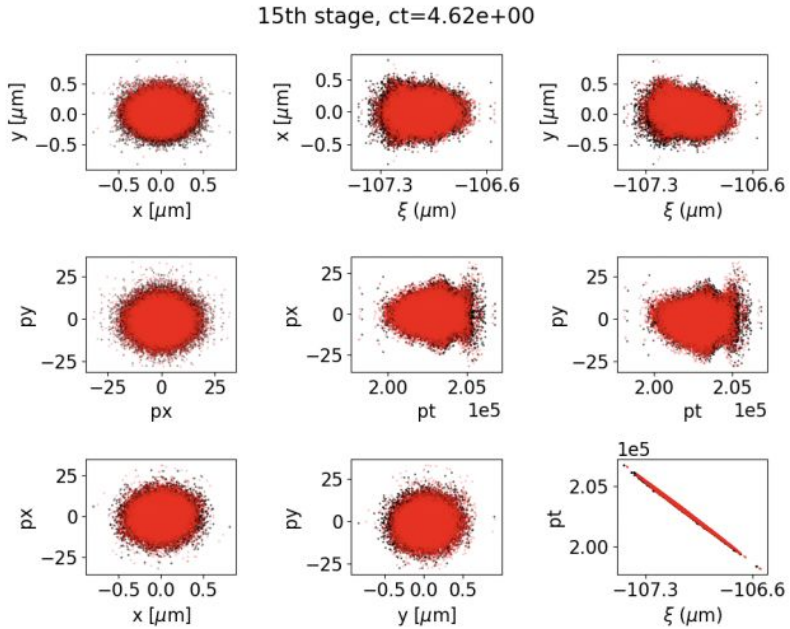
MPMD: Multiple Programs, Multiple Data
C++: computational fluid dynamics
Python: molecular dynamics and machine learning for rheology

```cpp
// main.cpp

#include <AMReX_MPMD.H>

int main(int argc, char* argv[])
{
    // Initialize amrex::MPMD to establish communication
    // across the two apps
    MPI_Comm comm = amrex::MPMD::Initialize(argc, argv);
    amrex::Initialize(argc,argv,true,comm);

    // C++: do work and communicate w/ python code

    amrex::Finalize();
}
```

```python
# main.py

from mpi4py import MPI

import amrex.space3d as amr

# Initialize amrex::MPMD to establish communication across
# the two apps
amr.MPMD_Initialize_without_split([])
app_comm_py = MPI.COMM_WORLD.Split(amr.MPMD_AppNum(),
                                   amr.MPMD_MyProc())
# Initialize AMReX
amr.initialize_when_MPMD([], app_comm_py)

// py: do work and communicate w/ python code
```

pyAMReX example: ImpactX

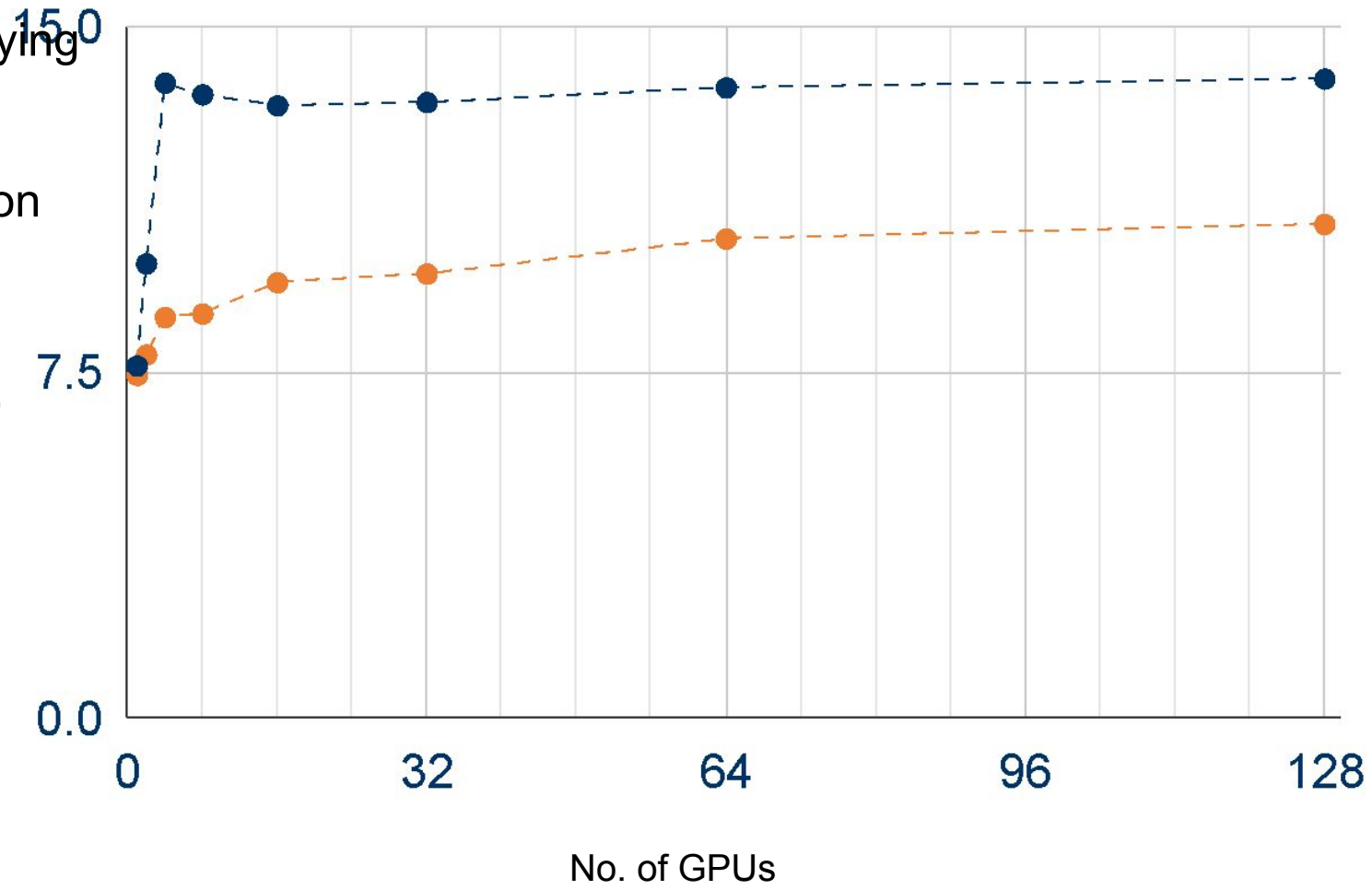




Synthesizing Particle-in-Cell Simulations Through Learning and GPU Computing for Hybrid Particle Accelerator Beamline, Sandberg et al 2024

GPU-optimized parallel communication

- All prep work done without copying data off the device
- Takes advantage of kernel fusion for buffer packing / unpacking
- GPU aware MPI:
  `amrex.gpu_aware_mpi=1`
- System specific setups



Scan (Prefix Sum, Partial Sum) on GPU

```
PrefixSum<T>(N, [=] AMREX_GPU_DEVICE (int i) -> T {
                    return .. }, // input
                [=] AMREX_GPU_DEVICE (int i, T const& x) {
                    // x is the sum of inputs from elements
                    // 0 to i-1.
                });
```

- Both input and output are lambda functions, not iterator based. Lambda is arguably much more powerful and convenient than iterators.
- The input can be computed on the fly. It can be simply return p[i], or more complicated.
- The output can be used without writing to the global device memory.
- Used extensively in AMReX particle operations. For example, we use it to implement partition functions.

Kernel Fusion

```
auto a = mfa.arrays();
auto b = mfb.const_arrays();
auto c = mfc.const_arrays();
ParallelFor(mfa, [=] AMREX_GPU_DEVICE (int boxno, int i, int j, int k) {
    a[boxno](i,j,k) = b[boxno](i,j,k) + c[boxno](i,j,k);
});
// A single GPU kernel working on multiple boxes. Box sizes are not necessarily the same.
// Memory is not contiguous across boxes.
// Fused kernel is often 2x - 10x faster depending on the box sizes.
```

```
// Hundreds of small kernels, very slow
for (int m = 0; m < n; ++m) {
  ParallelFor(boxes[m], [=] AMREX_GPU_DEVICE (int i, int j, int k) { … });
}

// Kernel fusion
amrex::Vector<Tag> tags; // Tag can be a user defined type or amrex predefined type
for (int m = 0; m < n; ++m) {
   tags.emplace_back(Tag{boxs[m], …}); // Store information needed for GPU kernel
}
// Launch a single GPU kernel, much faster
ParallelFor(tags, [=] AMREX_GPU_DEVICE (int i, int j, int k, Tag const& tag) { … });
```

Optimization of Kernels with Run Time Parameters

Branches in kernels can be very expensive not just because of thread divergence. If a branch uses a lot of registers, it can significantly affect the performance even if at run time the branch is never executed. One could move the branches out of kernels. But that sometimes results in a lot of code duplication. We provide compile time optimization by using C++17 fold expression to generate codes for all run time variants.

```
int A_runtime_option = ...;
int B_runtime_option = ...;
enum A_options : int { A0, A1, A2, A3};
enum B_options : int { B0, B1 };
// 4*2=8 ParallelFors will be generated.
ParallelFor(TypeList<CompileTimeOptions<A0,A1,A2,A3>,
                     CompileTimeOptions<B0,B1> > {},
            {A_runtime_option, B_runtime_option},
            N, [=] AMREX_GPU_DEVICE (int i, auto A_control, auto B_control)
{
   ...
   if constexpr (A_control.value == A3 && B_control.value == B1) {
       ...
   } else if constexpr (....) {
       ...
   }
   ...
});
```

WarpX PushPX kernel on MI250X

- QED module not always used.
- With the optimization
  - Time: 2.17 -> 1.34
  - NumVgprs: 256 -> 249
  - ScratchSize: 264 -> 144
  - Occupancy: 1 -> 2

Memory Arena

- Memory allocation using cudaMalloc etc. can be quite expensive.
- AMReX provides a number of memory arenas to speed up memory allocation. By default, we preallocate ¾ of the system's global device memory in one big chunk. Subsequent memory allocations from that arena are much faster.
- Vector resize. May not need to move the data even if its capacity is not big enough.
- The use of Arena allows us to implement a memory safety feature.

```
{
    FArrayBox tmp(...);
    async_gpu_kerel_using_tmp(tmp);
    Gpu::synchronize(); // Must sync!
    // otherwise there is a compiler inserted
    // destructor that will free the memory in tmp
    // before the async gpu kernel finishes.
}
```

```
{
    FArrayBox tmp(...,The_Async_Arena());
    async_gpu_kerel_using_tmp(tmp);
} // async safe
```

- The_Arena(): The default arena. Either managed or device.
- The_Async_Arena(): Async safe
- The_Device_Arena(): A separate device arena if The_Arena() is managed, otherwise alias to The_Arena().
- The_Managed_Arena(): A separate managed arena or alias to The_Arena().
- The_Pinned_Arena(): Pinned host memory.
- The_Cpu_Arena(): Pageable host memory.

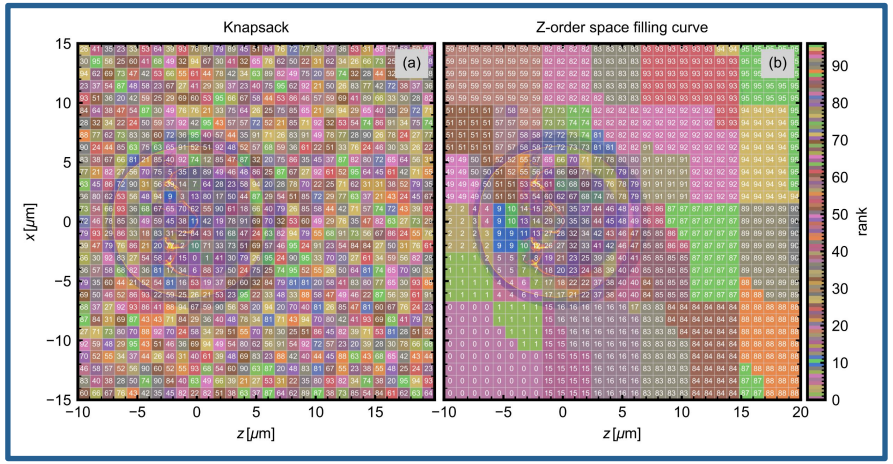
Tools for loading balancing

```
Box domain(IntVect(0), IntVect(127)); // a box with 128^3 cell
BoxArray ba(domain); // Make a new BoxArray w/ one Box
ba.maxSize(64); // Chop into Boxes of 64^3 cells
Print() << ba << "\n";
```

```
(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))
)
```

```
DistributionMapping dm(ba);
```

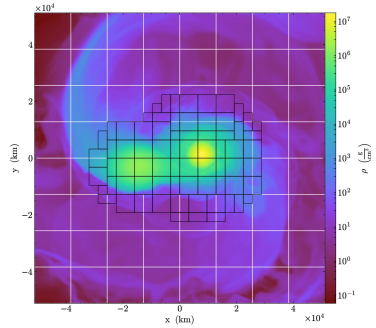
- BoxArray is an array of Boxes on a single AMR level.
- DistributionMapping describes which MPI process owns the data for Boxes in a BoxArray.
- DistributionMapping strategies: Space filling curve, knapsack, round robin, etc.



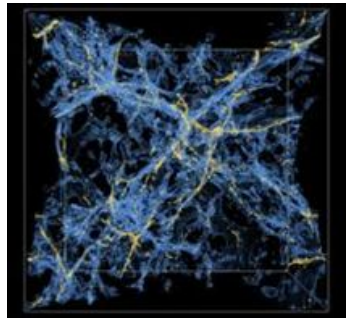
Visualization and I/O

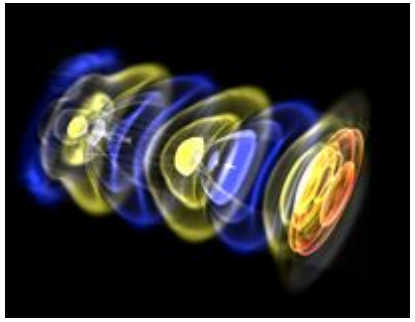
Native I/O format for plotfiles and checkpoint/restart

- multi-level mesh and particle data
- additional app-specific data may be included
- no restriction on number of ranks



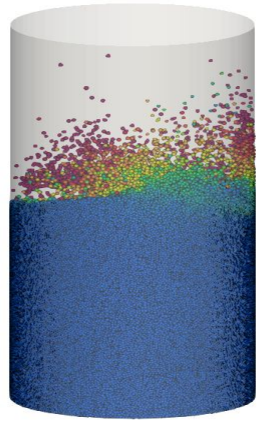
Plotfile format supported by Paraview, Visit, yt, AmrVis





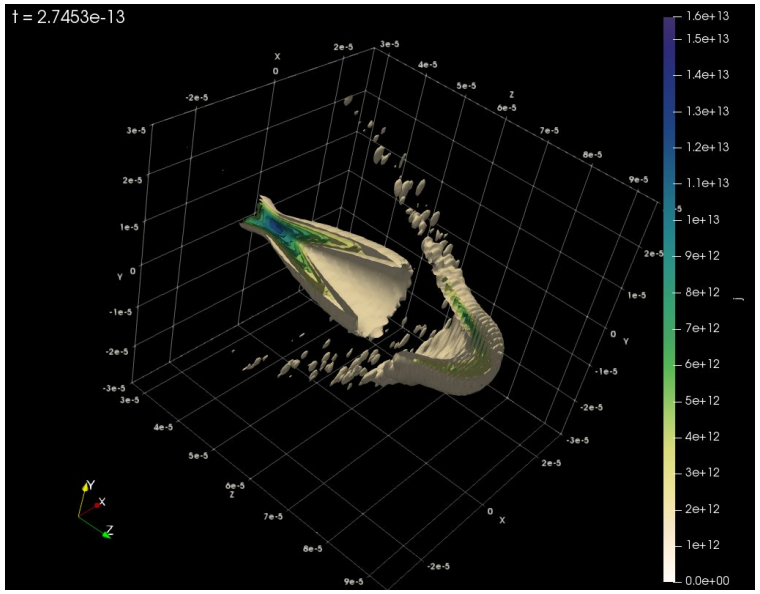
Optional HDF5 output format also supported (has compression)

Async capability developed for CPU/GPU systems

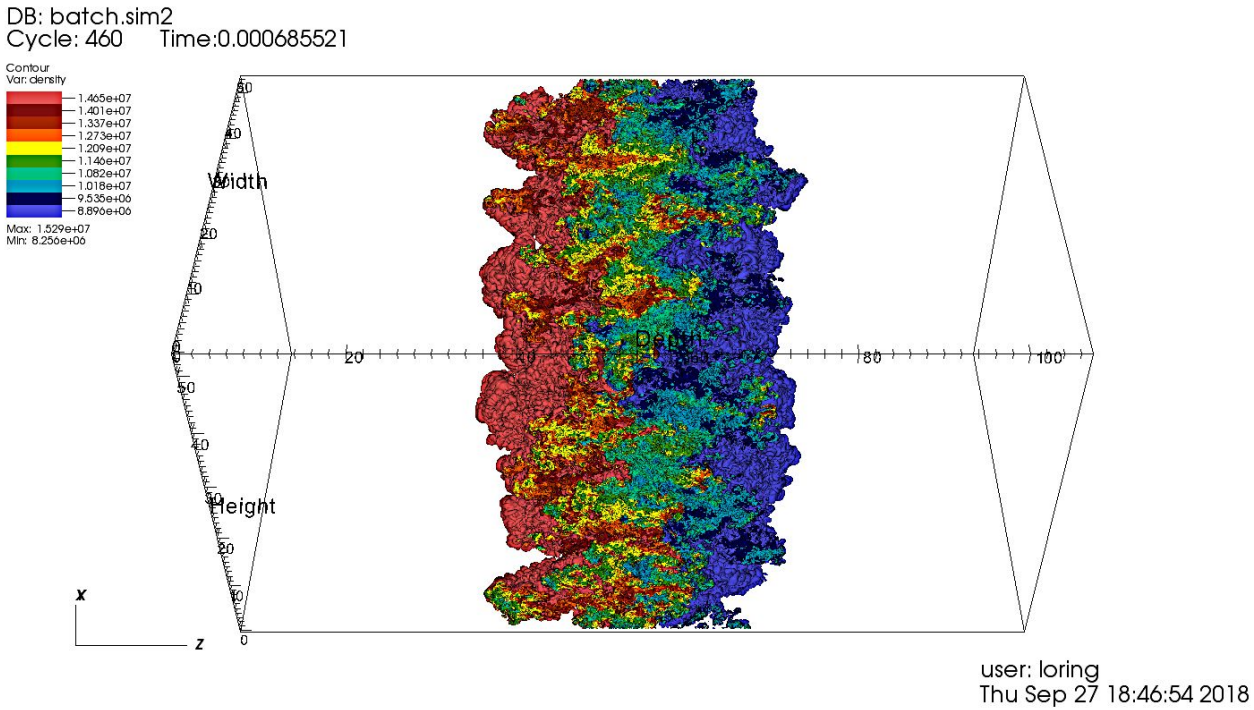


Some applications also interface with ADIOS

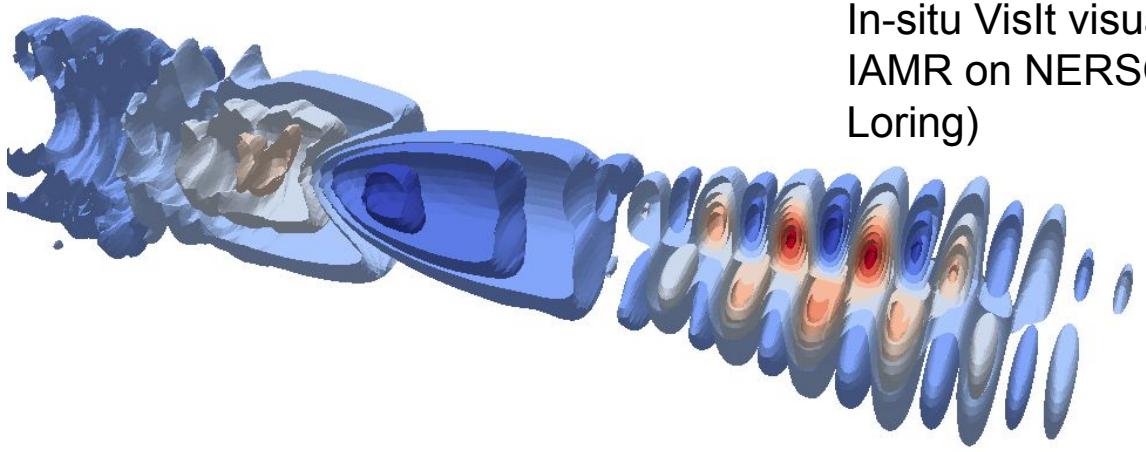
In-situ visualization with Ascent and SENSEI



In-situ visualization of an isocontour of j from WarpX using Paraview



In-situ VisIt visualization of a RT instability computing using IAMR on NERSC Cori with 2048 cores (courtesy of Burlen Loring)



Wakefield acceleration rendering using Ascent / VTKm

You shouldn’t have to use just one package:

Suppose your linear systems are too “hard” for geometric multigrid?

- Call **hypre/petsc** – as a solver for the full equation, or as a “bottom solver” in the GMG hierarchy

Suppose you want to experiment with different time-stepping schemes?

- AMReX is interoperable with **SUNDIALS** (see Time Integration section) – SUNDIALS time integrators understand MultiFABs …

Suppose your equations are too painful to type out in stencil form?

- Use “**CodeGen**” capability through python/sympy AMReX translator to express – in ready-to-compile code – the initial data and right hand side for your time evolution equation

For the hands-on session

Recall what we've seen in our examples…

- "**AMR 101**": AMR for scalar advection
  - Multilevel mesh data – fluid velocity defined on faces and scalar ("ink") defined on cell centers
  - Dynamic AMR
  - Strong scaling behavior and CPU vs GPU ("performance portability")

- "**AMR 102**" : use a Poisson solve to compute incompressible flow around an obstacles and advect the particles in that flow field
  - Single-level mesh data – fluid velocity on faces, EB obstacles defined by volume and area fractions
  - Linear solver (geometric multigrid)
  - Particle advection

- "**AMReX-Pachinko**" is an example of particles falling under gravity through an obstacle course, bouncing off the solid obstacles – here we see an example of particle-obstacle and particle-wall collisions

https://xsdk-project.github.io/MathPackagesTraining2026/lessons/amrex/

Take Away Messages

- Different problems require different spatial discretizations and different data structures – the most common are
  - Structured mesh
  - Unstructured mesh
  - Particles (which can be combined with structured and/or unstructured meshes)

- Structured mesh doesn't equal "just" flow in a box

- There are quite a few AMR software packages – they have several commonalities and a large number of differences, both in what functionality they support and on what architectures they are performant

- Interoperability is important! See the next few sessions for how different packages can be used together.

If you're interested in learning more about AMREX:

- the software: https://www.github.com/AMReX-Codes/amrex
- the documentation: https://amrex-codes.github.io/amrex
- the tutorials: https://amrex-codes.github.io/amrex/tutorials_html/
- some movies based on AMReX: https://amrex-codes.github.io/amrex/gallery.html