BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

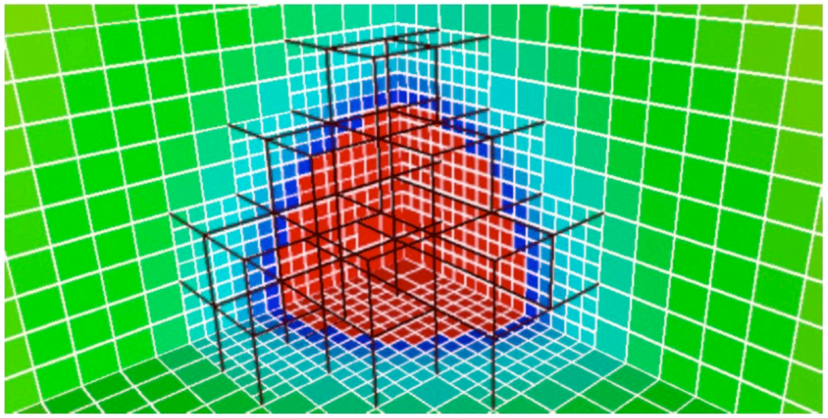
U.S. DEPARTMENT OF ENERGY

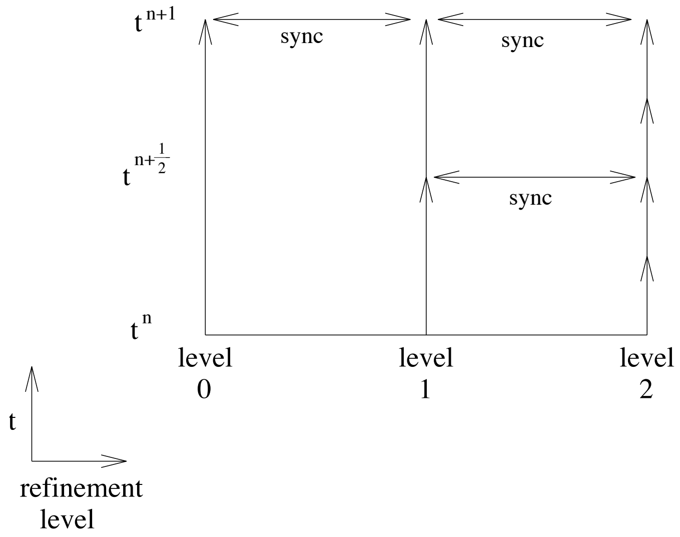
AMR Technologies

Anshu Dubey, Mark Adams,
Ann Almgren, Brian Van Straalen

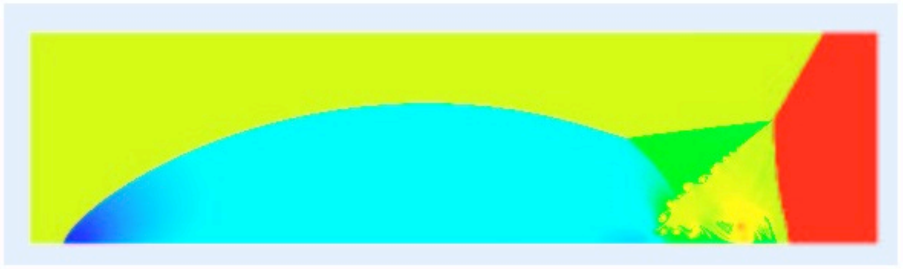
August 3, 2013

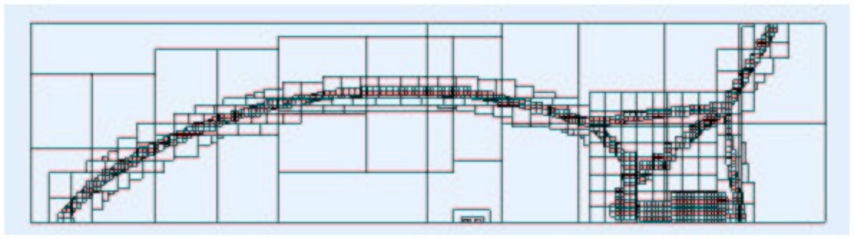
Block-Structured Local Refinement (Berger and Oliger, 1984)





Refined regions are organized into logically-rectangular patches.
Refinement is performed in time as well as in space.





Why use AMR and When ?

- Think of AMR as a compression technique for the discretized mesh

- Apply higher resolution in the domain only where it is needed

- When should you use AMR:
  - When you have a multi-scale problem
  - When a uniformly spaced grid is going to use more memory than you have available to achieve the resolution you need

- You cannot always use AMR even when the above conditions are met
- When should you not use AMR:
  - When the overhead costs start to exceed gains from compression
  - When fine-coarse boundaries compromise the solution accuracy beyond acceptability

Much as using any tool in scientific computing, you should know what are the benefits and limits of the technologies you are planning to use

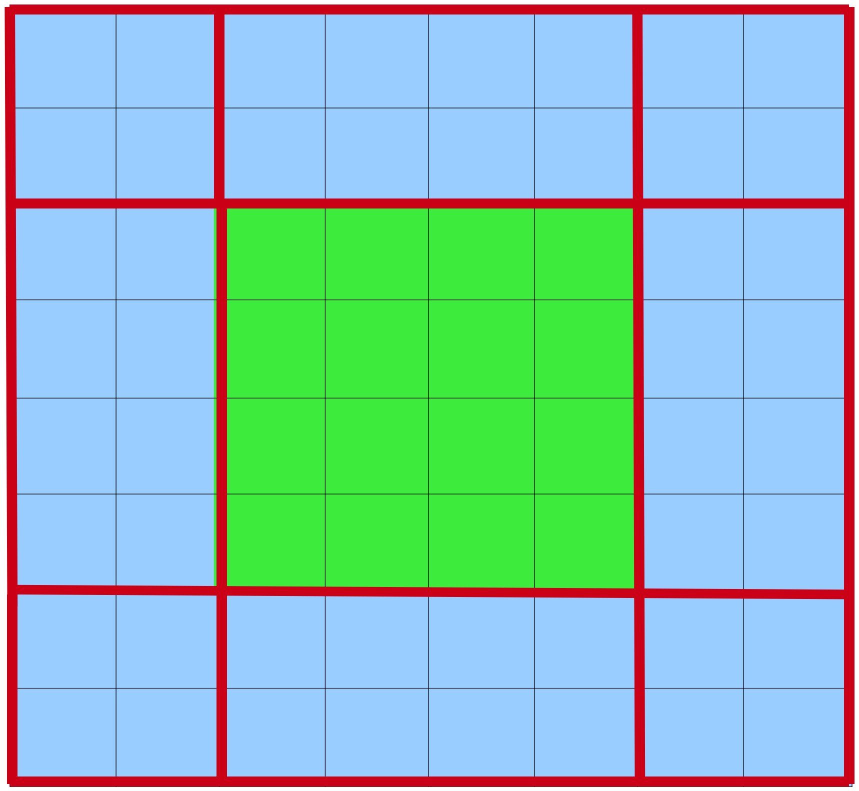
The Flip Side - Complexity

- Machinery needed for computations :
  - Interpolation, coarsening, flux corrections and other needed resolutions at fine-coarse boundaries

- Machinery needed for house keeping :
  - The relationships between entities at the same resolution levels
  - The relationships between entities at different resolution levels

- Machinery needed for parallelization :
  - Domain decomposition and distribution among processors
    - Sometimes conflicting goals of maintaining proximity and load balance
  - Redistribution of computational entities when the grid changes due to refinement
  - Gets more complicated when the solution method moves away from explicit solves

Abstraction for Explicit Methods



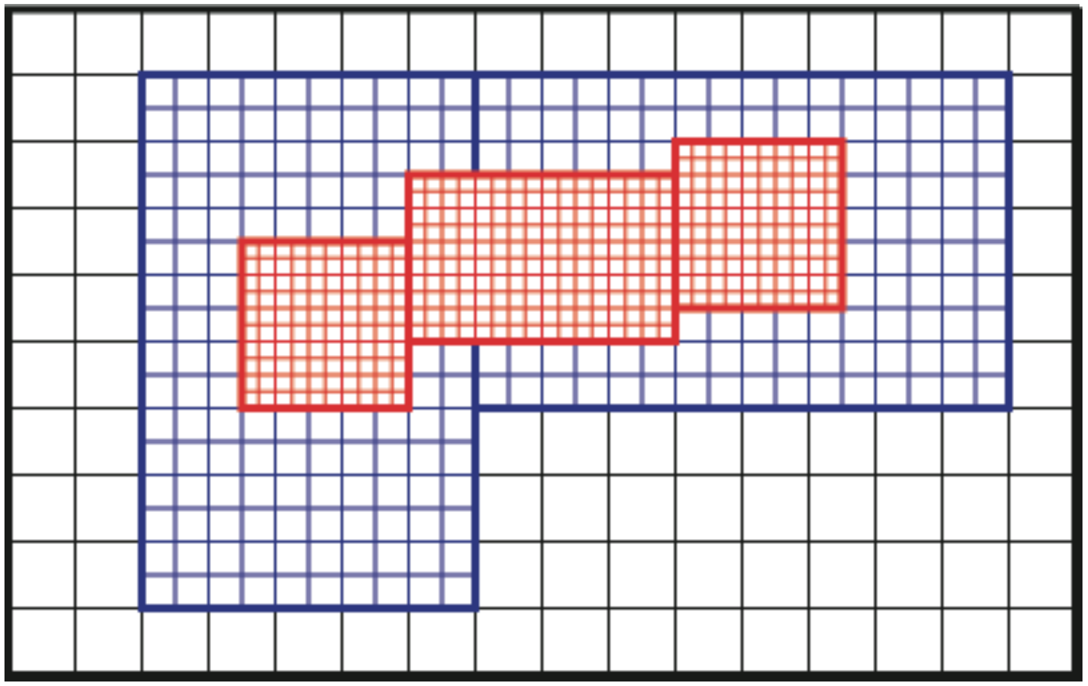
- A self contained computational domain
- Apply computational stencils
- The halo cells may come from same level exchanges or from a coarser level through interpolation
- If there is no sub-cycling, the interface is simple, all patches can get their halos filled simultaneously
- With sub-cycling either the application or the infrastructure can control what to fill when

Most structured AMR methods use the same abstraction for semi-implicit solvers such as multigrid, in the sense they operate on a block/box at a time, the operations in between and the orchestration gets more complicated

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

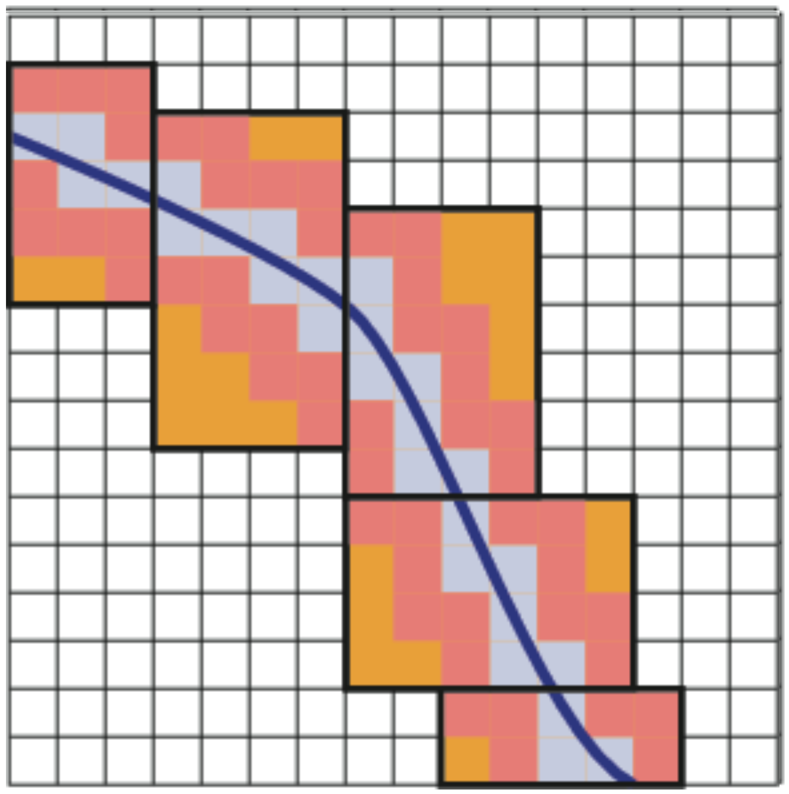
Approach

- Locally refine patches where needed to improve the solution.
- Each patch is a logically rectangular structured grid.
    - Better efficiency of data access.
    - Can amortize overhead of irregular operations over large number of regular operations.
- Refined grids are dynamically created and destroyed.



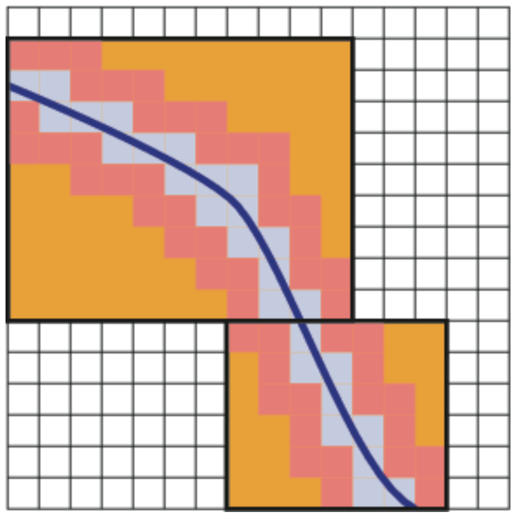
Building the Initial Hierarchy

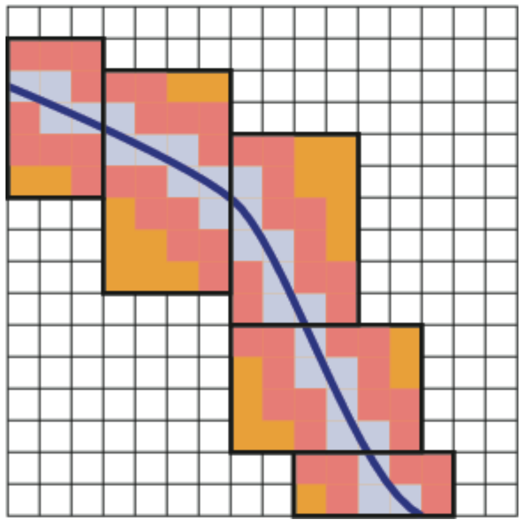
- Fill data at level 0
- Estimate where refinement is needed
- Group cells into patches according to constraints (refinement levels, grid efficiency etc)
- Repeat for the next level
- Maintain proper nesting

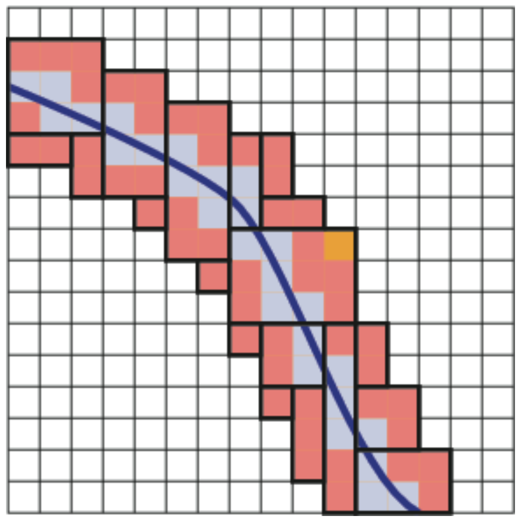


BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

How Efficiency Affects the Grid


Efficiency=0.5


Efficiency=0.7

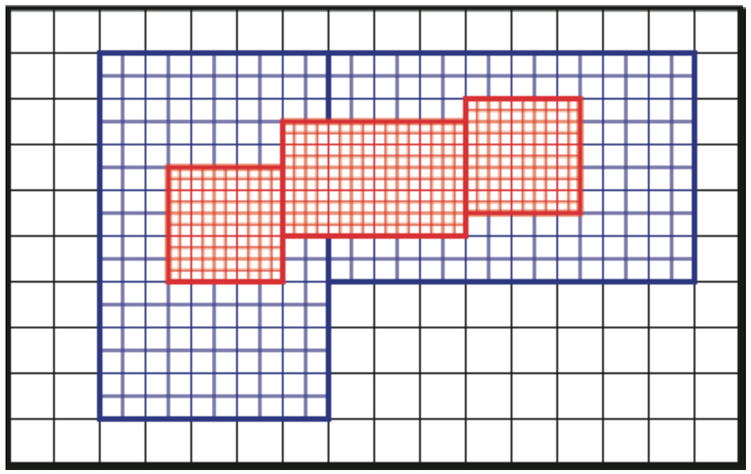

Efficiency=0.9

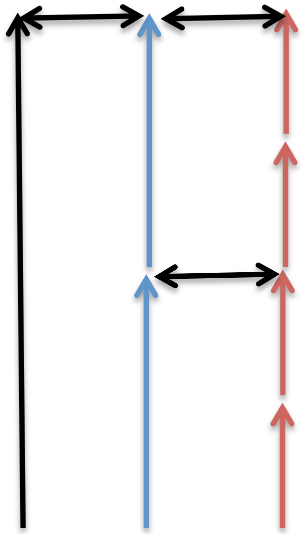
Adaptive in Time

- Consider two levels, coarse and fine with refinement ratio r

$$\Delta x_f = \Delta x_c / r \quad , \quad \Delta t_f = \Delta t_c / r,$$

- Advance $t_c \rightarrow t_c + \Delta t_c$
- Advance fine grids r times
- Synchronize fine and coarse data
- Apply recursively to all refinement levels





The Two Packages: Boxlib and Chombo

•Mixed-language model: C++ for higher-level data structures, Fortran for regular single-grid calculations.

•Reuseable components. Component design based on mapping of mathematical abstractions to classes.

•Build on public-domain standards: MPI.Chombo also uses HDF5

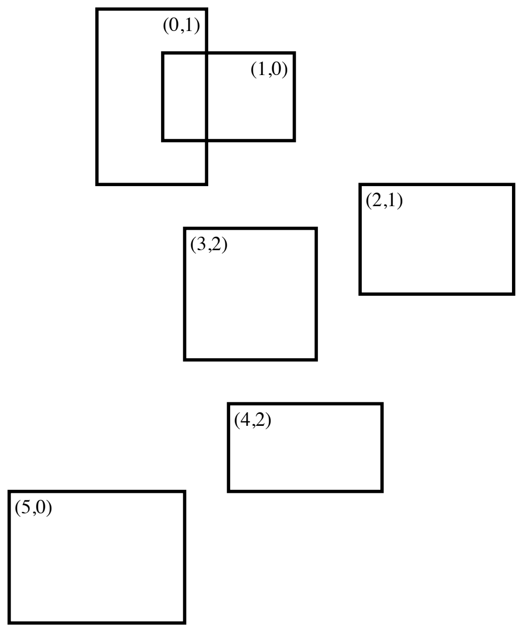
•Interoperability with other tools: VisIt, PETSc,hypre.

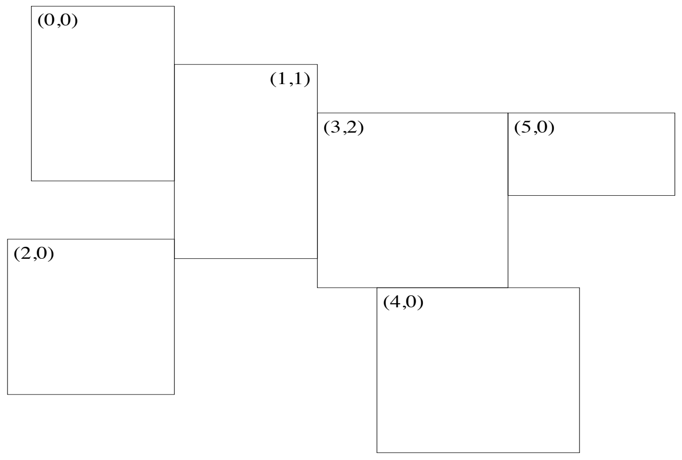
•The lowest levels are very similar – they had the same origin

•Examples from Chombo

Distributed Data on Unions of Rectangles

Provides a general mechanism for distributing data defined on unions of rectangles onto processors, and expressing communications between processors.

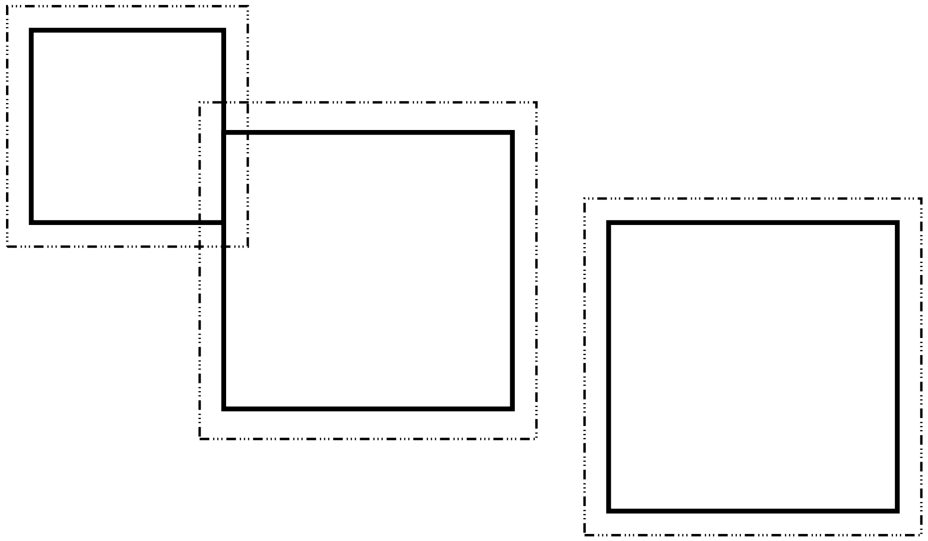




Metadata, of which all processors have a copy. `BoxLayout` is a collection of `Boxes` and processor assignments: $\{B_k,p_k\}_{k=1,ngrids}$.
`DisjointBoxLayout:public Boxlayout` is a `BoxLayout` for which the `Boxes` must be disjoint

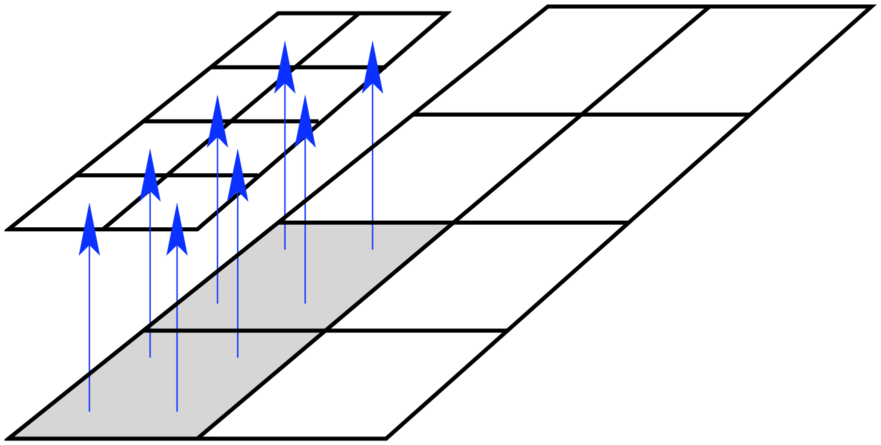
Data on Unions of boxes

Distributed data associated with a `DisjointBoxLayout`. Can have ghost cells around each box to handle intra-level, inter-level, and domain boundary conditions. Templated (LevelData) in Chombo.



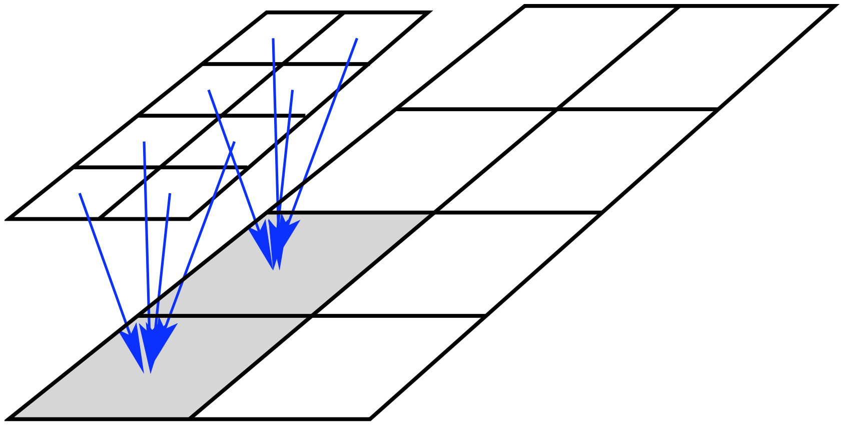
Interpolation from coarse to fine

- Linearly interpolates data from coarse cells to the overlaying fine cells.
- Useful when initializing newly-refined regions after regridding.



`CoarseAverage` Class

- Averages data from finer levels to covered regions in the next coarser level.
- Used for bringing coarse levels into sync with refined grids covering them.



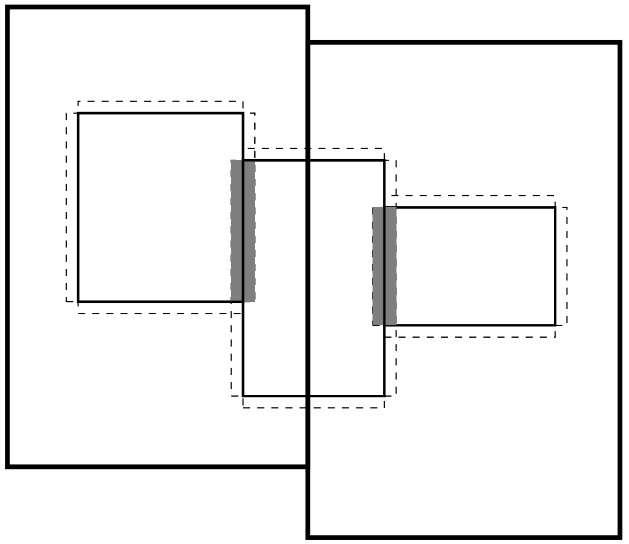
Coarse-Fine Interactions `(AMRTools in Chombo)`

The operations that couple different levels of refinement are among the most difficult to implement, as they typically involve a combination of interprocessor communication and irregular computation.

• Interpolation between levels (`FineInterp`).

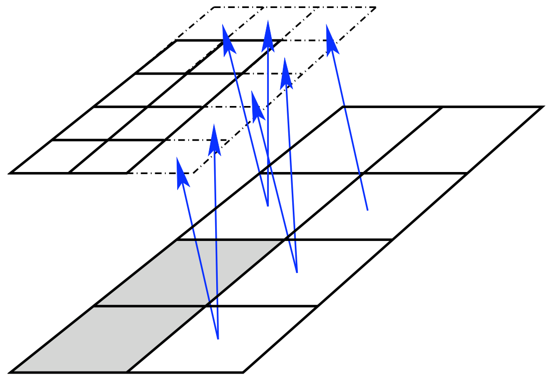
•Averaging down to coarser grids Interpolation of boundary conditions

• Managing conservation at refinement boundaries



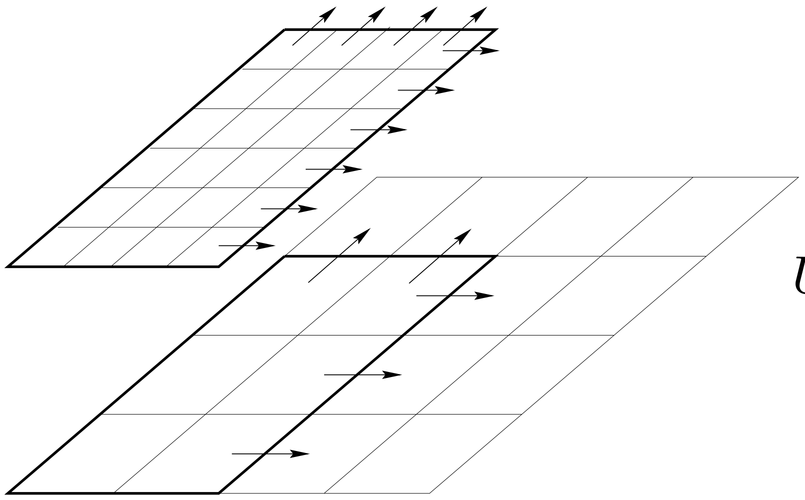
PiecewiseLinearFillPatch Class

Linear interpolation of coarse-level data (in time and space) into fine-level ghost cells.



BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

`LevelFluxRegister` Class



$$U^c := U^c + \Delta t^c \left(F^{c,s}_{\boldsymbol{i}^c - \frac{1}{2}\boldsymbol{e}} - \frac{1}{Z}\sum_{\boldsymbol{i}^f} F^{f,s}_{\boldsymbol{i}^f - \frac{1}{2}\boldsymbol{e}}\right)$$

The coarse and fine fluxes are computed at different points in the program, and on different processors. We rewrite the process in the following steps.

$$\delta F = 0$$

$$\delta F := \delta F - F^c$$

$$\delta F := \delta F + < F^f >$$

$$U^c := U^c + D_R(\delta F)$$

Matrix representation of operators

- We have seen how construct AMR operator as series of sub-operations
  - Coarse interpolation, fine interpolation, boundary conditions, etc.

- Matrix-free operators
  - Low memory: good for performance and memory complexity
  - Can use same technology to construct matrix-free equation solvers
    - Operator inverse
    - Use geometric multigrid (GMG)
    - Inherently somewhat isotropic

- Some applications have complex geometry and/or anisotropy
  - GMG looses efficacy
  - Solution: algebraic multigrid (AMG)

- Need explicit matrix representation of operator
  - Somewhat complex bookkeeping task but pretty mechanical
  - Recently developed infrastructure in Chombo support matrix construction
  - Apply series of transformations to matrix or stencil
    - Similar to operator but operating matrix/stencil instead of field data
  - Stencil: list of <Real weight, <cell, level>>
    - Stencil + map <cell, level> to global equation number: row of matrix
  - Start with A^0 : initial operator matrix
    - Eg, 1D 3-point stencil: {<-1.0, <i-1,lev>, <2.0, <i,lev>, <-1.0, <i+1,lev>}

Approach as matrix transformations

- We can think of these transformations as matrix or operators operating on one global matrix (not a good way to implement)
  - Range and domain space of these operators is critical

- Start with A^0 : initial operator matrix

- B: Boundary conditions for ghost cells off of domain
  - Need one op. for each direction (for corner points)
- C: Interpolate ghost cells on domain (supported by coarse cells)
- F: interpolate cells covered with fine cells
  - F removes covered cells from range and domain:
    Needs two operators F^2 & F^1
    left and right application

    Result: $A := F^2 \bullet A^0 \bullet B \bullet C \bullet F^1$

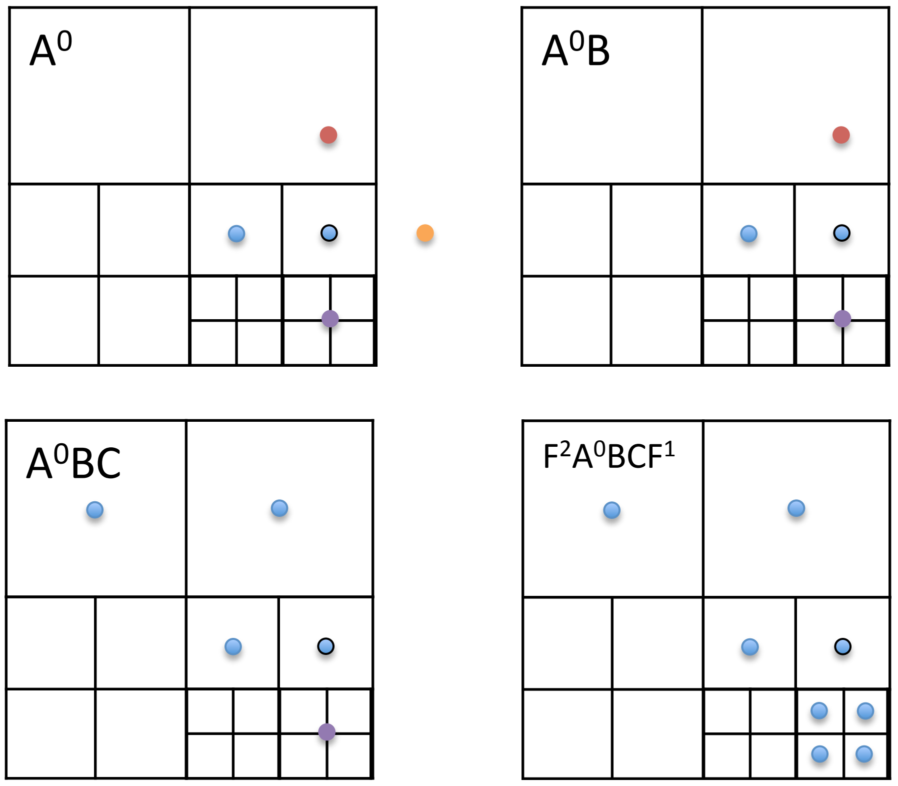
Approach from Stencil view

- Start with raw op stencil A^0, 5-point stencil

- 4 types of cells:
  - Valid (*V*)
    - Real degree of freedom cell in matrix
  - Boundary (*B*)
    - Ghost cell off of domain - BC
  - Coarse (*C*)
    - Ghost cell in domain
  - Fine (*F*)
    - Coarse cell covered by fine

- "raw" operator stencil A^0 composed of all 4 types
  - Transform stencil to have only valid cells

- B, C & F operator have
  - *domain space* with all types (ie, *B*, *C*, *F*)
  - range space w/o its corresponding cell type
    - That is, each operator filters its type
  - Thus after applying B, C & F only valid cells remain
  - Note, F removes *F* cells from range and domain:
    - Needs two operators F^2 & F^1
    - left and right application



Cartoon of stencil for cell ∘ as it is transformed

Example: Laplacian with 3 AMR levels

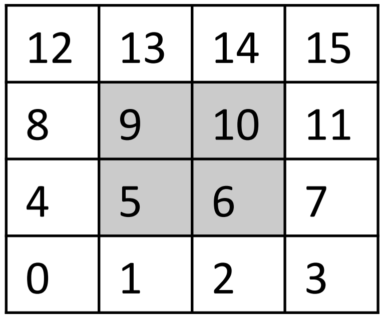
(dx = $6^{1/2}$ on level 1)

(**Invalid nesting region!!!**)

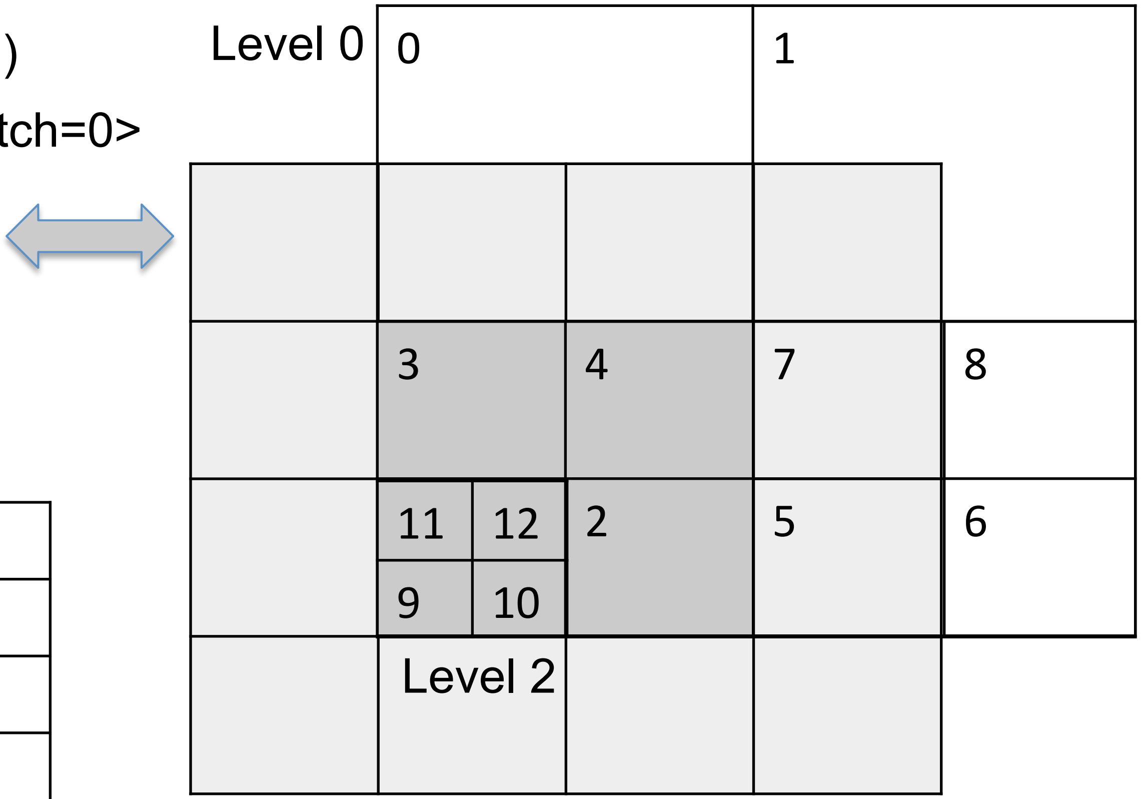
Extended patch <level=1, patch=0>

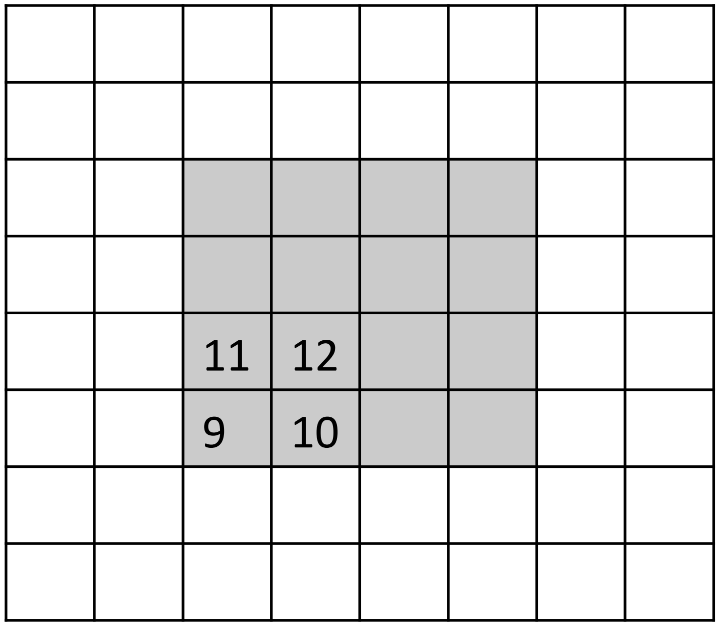
local cell IDs

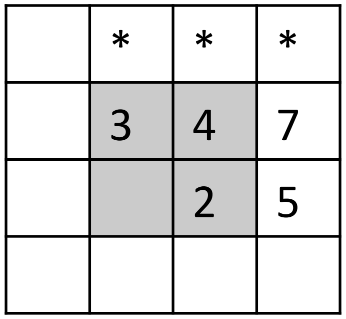
(implicit ordering from box iterators)

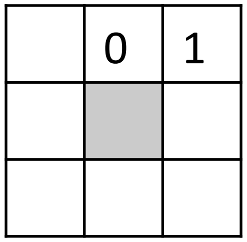


Problem domain – global cell IDs



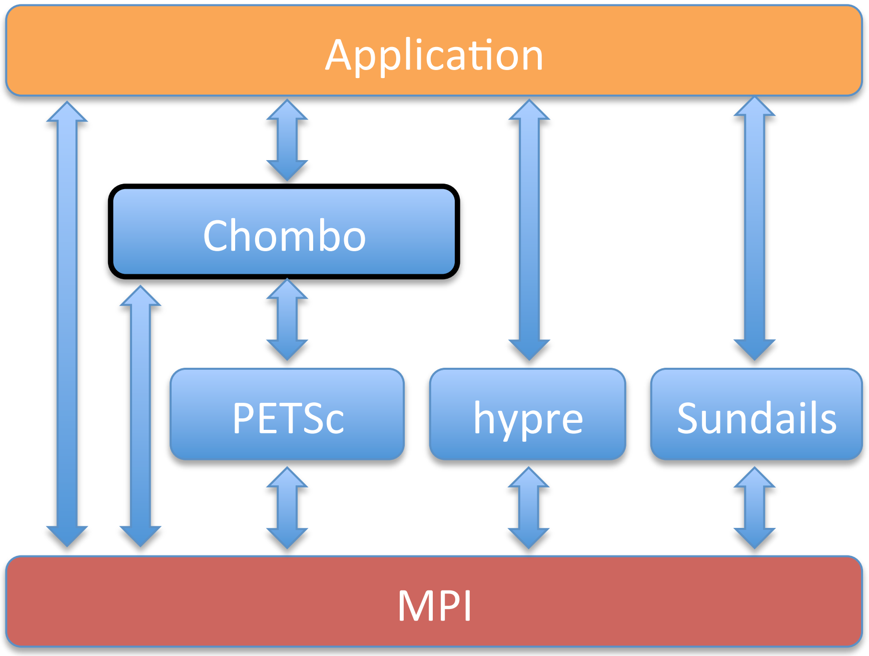






level 2 to global IDs (GIDs) for <1,0> Level 1 GIDs for <1,0> Level 0 GIDs for <1,0>

Getting access to solvers from libraries



Boxlib interoperability with solver libraries would look very similar.

Initial FABMatrix A^0

Local IDs on level 1.
Simplify notation:
eg, 5 == <5,1>

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	-1	-4	-1		-4	20	-4		-1	-4	-1					
6		-1	-4	-1		-4	20	-4		-1	-4	-1				
9					-1	-4	-1		-4	20	-4		-1	-4	-1	
10						-1	-4	-1		-4	20	-4		-1	-4	-1

FABMatrix B
* = -1 for Dirichlet
* = 1 for Neumann.
Use higher order
in practice.
Split operator into
SpaceDim parts
(x & y here).

B^x

	5	6	7	9	10	11	13	14	15	1	2	3
0										*		
1										I_{3x3}		
2		*										
3			*									
4	*											
5 6 7	I_{3x3}											
8				*								
9 10 11				I_{3x3}								
12							*					
13 14 15							I_{3x3}					

B^y

	5	6	7	9	10	11	13	14	15
5	I_{3x3}								
6									
7									
9				I_{3x3}					
10									
11									
13							I_{3x3}		
14									
15									
1	*								
2		*							
3			*						

Coarse interpolation operator C (eg, `QuadCFInterp`)

$$C^1 = \begin{array}{c|ccccccccc}
 & 5 & 6 & 7 & 9 & 10 & 11 & \mathbf{13} & \mathbf{14} & \mathbf{15} \\
\hline
5 & 1 & & & & & & & & \\
6 & & 1 & & & & & & & \\
7 & & & 1 & & & & & & \\
9 & & & & 1 & & & & & \\
10 & & & & & 1 & & & & \\
11 & & & & & & 1 & & & \\
13 & -1/5 & & & 2/3 & & & 8/15 & & \\
14 & & -1/5 & & & 2/3 & & & 8/15 & \\
15 & & & -1/5 & & & 2/3 & & & 8/15
\end{array}$$

$$C^2 = \begin{array}{c|cccccccc}
 & 5 & 6 & 7 & 9 & 10 & 11 & \langle 0,0\rangle & \langle 1,0\rangle \\
\hline
5 & 1 & & & & & & & \\
6 & & 1 & & & & & & \\
7 & & & 1 & & & & & \\
9 & & & & 1 & & & & \\
10 & & & & & 1 & & & \\
11 & & & & & & 1 & & \\
\mathbf{13} & & & & & & & 1.25 & -.25 \\
\mathbf{14} & & & & & & & .75 & .25 \\
\mathbf{15} & & & & & & & .25 & .75
\end{array}$$

* (**bold**) "phiStar" cells live next to ghost cells.
Are not real dofs but just temporary.
Columns are interpolation in
`QuadCFInterp::interpOnIVS.`
N.B. Abuse notation by using <GID,level>.
Code uses <<i,j>,level>

(nontrivial) Columns of C^2 could be sampled by applying `QuadCFInterp::getPhiStar` to (coarse grid) basis vectors.
New off-diagonal elements to coarse grid equations <0,0> & <1,0>.

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

Fine coverer cell interpolation (simple averaging & deleting rows)

$$F^1 = \begin{array}{c|ccccccccccc}
 & 6 & 7 & 9 & 10 & 11 & \langle 0,0\rangle & \langle 1,0\rangle & \langle 9,2\rangle & \langle 10,2\rangle & \langle 11,2\rangle & \langle 12,2\rangle \\
\hline
5 & & & & & & & & 1/4 & 1/4 & 1/4 & 1/4 \\
6 & 1 & & & & & & & & & & \\
7 & & 1 & & & & & & & & & \\
9 & & & 1 & & & & & & & & \\
10 & & & & 1 & & & & & & & \\
11 & & & & & 1 & & & & & & \\
\langle 0,0\rangle & & & & & & 1 & & & & & \\
\langle 1,0\rangle & & & & & & & 1 & & & &
\end{array}$$

$$F^2 = \begin{array}{c|cccc}
 & 5 & 6 & 9 & 10 \\
\hline
6 & & 1 & & \\
9 & & & 1 & \\
10 & & & & 1
\end{array}$$

Deleting cell <5,1> and distributing it to fine grid

ChomboFortran

ChomboFortran is a set of macros used by Chombo for:

- Managing the C++ / Fortran Interface.
- Writing dimension-independent Fortran code.

Advantages to ChomboFortran:

- Enables fast (2D) prototyping, and nearly immediate extension to 3D.
- Simplifies code maintenance and duplication by reducing the replication of dimension-specific code.

Previous C++/Fortran Interface

- C++ call site:

```
heatsub2d_(soln.dataPtr(0),
           &(soln.loVect()[0]), &(soln.hiVect()[0]),
           &(soln.loVect()[1]), &(soln.hiVect()[1]),
             domain.loVect(), domain.hiVect(),
             &dt, &dx, &nu);
```

- Fortran code:

```
      subroutine heatsub2d(phi,iphilo0, iphihi0,iphilo1, iphihi1,
     &                    domboxlo, domboxhi, dt, dx, nu)

      real*8  phi(iphilo0:iphihi0,iphilo1:iphihi1)
      real*8 dt,dx,nu
      integer domboxlo(2),domboxhi(2)
```

Managing such an interface is error-prone and dimensionally dependent (since 3D will have more index arguments for array sizing).

C++ / Fortran Interface with ChomboFortran

- C++ call site:

```
FORT_HEATSUB(CHF_FRA(soln),
             CHF_BOX(domain),
             CHF_REAL(dt), CHF_REAL(dx), CHF_REAL(nu));
```

- Fortran code:

```
subroutine heatsub(CHF_FRA[phi], CHF_BOX[domain],
  &                CHF_REAL[dt], CHF_REAL[dx], CHF_REAL[nu])
```

ChomboFortran expands the argument lists on both sides depending on the dimensionality of the problem. On the Fortran side, it also generates the type declarations for the arguments automatically, along with appropriate header files to be included in the C++ code.

Dimension-independence with ChomboFortran

- Looping macros: `CHF_MULTIDO`
- Array indexing: `CHF_IX`

Replace

```
      do j = nlreg(2), nhreg(2)
         do i = nlreg(1), nhreg(1)
            phi(i,j) = phi(i,j) + nu*dt*lphi(i,j)
         enddo
      enddo
```

with

```
 CHF_MULTIDO[dombox; i;j;k]
        phi(CHF_IX[i;j;k]) = phi(CHF_IX[i;j;k])
     &                     + nu*dt*lphi(CHF_IX[i;j;k])
      CHF_ENDDO
```

Prior to compilation, ChomboFortran replaces the indexing and looping macros with code appropriate to the dimensionality of the problem.

SPMD Parallelism

SPMD = **S**ingle **P**rogram, **M**ultiple **D**ata

- All processors execute the same code.
- Computation can only be done using data that is resident on the processor.
- Communication of data between processors is done by separate, explicit calls to communications libraries (MPI).
- Chombo and Boxlib hide the low-level details of communication through higher-level libraries, and the use of iterators that restrict computation to data that is resident on the processor.

`ProblemDomain` Class

Class that encapsulates the computational domain.

Basic implementation: essentially a `Box` with periodicity information.

In most cases, periodicity is hidden from the user and is handled as a wrapping of the index space.

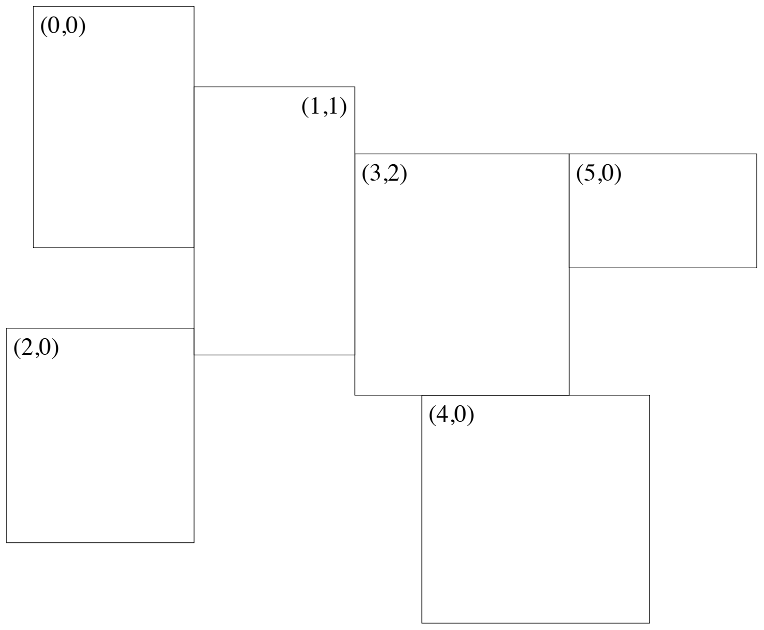
`BoxLayout` Operations

Set of `Boxes` which comprise the valid regions on a single level, including processor assignments for each rectangular region.

Two ways to iterate through a `BoxLayout`

- `LayoutIterator` – iterates through **all** boxes in the `BoxLayout`, regardless of which processor they are assigned to.

- `DataIterator` – iterates only through the boxes on **this** processor.



Example

```
Vector<Box> boxes;       // boxes and processor assignments
Vector<int> procAssign;
ProblemDomain domain;
DisjointBoxLayout dbl(boxes, procAssign, domain);  // define dbl
// access _all_ boxes
LayoutIterator lit = dbl.layoutIterator();
for (lit.begin(); lit.ok(); ++lit)
{
  const Box thisBox = dbl[lit];
}
// access only local boxes
DataIterator dit =   dbl.dataIterator();
for (dit.begin(); dit.ok(); ++dit)

{

  const Box thisLocalBox = dbl[dit];

}
```

Software Reuse by Templating Dataholders

Classes can be parameterized by types, using the class template language feature in C++.

- `BaseFAB<T>` is a multidimensional array for any type `T`.
- `FArrayBox: public BaseFAB<Real>`
- `LevelData<T>, T` can be any type that “looks like” a multidimensional array.
  - Ordinary multidimensional arrays: `LevelData<FArrayBox>`
  - Binsorted lists of particles: `LevelData<BaseFAB<List<ParticleType>>>`
  - Data structures for embedded boundary methods.

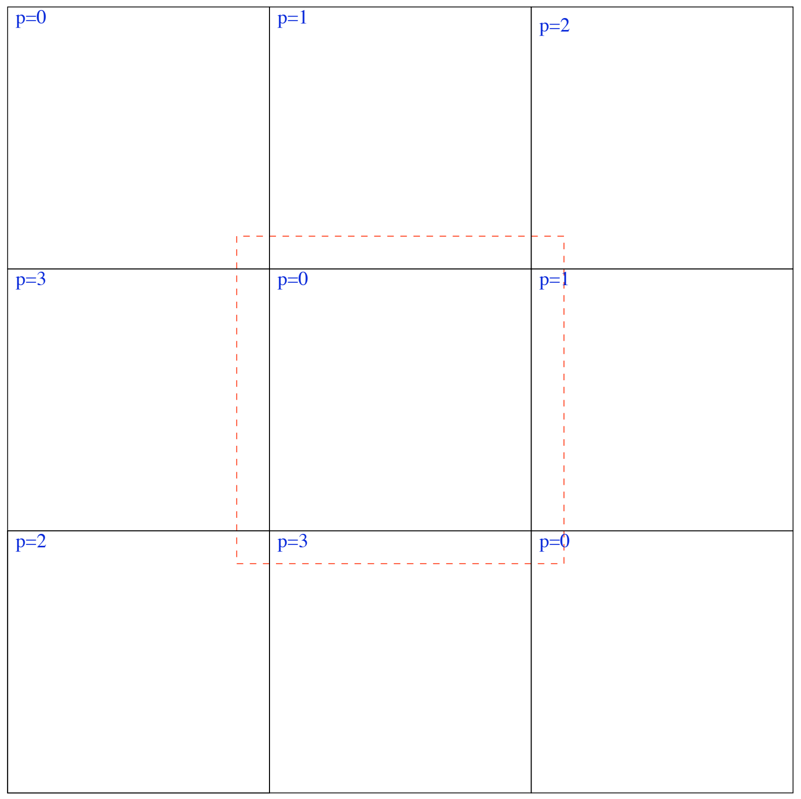
Example

```
DisjointBoxLayout grids;
int nComp = 2;                        // two data components
IntVect ghostVect(IntVect::Unit)      // one layer of ghost cells

LevelData<FArrayBox> ldf(grids, nComp, ghostVect);
                                       // Real distributed data.
LevelData<FArrayBox> ldf2(grids, nComp, ghostVect);

DataIterator dit = grids.dataIterator(); // iterate local data
for (dit.begin(); dit.ok(); ++dit)
{
  FArrayBox& thisFAB = ldf[dit];
  thisFAB.setVal(procID());
}

// fill ghost cells with "valid" data from neighboring grids
ldf.exchange();

ldf.copyTo(ldf2); \\ copy from ldf->ldf2
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

Example Explicit Heat Equation Solver, Parallel Case

Want to apply the same algorithm as before, except that the data for the domain is decomposed into pieces and distributed to processors.



Example Explicit Heat Equation Solver, Parallel Case

```
// C++ code:
  Box domain(IntVect:Zero,(nx-1)*IntVect:Unit);
  DisjointBoxLayout dbl;
// Break domain into blocks, and construct the DisjointBoxLayout.
  makeGrids(domain,dbl,nx);

  LevelData<FArrayBox> phi(dbl, 1, IntVect::TheUnitVector());

  for (int nstep = 0;nstep < 100;nstep++)
  {
…
// Apply one time step of explicit heat solver: fill ghost cell values,
// and apply the operator to data on each of the Boxes owned by this
// processor.

   phi.exchange();
```

Example Explicit Heat Equation Solver, Parallel Case

```
DataIterator dit = dbl.dataIterator();
   for (dit.reset();dit.ok();++dit)
   {
     FArrayBox& soln = phi[dit()];
     Box& region = dbl[dit()];
     FORT_HEATSUB(CHF_FRA(soln),
                  CHF_BOX(region),
                  CHF_BOX(domain),
                  CHF_REAL(dt), CHF_REAL(dx), CHF_REAL(nu));
  }
 }
```

`LevelFluxRegister` Class

A `LevelFluxRegister` object encapsulates these operations.

- `LevelFluxRegister::setToZero()`
- `LevelFluxRegister::incrementCoarse`: given a flux in a direction for one of the patches at the coarse level, increment the flux register for that direction.
- `LevelFluxRegister::incrementFine`: given a flux in a direction for one of the patches at the fine level, increment the flux register with the average of that flux onto the coarser level for that direction.
- `LevelFluxRegister::reflux`: given the data for the entire coarse level, increment the solution with the flux register data for all of the coordinate directions.

Layer 3 Classes: Reusing control structures via inheritance (`AMRTimeDependent, AMRElliptic`)

AMR has multilevel control structures that are largely independent of the operations and data.

• Berger-Oliger timestepping (refinement in time).

• Various linear solver operations, such as multigrid on a single level, multigrid on an AMR hierarchy.

To separate the control structure from the the details of the operation that are being controlled, we use C++ inheritance in the form of *interface classes*.

Elliptic Solver Example: `LinearSolver` virtual base class

```
class LinearSolver<T>
{
// define solver
virtual void define(LinearOp<T>* a_operator, bool
a_homogeneous) = 0;


// Solve L(phi) = rhs
virtual void solve(T& a_phi, const T& a_rhs) = 0;
...
}
```

`LinearOp<T>` defines what it means to evaluate the operator (for example, a Poisson Operator) and other functions associated with that operator. `T` can be an `FArrayBox` (single grid), `LevelData<FArrayBox>` (single-level), `Vector<LevelData<FArrayBox>*>` (AMR hierarchy).

Elliptic Solver Example

`LinearOp` –derived operator classes:

- `AMRPoissonOp:` constant-coefficient Poisson's equation solver on AMR hierarchy $(\alpha\mathbb{I} - \nabla \cdot \beta\nabla)\phi = \rho$.
- `VCAMRPoissonOp:` variable-coefficient elliptic solver on AMR hierarchy.

`LinearSolver` –derived control-structure classes:

- `AMRMultigrid:` use multigrid to solve an elliptic equation on a multilevel hierachy of grids, using composite AMR operators. If base level $\ell_{base} \neq 0$, uses interpolated boundary conditions from coarser level. .
- `BiCGStabSolver:` Use BiConjugate Gradient Stabilized method to solve an elliptic equation (can be single-level, or multilevel). Useful for variable-coefficient problems with strongly-varying coefficients. Also useful as a "bottom solver" for `AMRMultiGrid.`

Factory Classes

Instead of a single `LinearOp`, `AMRMultigrid` needs a set of `AMRPoissonOps` (one for each level in the AMR hierarchy, along with auxiliary levels required by multigrid.

Solution: Factory classes (`AMRPoissonOpFactory`). The factory class contains everything required to define the appropriate operator class at any required spatial resolution.

Define function for the AMR hierarchy.

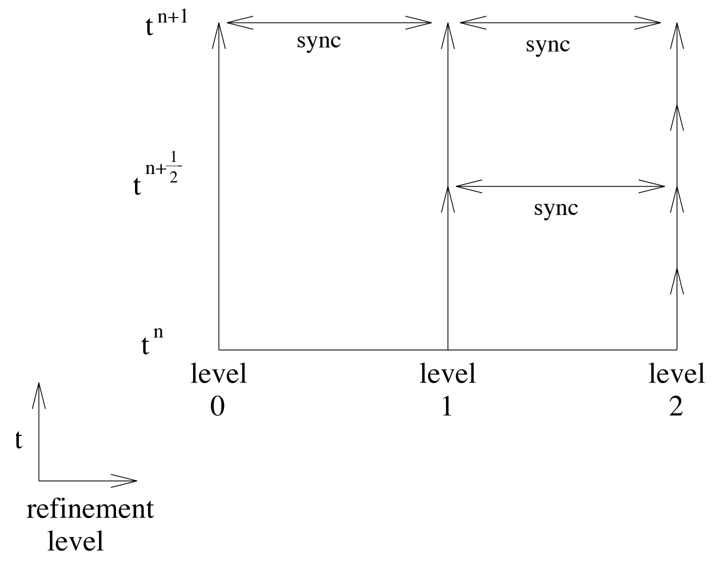
```
 void define(const ProblemDomain& a_coarseDomain,
             const Vector<DisjointBoxLayout>& a_grids,
             const Vector<int>& a_refRatios,
             const Real&        a_coarsedx,
             BCHolder a_bc,
             Real   a_alpha = 0.,
             Real   a_beta  = 1.);
```

`AMRPoissonOp* AMRnewOp(const ProblemDomain& a_indexSpace)` returns an `AMRPoissonOp` defined for the given level.

Example: AMR Poisson solve

```
 int numlevels, baselevel;
  Vector<DisjointBoxLayout> vectGrids;
  Vector<ProblemDomain> vectDomain;
  Vector<int> vectRefRatio;
  Real dxCrse;
  setGrids(vectGrids, vectDomain,dxCrse,
           vectRefRatio, numlevels, baselevel);

  AMRPoissonOpFactory opFactory;
  opFactory.define(vectDomain[0], vectGrids, vectRefRatio,
                   dxCrse, &bcFunc);

  AMRMultiGrid solver;
  solver.define(vectDomain[0], opFactory, &bottomSolver,
numLevels);

  Vector<LevelData<FArrayBox>* > phi(numlevels, NULL);
  Vector<LevelData<FArrayBox>* > rhs(numlevels, NULL);
  defineStorageAndRHS(phi, rhs, vectGrids);

  solver.solveAMR(phi, rhs, numlevels-1, baseLevel);
```

Example: `AMR / AMRLevel` Interface for Berger-Oliger Time Stepping



We implement this control structure using a pair of classes:

- class AMR: manages the timestepping process.
- class AMRLevel : collection of virtual functions called by an AMR object that performs the operations on the data at a level.
    - `virtual void AMRLevel::advance() = 0` advances the data at a level by one time step.
    - `virtual void AMRLevel::postTimeStep() = 0` performs whatever sychronization operations required after all the finer levels have been updated.

`AMR / AMRLevel` Interface

AMR has as member data a collection of pointers to objects of type AMRLevel, one for each level of refinement.

```
Vector<AMRLevel*> m_amrlevels;
```

AMR calls the various member functions of AMRLevel as it advances the solution in time.

```
m_amrlevels[currentLevel]->advance();
```

The user implements a class derived from AMRLevel that contains all of the functions in AMRLevel:

```
class AMRLevelUpwind : public AMRLevel
// Defines functions in the interface, as well as data.
...
virtual void AMRLevelUpwind::advance()
{
// Advances the solution for one time step.
...}
```

To use the AMR class for a particular applications, `m_amrlevel[k]` will point to objects in the derived class

```
AMRLevelUpwind* amrLevelUpwindPtr = new AMRLevelUpwind(...);
m_amrlevel[k] = static_cast <AMRLevel*> (amrLevelUpwindPtr);
```

Upwind Advection Solver

Simple constant-velocity advection equation:

$$\frac{\partial U}{\partial t} + \vec{A}\nabla \cdot U = 0$$

Discretize on AMR grid using simple 1st-order upwind approach. Piecewise-linear interpolation in space for coarse-fine boundary conditions. .

Refinement in time: linear interpolation in time for coarse-fine boundary conditions. U is a conserved quantity, maintain conservation at coarse-fine interface using refluxing.

Upwind Advection Solver

- `AMRLevelUpwind: public AMRLevel` class derived from base `AMRLevel` class, fills in specific functionality for implementing the upwind advection algorithm.
    - `advance()` – advance a single AMR level by one time step using first-order upwind. Initialize / increment flux registers as appropriate.
    - `postTimestep()` – synchronization operations: averaging next finer level onto covered region; refluxing.
    - `tagcells(IntVectSet& tags)` – specify which cells on a given level will be tagged for refinement.
    - `regrid(const Vector<Box>& newGrids)` - given a new grid configuration for this level, re-initialize data.
    - `initialData()` - initialize data at the start of the computation.
    - `computeDt()` - compute the maximum allowable timestep based on the solution on this level

Upwind Advection Solver

- `AMRUpwindLevelFactory: public AMRLevelFactory` class derived from base `AMRLevel` class, derived from base `AMRLevelFactory` class. Used by AMR to define a new `AMRLevelUpwind` object.
  - `virtual AMRLevel* new_amrlevel` – returns a pointer to a new `AMRLevelUpwind` object.
  - `postTimestep()` – Can also be used to pass information through to all `AMRLevelUpwind` objects in a consistent manner (advection velocity, CFL number,…)

Sample Main Program

```
 // Set up the AMRLevel... factory
AMRLevelUpwindFactory amrLevelFact;
amrLevelFact.CFL(cfl);
amrLevelFact.advectionVel(advection_velocity);

AMR amr;

// Set up the AMR object with AMRLevelUpwindFactory
amr.define(maxLevel,refRatios,probDomain,&amrLevelFact);

// initialize hierarchy of levels from scratch for AMR run
amr.setupForNewAMRRun();

amr.run(stopTime,nstop);

amr.conclude();
```

AMRLevelUpwind::advance()

```
Real AMRLevelUpwind::advance()
{
  // Copy the new to the old
  m_UNew.copyTo(m_UOld);

  // fill in ghost cells, if necessary
  AMRLevelUpwind* coarserLevelPtr = NULL;
  // interpolate from coarser level, if appropriate
  if (m_level > 0)
  {
    coarserLevelPtr = getCoarserLevel();

    // get old and new coarse-level data
    LevelData<FArrayBox>& crseDataOld = coarserLevelPtr->m_UOld;
    LevelData<FArrayBox>& crseDataNew = coarserLevelPtr->m_UNew;
    const DisjointBoxLayout& crseGrids = crseDataNew.getBoxes();

    Real newCrseTime = coarserLevelPtr->m_time;
    Real oldCrseTime = newCrseTime - coarserLevelPtr->m_dt;
    Real coeff = (m_time - oldCrseTime)/coarserLevelPtr->m_dt;
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

AMRLevelUpwind::advance()

```
   const ProblemDomain& crseDomain = coarserLevelPtr->
  m_problem_domain;
   int nRefCrse = coarserLevelPtr->refRatio();
   int nGhost = 1;

   PiecewiseLinearFillPatch filpatcher(m_grids, crseGrids,
                                       m_UNew.nComp(),
                                        crseDomain,
                                       nRefCrse, nGhost);


   filpatcher.fillInterp(m_UOld, crseDataOld, crseDataNew,
                         coeff, 0, 0, m_UNew.nComp());

  }
// exchange copies overlapping ghost cells on this level
  m_UOld.exchange();
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

AMRLevelUpwind::advance()

```
// now go patch-by-patch, compute upwind flux, and do update
// iterator will only reference patches on this processor
for (dit.begin(); dit.ok(); ++dit)
{
  const Box gridBox = m_grids.get(dit());
  FArrayBox& thisOldSoln = m_UOld[dit];
  FArrayBox& thisNewSoln = m_UNew[dit];
  FluxBox fluxes(gridBox, thisOldSoln.nComp());

  // loop over directions
  for (int dir=0; dir<SpaceDim; dir++)
  {
    // note that gridbox will be the face-centered one
    Box faceBox = fluxes[dir].box();
    FORT_UPWIND(CHF_FRA(fluxes[dir]),
                CHF_FRA(thisOldSoln),
                CHF_REALVECT(m_advectionVel),
                CHF_REAL(m_dt),
                CHF_REAL(m_dx),
                CHF_BOX(faceBox),
                CHF_INT(dir));
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

AMRLevelUpwind::advance()

```
 // increment flux registers with fluxes
      Interval UInterval = m_UNew.interval();

      if (m_hasFiner)
      {
 // this level's FR goes between this level and the next finer
        m_fluxRegister.incrementCoarse(fluxes[dir], m_dt, dit(),
                                       UInterval, UInterval, dir);
      }
      if (m_level > 0)
      {
        LevelFluxRegister& crseFluxReg = coarserLevelPtr->
          m_fluxRegister;

        crseFluxReg.incrementFine(fluxes[dir], m_dt, dit(),
                                  UInterval, UInterval, dir,
Side::Lo);
        crseFluxReg.incrementFine(fluxes[dir], m_dt, dit(),
                      UInterval, UInterval, dir, Side::Hi);
      }
    } // end loop over directions
```

AMRLevelUpwind::advance()

```
    // do flux difference to increment solution
    thisNewSoln.copy(thisOldSoln);

    for (int dir=0; dir<SpaceDim; dir++)
    {
      FORT_INCREMENTDIVDIR(CHF_FRA(thisNewSoln),
                           CHF_FRA(fluxes[dir]),
                           CHF_BOX(gridBox),
                           CHF_REAL(m_dx),
                           CHF_REAL(m_dt),
                           CHF_INT(dir));
    }
  } // end loop over grid boxes

  // Update the time and store the new timestep
  m_time += m_dt;
  return m_dt;
```

AMRLevelUpwind::postTimeStep()

```
    void AMRLevelUpwind::postTimeStep()
{
  if (m_hasFiner)
  {
    // Reflux
    Real scale = -1.0/m_dx;
    m_fluxRegister.reflux(m_UNew,scale);

    // Average from finer level data
    AMRLevelUpwind* finerLevelPtr = getFinerLevel();
    LevelData<FArrayBox>& fineU = finerLevelPtr->m_UNew;

    finerLevelPtr->m_coarseAverage.averageToCoarse(m_UNew,
                                                   fineU);
  }
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

Outline

1. Introduction
2. Layered Interface
3. Layer 1: `BoxTools`
    - Example: explicit heat solver on a single grid.
4. Chombo Fortran
5. Layer 1, cont’d
    - `DisjointBoxLayout`
    - `LevelData`
    - Parallel heat solver example
6. Layer 2: `AMRTools`
7. Layer 3: `AMRElliptic, AMRTimeDependent`
    - Example: Advection Solver
8. Layer 4: AMR Applications
9. Utilities

FAQ

•Chombo -- Swahili word meaning ``box''', ``container'', or ``useful thing'’

• Freely available, subject to export controls and BSD-like license agreement

•Requirements: C++, Fortran compilers, PERL, HDF5 (for I/O), MPI

•Supports different precisions (float, double) through use of {\tt Real} data type.

•Online doxygen documentation: http://davis.lbl.gov/Manuals/CHOMBO-CVS

•E-mail support: chombo@anag.lbl.gov

•Chombo users e-mail group: *chombousers@hpcrd.lbl.gov*

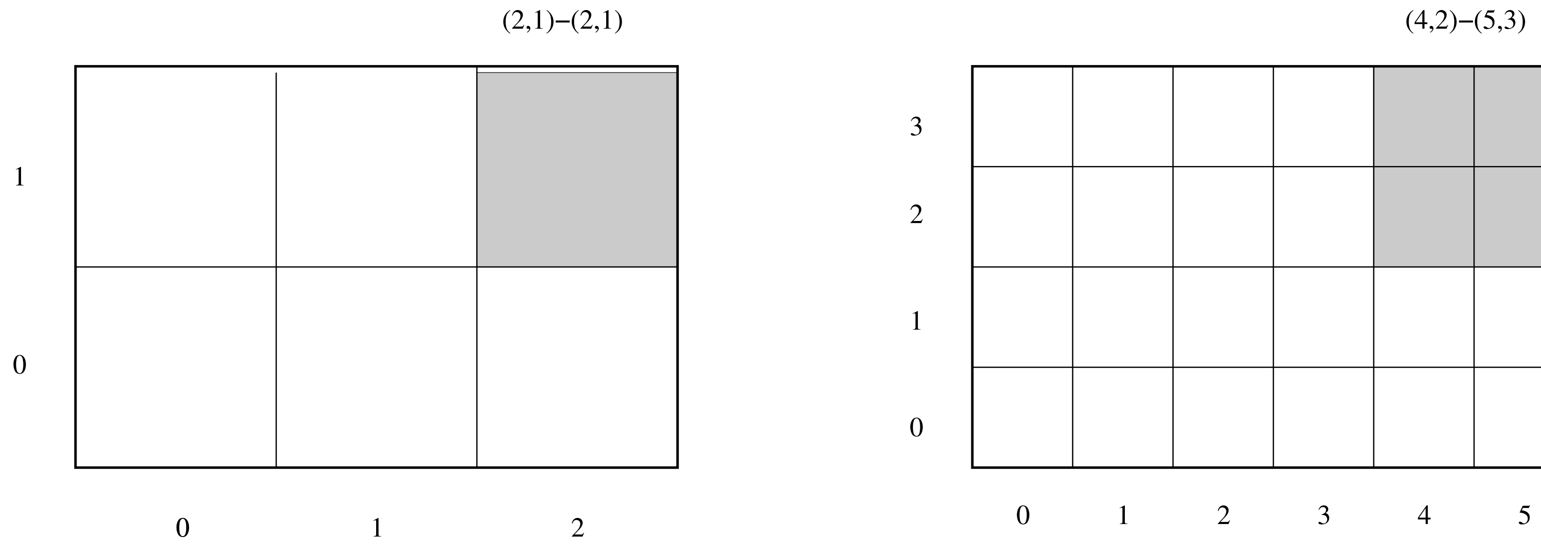
Lowest Layer

Global index spaces:

• Each AMR level uses a global index space to locate points in space.

•Each level's index space is related to the others by simple coarsening and refinement operations.

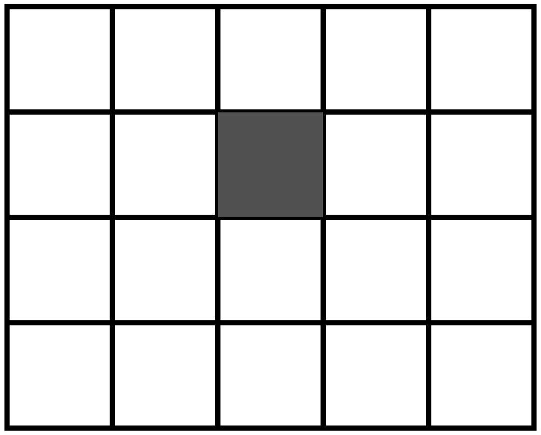
•Makes it easy to organize interlevel operations and to perform operations on unions of rectangular grids at a level.



IntVect Class

Location in index space: $\boldsymbol{i} \in \mathbb{Z}^d$

Can translate $\boldsymbol{i_1} \pm \boldsymbol{i_2}$,coarsen $\frac{\boldsymbol{i}}{s}$, refine $\boldsymbol{i} * s$.



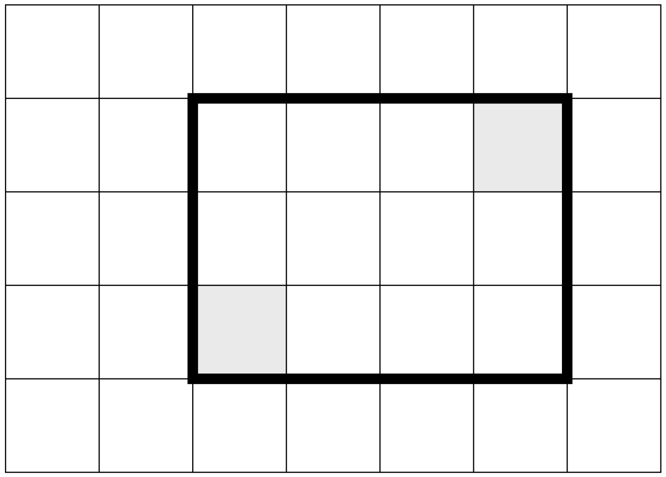
2D Example:

```
IntVect iv(2,3);    \\ create IntVect
iv *= 2;            \\ multiply by a factor of 2 (now (4,6)
iv.coarsen(2);      \\ coarsen by factor of 2 (now 2,3)

IntVect iv2(1,2);   \\ second IntVect
iv += iv2;          \\ add iv2 to iv -- iv = (3,5)

int i = iv[0];      \\ access 0th component (i = 3)
```

Box Class

$B \subset \mathbb{Z}^d$ is a rectangle in space: $B = [\boldsymbol{i}_{low}, \boldsymbol{i}_{high}]$

B can be translated, coarsened , refined.
Supports different centerings (node-centered vs. face-centered) in each coordinate direction.



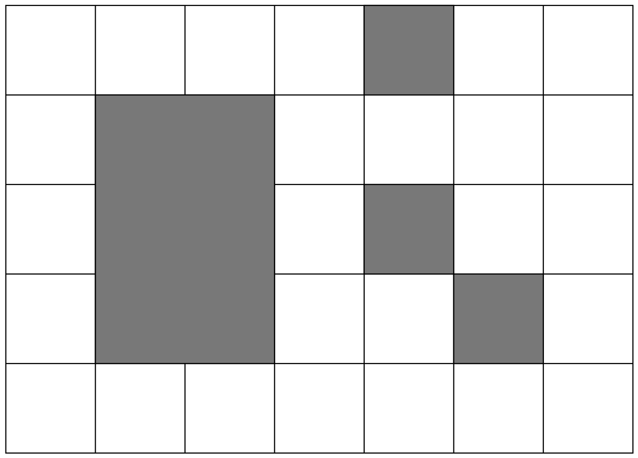
2D Example:

```
IntVect lo(1,1), hi(2,3);\\ IntVects to define box extents
Box b(lo,hi);            \\ define cell-centered box
b.refine(2);             \\ refine by factor of 2: now (2,2)-(5,7)
b.coarsen(2);            \\ coarsen: now back to (1,1)-(2,3)
b.surroundingNodes()     \\ convert to node-centering -- (1,1)-(3,4)
b.enclosedCells()        \\ back to cell-centering -- (1,1)-(2,3)

Box b2(b)                \\ copy constructor
b2.shift(IntVect::Unit); \\ shift b2 by (1,1) -- now (2,2)-(3,4)
b &= b2 ;                \\ intersect b with b2 -- b now (2,2)-(2,3)
```

`IntVectSet` Class : Specific to Chombo

$\mathcal{I} \subset \mathbb{Z}^d$ is an arbitrary subset of $\mathbb{Z}^d$. $\mathcal{I}$ Can be shifted, coarsened, refined.

One can take unions and intersections with other `IntVectSets` and with `Boxes`, and iterate over an `IntVectSet.` Useful for representing irregular sets and calculations over such sets.



2D Example:

```
IntVect iv1, iv2, iv3; \\ various IntVects
Box b;                 \\ box region
IntVectSet ivSet(b)    \\ set of all Index locations in b
ivSet |= iv1;          \\ union operator
ivSet.refine(2);       \\ refinement of IntVectSet
ivSet.coarsen(2);      \\ coarsen back to original
ivSet -= iv2;          \\ remove iv2 from intVectSet
ivSet.grow(1);         \\ grow intVectSet by a radius of 1
```

BaseFAB<T> Container Class

Templated muiltidimensional array container class for (`int, Real`, etc) over region defined by a `Box`.

2D Example:

```
Box domain(IntVect::Zero, 3*IntVect::Unit); // box from (0,0)->(3,3)
int nComp = 2;                              // 2 components
BaseFab<int> intfab(domain, nComp);         // define container for int's
intfab.setVal(1);                           // set values to 1

BoxIterator bit(domain);                    // iterator over domain
for (bit.begin(); bit.ok(); ++bit)
{
   iv = bit();
   ival(iv,0) = iv[0];                      // set 0th component at
}                                           // iv to index
int* ptr = ifab.dataPtr();                  // pointer to the contiguous
                                            // block of data which can
                                             // be passed to Fortran.
```

`FArrayBox` Class

Specialized `BaseFab<T>` with additional floating-point operations – can add, multiply, divide, compute norms.

Example:

```
Box domain;
FArrayBox afab(domain, 1);  // define single-component FAB's
FArrayBox bfab(domain, 1);

afab.setVal(1.0);          // afab = 1 everywhere
bfab.setVal(2.0);          // set bfab to be 2

afab.mult(2.0);            // multiply all values in afab by 2
afab.plus(bfab);           // add bfab to afab (modifying afab)
```

If any boxlib specific detail needed

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY