Block-Structured Adaptive Mesh Refinement Algorithms and Software

Phillip Colella
Lawrence Berkeley National Laboratory

Adaptive Mesh Refinement (AMR)

Modified equation analysis: finite difference solutions to partial differential equations behave like solutions to the original equations with a modified right-hand side.

For linear steady-state problems $LU = f$:

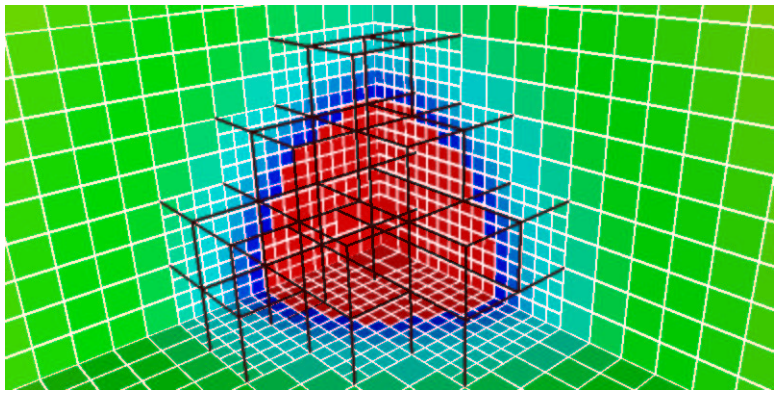
$$\epsilon = U^h - U \quad , \quad \epsilon \approx L^{-1}\tau$$

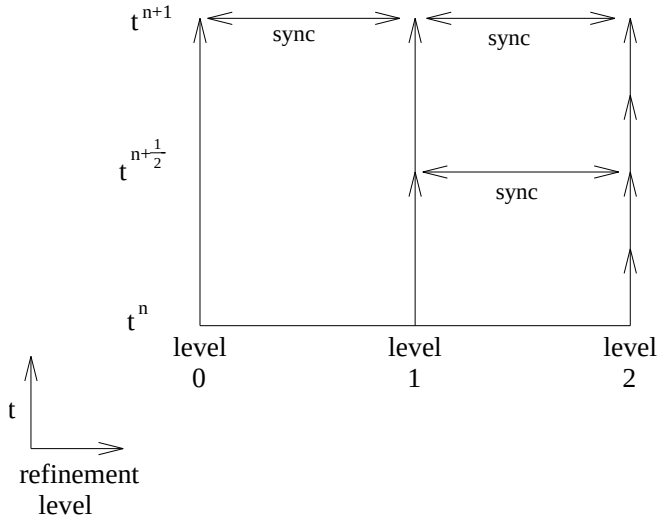
For nonlinear, time=dependent problems

$$\frac{\partial U}{\partial t} + L(U) = 0 \Rightarrow \frac{\partial U^h}{\partial t} + L(U^h) = \tau$$

In both cases, The truncation error $\tau = \tau(U) = (\Delta x)^p M(U)$, where M is a $(p+q)$-order differential operator.

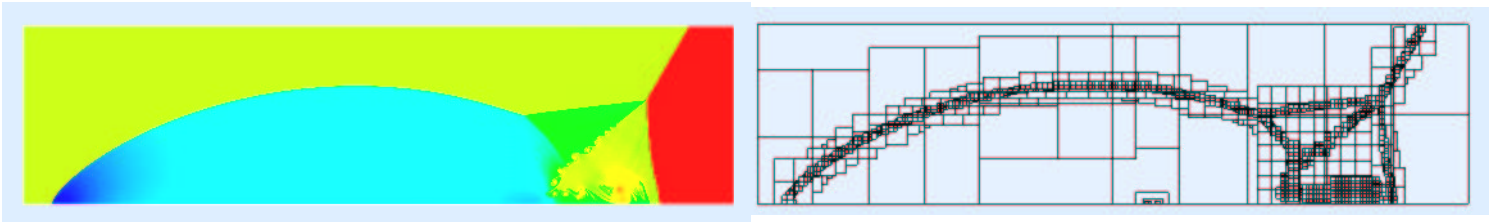
Block-Structured Local Refinement (Berger and Oliger, 1984)





Refined regions are organized into rectangular patches

Refinement performed in time as well as in space.



Why a Framework ?

Variety of problems that exhibit multiscale behavior.

- Shocks and interfaces.
- Self-gravitating flows in astrophysics.
- Complex engineering geometries.
- Combustion.
- Magnetohydrodynamics: space weather, magnetic fusion.
- Biological modeling (systems biology, bio-fluids, hemodynamics).

All of these problems as described by the same types of classical PDE, but in different combinations. Want to maximize reuse across applications.

We need to identify the mathematical components that make up the algorithm space.

Equations of Classical Mathematical Physics

Hyperbolic:

$$\frac{\partial U}{\partial t} + A\frac{\partial U}{\partial x} = f \text{ , } A \text{ diagonalizable with real eigenvalues}$$

Explicit methods, $\Delta t = O(\Delta x)$ required for both stability and accuracy.

Elliptic:

$$\Delta\phi = f \text{ , } \Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$$

Local elliptic regularity leads to well-behaved linear systems.

Parabolic:

$$\frac{\partial T}{\partial t} = \Delta T + f$$

Implicit discretization in time leads to elliptic problems.

All of the above problems lead to algorithmically scalable computational problems, with $O(1)$ floating point operations per word of data generated.

To treat more complex problems, we

• Decompose them into pieces, each one of which is well-understood, and between which the coupling is not too strong;

• Use numerical methods based on our understanding of the components, coupled together using predictor-corrector methods in time.

Example: Incompressible Navier-Stokes equations

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \nabla p = \nu \Delta \vec{u}$$
$$\nabla \cdot \vec{u} = 0$$

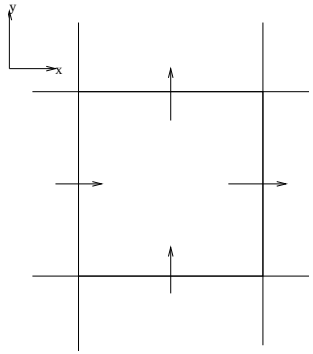
These equations can be splitting into three pieces:

$$\begin{aligned}
&\text{Hyperbolic:} && \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = 0 \\
&\text{Parabolic:} && \frac{\partial \vec{u}}{\partial t} = \nu \Delta \vec{u} \\
&\text{Elliptic:} && \Delta p = \nabla \cdot (-\vec{u} \cdot \nabla \vec{u} + \nu \Delta \vec{u})
\end{aligned}$$

Conservation Form

Our spatial discretizations are based on conservative finite difference approximations.

$$\nabla \cdot \vec{F} \approx D(\vec{F}) \equiv \frac{1}{\Delta x} \sum_{s=1}^{d} \left(F^s_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s} - F^s_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^s} \right)$$



Such methods satisfy a discrete form of the divergence theorem:

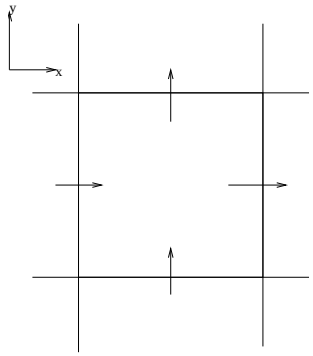
$$\sum_{\boldsymbol{j}\in\Omega} D(F)_{\boldsymbol{j}} = \frac{1}{\Delta x} \sum_{\boldsymbol{j}+\frac{1}{2}\boldsymbol{e}^s \in \partial\Omega} \pm F^s_{\boldsymbol{j}+\frac{1}{2}\boldsymbol{e}^s}$$

This class of discretizations essential for discontinuities, desirable for a large class of engineering applications.

AMR for Hyperbolic Conservation Laws (Berger and Colella, 1989)

We assume that the underlying uniform-grid method is an explicit conservative difference method.

$$U^{new} := U^{old} - \Delta t(D\vec{F}) \ , \ \ \vec{F} = \vec{F}(U^{old})$$

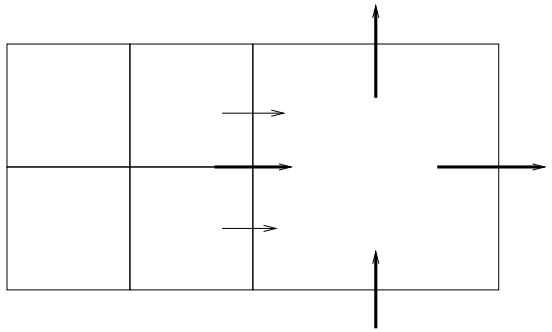


On a two-level AMR grid, we have U^c, U^f, and the update is performed in the following steps.

- Update solution on entire coarse grid: $U^c := U^c - \Delta t^c D^c \vec{F}^c$.
- Update solution on entire fine grid: $U^f := U^f - \Delta t^f D^f \vec{F}^f$ (n_{refine} times).
- Synchronize coarse and fine grid solutions.

Synchronization of Multilevel Solution

- Average coarse-grid solution onto fine grid.
- Correct coarse cells adjacent to fine grid to maintain conservation.



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

Typically, need a generalization of GKS theory for free boundary problem to guarantee stability (Berger, 1985). Stability not a problem for upwind methods.

Discretizing Elliptic PDE's

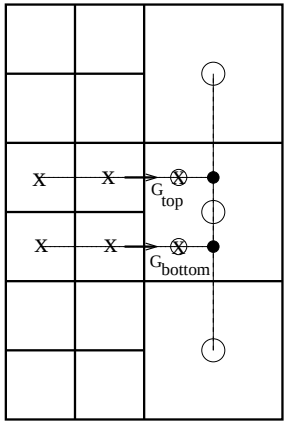
We compute $\psi^{comp} = \{\psi^c, \psi^f\}$, the solution of the properly-posed problem on the composite grid.

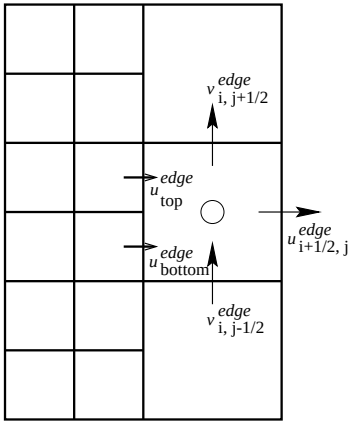
$$\Delta^c \psi^c = g^c \text{ on } \Omega^c - \mathcal{C}(\Omega^f)$$

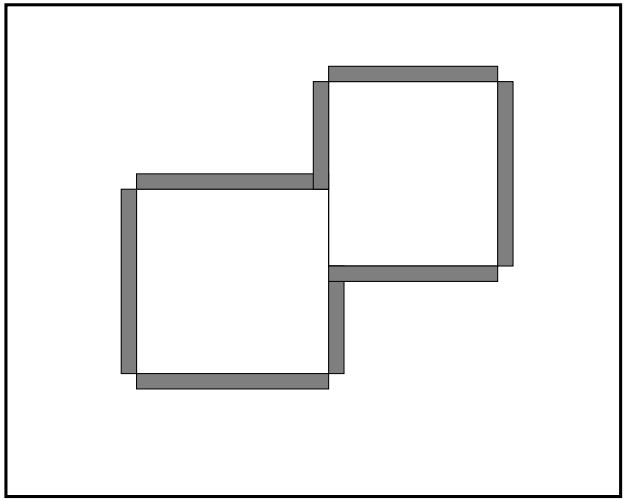
$$\Delta^f \psi^f = g^f \text{ on } \Omega^f$$

$$\psi^c = \psi^f, \quad \frac{\partial \psi^c}{\partial n} = \frac{\partial \psi^f}{\partial n} \text{ on } \partial\Omega^{c/f}$$

The Neumann matching conditions are flux matching conditions, and are discretized by computing a single-valued flux at the boundary. On the fine grid, one is still solving the same BVP as in the naive case, but with coarse grid values that have been modified to account for the Neumann matching conditions.







Chombo: a Software Framework for Block-Structured AMR

Requirement: to support a wide variety of applications that use block-structured AMR using a common software framework.

• 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, HDF5, VTK.

• Interoperability with other tools.

Previous work: BoxLib (LBNL/CCSE), KeLP (Baden, et. al., UCSD), FIDIL (Hilfinger and Colella).

Layered Design

• **Layer 1.** Data and operations on unions of boxes – set calculus, rectangular array library (with interface to Fortran), data on unions of rectangles, with SPMD parallelism implemented by distributing boxes over processors.

• **Layer 2.** Tools for managing interactions between different levels of refinement in an AMR calculation – interpolation, averaging operators, coarse-fine boundary conditions.

• **Layer 3.** Solver libraries – AMR-multigrid solvers, Berger-Oliger time-stepping.

• **Layer 4.** Complete parallel applications.

• **Utility layer.** Support, interoperability libraries – API for HDF5 I/O, visualization package implemented on top of VTK, C API's.

Examples of Layer 1 Classes (`BoxTools`)

• `IntVect` $\boldsymbol{i} \in \mathbb{Z}^d$. Can translate $\boldsymbol{i}_1 \pm \boldsymbol{i}_2$, coarsen $\frac{\boldsymbol{i}}{s}$, refine $\boldsymbol{i} * s$.

• `Box` $B \subset \mathbb{Z}^d$ is a rectangle: $B = [\boldsymbol{i}_{low}, \boldsymbol{i}_{high}]$. B can be translated, coarsened, refined. Supports different centerings (node-centered vs. cell-centered) in each coordinate direction.

• `IntVectSet` $\mathcal{I} \subset \mathbb{Z}^d$ is an arbitrary subset of $\mathbb{Z}^d$. $\mathcal{I}$ can be shifted, coarsened, refined. One can take unions and intersections, with other `IntVectSet`s and with `Box`es, and iterate over an `IntVectSet`.

• `FArrayBox A(Box B, int nComps)`: multidimensional arrays of Reals constructed with `B` specifying the range of indices in space, `nComp` the number of components. `Real* FArrayBox::dataPointer` returns pointer to the contiguous block of data that can be passed to Fortran.

Example: explicit heat equation solver on a single grid

```
// C++ code:

  Box domain(-IntVect:Unit,nx*IntVect:Unit);
  FArrayBox soln(grow(domain,1), 1);
  soln.setVal(1.0);

  for (int nstep = 0;nstep < 100; nstep++)
 {
  heatsub2d_(soln.dataPtr(0),
           &(soln.loVect()[0]), &(soln.hiVect()[0]),
           &(soln.loVect()[1]), &(soln.hiVect()[1]),
             region.loVect(), region.hiVect(),
             &dt, &dx, &nu);
 }
```

```
c Fortran code:
      subroutine heatsub2d(phi,nlphi0, nhphi0,nlphi1, nhphi1,
     &     nlreg, nhreg, dt, dx, nu)

      real*8 lphi(nlphi0:nhphi0,nlphi1:nhphi1)
      real*8  phi(nlphi0:nhphi0,nlphi1:nhphi1)
      real*8 dt,dx,nu
      integer nlreg(2),nhreg(2)

c Remaining declarations, setting of boundary conditions goes here.
...
       do j = nlreg(2), nhreg(2)
         do i = nlreg(1), nhreg(1)
            lapphi =
     &           (phi(i+1,j)+phi(i,j+1)
     &           +phi(i-1,j)+phi(i,j-1)
     &           -4.0d0*phi(i,j))/(dx*dx)

            lphi(i,j) = lapphi
         enddo
       enddo
```

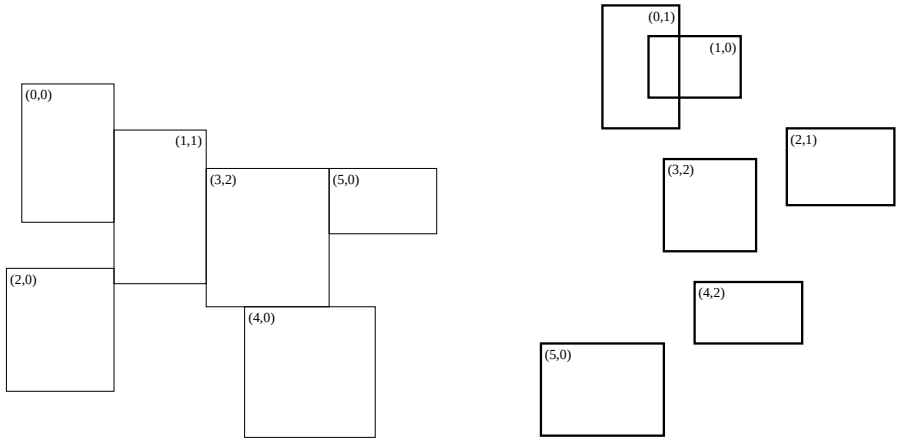
```
c Increment solution with rhs.

      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

      return
      end
```

Distributed Data on Unions of Rectangles

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



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

• `template <class T> LevelData<T>` and other container classes hold data distributed over multiple processors. For each `k = 1 ... nGrids`, an "array" of type `T` corresponding to the box B_k is allocated on processor p_k. Straightforward API's for copying, exchanging ghost cell data, iterating over the arrays on your processor in a SPMD manner.

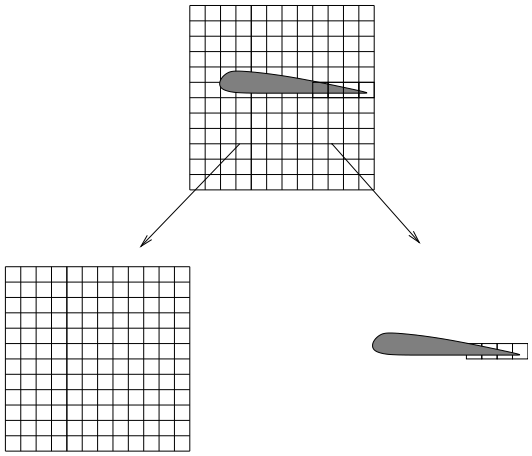
Software Reuse by Templating Dataholders

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

`BaseFAB<T>` is a multidimensional array whihc can be defined for for any type `T`. `FArrayBox: public BaseFAB<Real>`

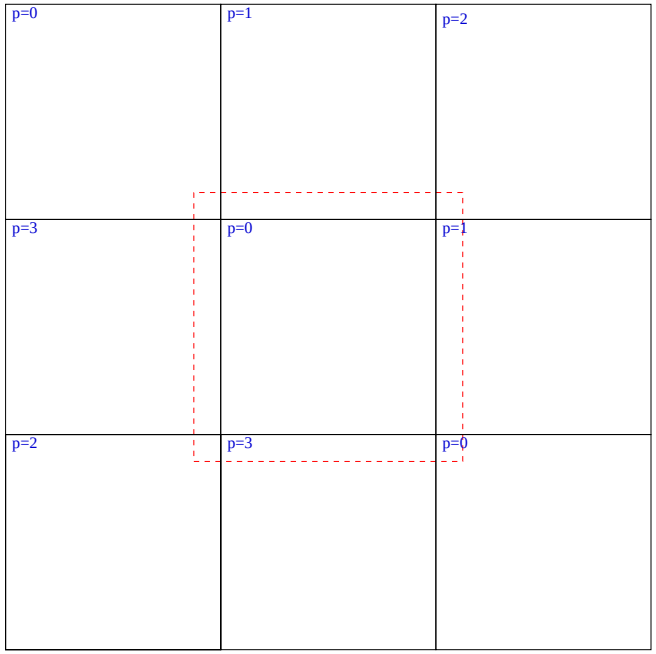
In `LevelData<T>`, `T` can be any type that "looks like" a multidimensional array. Examples include:

- Ordinary multidimensional arrays, e.g. `LevelData<FArrayBox>`.
- A composite array type for supporting embedded boundary computations:



- Binsorted lists of particles, e.g. `BaseFab<List<ParticleType>>`

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.

• `LevelData<T>::exchange()`: obtains ghost cell data from valid regions on other patches.

• `DataIterator`: iterates over only the patches that are owned on the current processor.

```
// C++ code:
  Box domain;
  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 value
// and apply the operator to data on each of the Boxes owned by this
// processor.

  phi.exchange();
  DataIterator dit = dbl.dataIterator();

// Iterator iterates only over those boxes that are on this processor
```

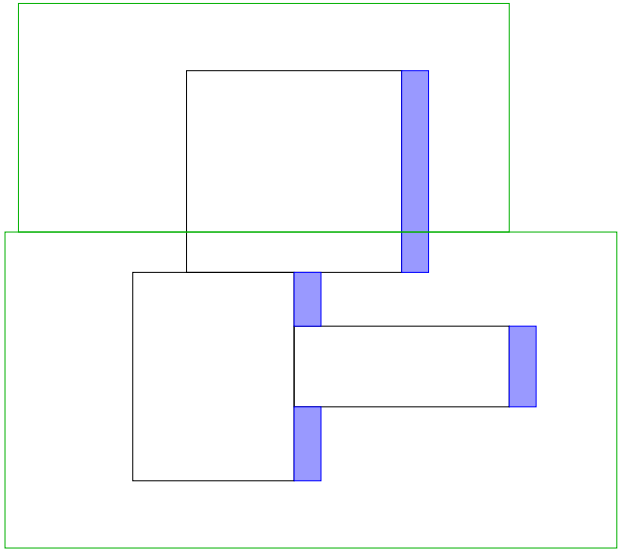
```
for (dit.reset();dit.ok();++dit)
{
FArrayBox& soln = phi[dit()];
Box& region = dbl[dit()];
heatsub2d_(soln.dataPtr(0),
         &(soln.loVect()[0]), &(soln.hiVect()[0]),
         &(soln.loVect()[1]), &(soln.hiVect()[1]),
           region.loVect(), region.hiVect(),
           domain.loVect(), domain.hiVect(),
           &dt, &dx, &nu);
}
}
```

Layer 2: Coarse-Fine Interactions (`AMRTools`).
The operations that couple different levels of refinement are among the most difficult to implement AMR.

- Interpolating between levels (`FineInterp`).
- Interpolation of boundary conditions (`PWLFillpatch, QuadCFInterp`).
- Averaging down onto coarser grids (`CoarseAverage`).
- Managing conservation at coarse-fine boundaries (`LevelFluxRegister`).

These operations typically involve interprocessor communication and irregular computation.

Example: `class LevelFluxRegister`



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

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

$$\delta F = 0$$

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

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

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

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: Reusing Control Structures Via Inheritance (`AMRTimeDependent`, `AMRElliptic`).
AMR has multilevel control structures are largely independent of the details of the operators and the data.

- Berger-Oliger refinement in time.
- Multigrid iteration on a union of rectangles.
- Multigrid iteration on an AMR hierarchy.
- Various Godunov-type methods for hyperbolic conservation laws.

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

Example: The `PatchGodunov / GodunovPhysics` Interface.
Unsplit Godunov methods (Colella, 1990, LeVeque 1992, Saltzman, 1994, Miller and Colella, 2002) are elaborate predictor-corrector methods for computing the fluxes for a system of hyperbolic conservation laws that include the effect of corner coupling. Algorithmically, there is a clean separation between the physics-dependent parts and an elaborate predictor-corrector calculation to compute the effect of various upwind and limited central differences on the flux at a face. We implement this control structure using a pair of classes.

`class PatchGodunov`: manages the predictor-corrector calculation.

`class GodunovPhysics`: collection of virtual functions called by a `PatchGodunov` object that perform the problem-dependent calculations:

- `virtual void GodunovPhysics::charAnalysis(int d,...)`: computes the expansion of an increment of the solution in terms of right eigenvectors at each point on the grid.
- `virtual void GodunovPhysics::charSynthesis(int d,...)`: computes an increment in the solution given the expansion coefficients at each point on the grid.
- `virtual void GodunovPhysics:riemann(int d, ...)`: given left and right states, compute the solution to the Riemann problem at each point on the grid.

For each method, the input parameter d is the coordinate direction.

PatchGodunov has as member data a pointer to an object of type GodunovPhysics, one for each level of refinement:

```
GodunovPhysics* m_physics;
```

PatchGodunov calls the various member functions of GodunovPhysics as it advances the solution in time:

```
m_physics->riemann(d,...);
```

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

```
class PolytropicPhysics : public GodunovPhysics
// Defines functions in the interface, as well as data.
virtual void riemann(int d, ...)
{
// computes solution to the Riemann problem.
...
}
```

To use the PatchGodunov class for this particular application, m_physics will point to objects in the derived class, e.g.,

```
PolytropicPhysics* polytropicPtr = new Polytropic(...);
GodunovPhysics* physicsPtr = static_cast <GodunovPhysics*> (polytropicPtr);
```

AMR Utility Layer

• API for HDF5 I/O.

• Interoperability tools. We are developing a framework-neutral representation for pointers to AMR data, using opaque handles. This will allow us to wrap Chombo classes with a C interface and call them from other AMR applications.

• Chombo Fortran - a macro package for writing dimension-independent Fortran and managing the Fortran / C interface.

• `Parmparse` class from BoxLib for handling input files.

• Visualization and analysis tools (ChomboVis).

I/O Using HDF5

NSCA's HDF5 mimics the Unix file system.

• Disk file $\leftrightarrow$ "/".

• Group $\leftrightarrow$ subdirectory .

• Attribute, dataset $\leftrightarrow$ files. Attribute: small metadata that multiple processes in a SPMD program may write out redundantly. Dataset: large data, each processor writes only the data it owns.

Chombo API for HDF5

• Parallel neutral: can change processor layout when re-inputting output data.

• Dataset creation is expensive - one does not want to create one per rectangular grid. Instead, create one dataset for each `BoxLayoutData` or `LevelData`. Each grid's data is written into offsets from the origin of that dataset.

Load Balancing

For parallel performance, need to obtain approximately the same work load on each processor.

• Unequal-sized grids: knapsack algorithm provides good efficiencies provided the number of grids / processor ≥ 3 (Crutchfield, 1993). Has been modified to better preserve locality.

• Equal-sized grids can provide perfect load balancing if algorithm is reasonably homogeneous. Disadvantage: many small patches can lead to large amounts of redundant work.

Both methods obtain good scaling into 100's of nodes for hyperbolic problems. Alternative approach: space-filling curves using equal-sized grids, followed by agglomeration.