SAMRAI Concepts and Software Design

Version 0.1

R.W. Anderson, W.J. Arrighi, N.S. Elliott, B.T. Gunney, R.D. Hornung

LLNL-SM-617092-DRAFT

February 14, 2013

Table of Contents

1 Introduction

1.1 Purpose of this document

SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) is an object-oriented C++ class library developed in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). It provides extensive support for development of parallel structured adaptive mesh refinement (SAMR) applications. This document describes the organization and design of the SAMRAI library. The primary intent of this material is to help SAMRAI users and developers understand how SAMRAI works and how it may be used in (SAMR) application development.

1.2 Disclaimer and auspices statements

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

2 SAMRAI and your application

Before getting into a detailed discussion of SAMRAI it is worth discussing some misperceptions about the library that the SAMRAI development team has encountered in discussions with potential users. One frequently held notion is that SAMRAI "owns" the application data. This most likely has arisen because SAMRAI does provide a set of PatchData types and our examples make use of them. However, there is no requirement that an application must use these classes. The class PatchData is an abstract base class that exists to provide an abstract interface. Application developers are free to implement their own data classes that derive from PatchData and implement that class' abstract interface. As an application developer you are not in any way bound to use SAMRAI's data classes unless you wish to.

Another misconception that we have heard is that SAMRAI "controls" the application. This notion may have come about due to the existence of the "algs" and "appu" components in the library. These components add things like time integrators, level integrators, specific communication transactions, boundary data treatments, etc. Theses components exist for 2 reasons:

1. To complete implementations of SAMRAI abstract base classes so that the SAMRAI library can create complete tests
2. So that users have examples from which they may pattern their own implementations
3. They work and may be used directly if they suit your application

Again, there is no requirement that an application developer must use one of these time integrators or something else from the algs and appu components unless he wants to.

3 SAMR and SAMRAI

3.1 Structured Adaptive Mesh Refinement (SAMR)

Many science and engineering simulation problems exhibit solutions with localized features, such as large gradients, separated by relatively large regions in which the solution is smooth or varies little. A fine computational mesh (i.e., the discrete time-space domain on which the equations are approximated) is often required to resolve certain local features while a coarser mesh suffices elsewhere. Since mesh resolution determines the accuracy and cost of a computation, using fine mesh everywhere may be inefficient or worse, unacceptably expensive. In many problems, the location and resolution of fine mesh required for a desired level of accuracy may not be known *a priori*. Adaptive mesh refinement (AMR) is a computational technique used to focus mesh resolution where it is needed dynamically. When applied properly, AMR is a powerful tool that can adjust mesh resolution to resolve local features with sufficient accuracy without incurring the cost of a globally fine mesh.

Structured AMR (SAMR) refers to the use of structured mesh components (i.e., logically-rectangular mesh blocks) in the implementation of an adaptive mesh. SAMR codes typically adopt one of two implementation strategies: *patch-based* and *tree-based*. The approaches differ in data structures and algorithms used to manage and adapt the mesh. SAMRAI utilizes the patch-based approach.

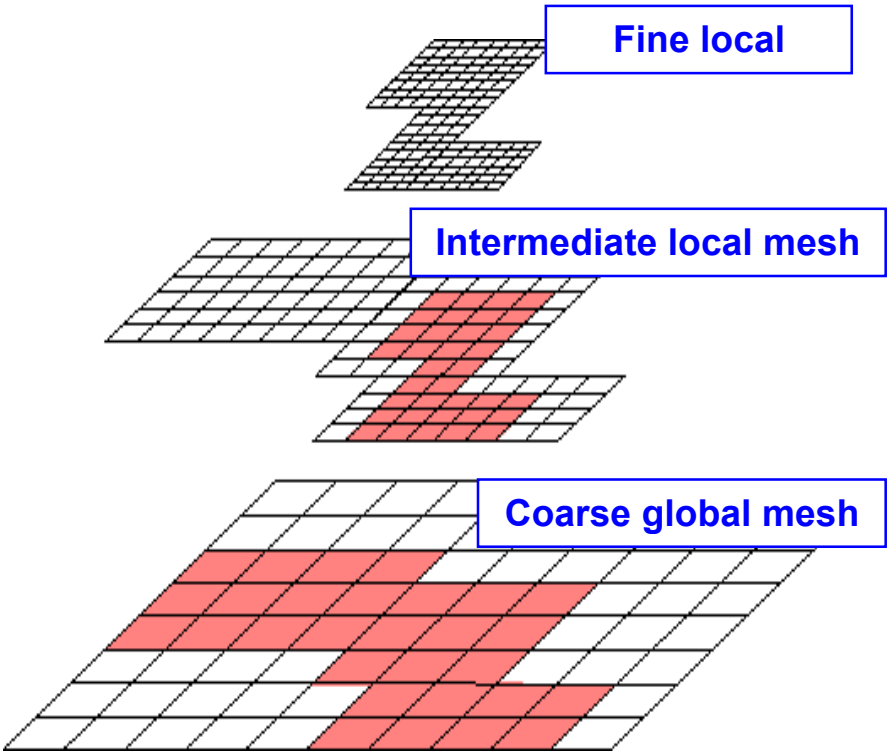


Figure 1. A simple two-dimensional, three-level structured mesh hierarchy. The coarsest level covers the entire computational domain. Each finer level is nested within the next coarser level. Refined cells on each level are shown in red.

In an SAMR application, the computational mesh consists of a *hierarchy of nested levels* of mesh refinement; see Figure 1. The *coarsest* level covers the entire computational problem domain. Each successively finer level in the hierarchy is contained within the interior of the next coarser level. A level represents a "uniform" degree of mesh resolution meaning that a finer level is related to its next coarsest level by a *refinement ratio* (typically, an integer vector). This ratio defines the number of finer level mesh zones contained within a zone on the coarser level. A level is partitioned into a disjoint union *box* regions, each representing a logically rectangular extent (i.e., multidimensional interval) in mesh index space. A *patch* is a container that holds the simulation data on a portion of mesh defined by a box. During mesh adaptation, cells on a level are selected (or "tagged") for refinement using some error estimation or feature detection criteria that is usually specific to an application. Tagged cells are covered by a disjoint union of boxes which form a new finer mesh level in the hierarchy.

Similar to non-adaptive, parallel block-structured mesh computations, SAMR algorithms are organized into numerical routines that operate on data on spatially-distributed box regions interleaved with communication operations that pass data between these regions; for example, to fill "ghost'' data at box boundaries. In addition to communication between boxes on a single level, SAMR requires *inter-level* data communication to refine and coarsen simulation data between levels in the mesh hierarchy. Such inter-level data communication depends on the numerical approximations employed in an application.

3.2 What is SAMRAI?

SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) is an object-oriented C++ class library developed in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). It provides extensive support for parallel SAMR application development and it is designed to serve as a framework of software components that can be shared across a diverse range of SAMR applications. For example, SAMRAI frees application developers from most low-level programming details related to SAMR mesh and data management and communication. In addition, it is fairly straightforward to use SAMRAI to evolve an existing serial code that solves problems on a block-structured mesh to work in parallel with or without adaptive meshing.

Beyond core SAMR mesh and data management, the library provides some computational algorithms that are common in SAMR applications. Classes that perform time integration and solve Poisson problems, for example, which have been developed for specific applications built on SAMRAI, are included in the library. Often, such algorithms are designed for specialization, using techniques such as Strategy and Template Method design patterns, for new applications.

It is important to emphasize that SAMRAI does not take away user control over numerical methods employed in applications. Developers are entirely responsible for the implementation of numerical routines they use. Generally, the pieces of SAMRAI are decoupled from each other and from application-specific operations to provide flexibility and promote software reuse. For example, SAMRAI's parallel data communication infrastructure is independent of the actual data being communicated so that users may integrate their own data structures into a parallel SAMR environment by providing only a handful of methods defined by the SAMRAI mesh data API.

3.3 SAMRAI library organization

The SAMRAI library is comprised of a collection of software "packages". Strict dependency relationships are maintained among these packages making it easier for users to focus only on the portions of the library that they require for development. The following list provides a brief description of each package with the four-letter SAMRAI namespace identifier in parentheses. The ordering in this list indicates inter-package dependencies; classes in package 1 depend only on classes in package 1, classes in package 2 depend only on classes in packages 1 and 2, etc.

1. **Toolbox (tbox).** Basic utility classes used in the library and in application development including: MPI wrappers and utilities; stream classes for interprocess communication; tools for managing input and restart files; and event logging, tracing, and timing.
2. **Hierarchy (hier).** Classes used to define and operate on mesh index spaces, to describe SAMR patch hierarchy structure, and base class interfaces for variables and simulation data.
3. **Transfer (xfer).** General support for interpatch data communication on a level and between levels in an SAMR patch hierarchy.
4. **Patch Data (pdat).** Various array-based patch data types for different mesh centerings such as cell, node, face, edge, etc. as well as support for sparse data representations.
5. **Math Operations (math).** Basic arithmetic and other operations, such as dot products and norms, that are needed for vector kernels for all array-based patch data types in SAMRAI.
6. **Meshing (mesh).** Algorithms and interfaces for creating and adaptive regridding of levels in an SAMR patch hierarchy.
7. **Algorithms (algs).** Classes used to construct solution algorithms for certain types of PDE problems such as explicit time integration with time subcycling on finer levels.
8. **Solvers (solv).** Classes useful for applying linear and nonlinear solver methods in SAMR including: assembling arbitrary collections of patch data into solution vectors, interfaces and wrapper classes for PETSc, Sundials and hypre, as well as an FAC Poisson solver.
9. **Mesh geometry (geom.).** Support for mesh coordinate systems and associated interlevel interpolation and coarsening operators.
10. **Application Utilities (appu).** Simple utilities for setting physical boundary conditions and generating Visit data files that are useful for application development.

4 SAMRAI application development

4.1 Basic structure of a SAMRAI application

Since SAMRAI is a class library, it is the responsibility of an application developer to implement the main program that executes a simulation. A typical SAMRAI application employs the same basic concepts found in most scientific computing applications. A common main program structure contains the following operations: initialize the simulation environment (MPI and SAMRAI), read input file data, compose objects from the library with application-specific entities to form a simulation algorithm, execute simulation control logic including writing/reading restart files and writing visualization data, clean up memory and exit.

4.2 An example main program for a SAMRAI application

In this section, we show an abridged listing of the main program that appears in the SAMRAI linear advection example code that is part of the library. This example illustrates in some detail the discussion in the preceding section. Details describing parts of the code enumerated by the red comments are discussed below. The associations among the objects composed by this construction pattern are shown in Figure (add figure? Would this help?).

```
//
// 0:  Header file inclusions
//

int main( int argc, char* argv[])
{
   // 1:  Initialize MPI and SAMRAI
   tbox::SAMRAI_MPI::init(&argc, &argv);
   tbox::SAMRAIManager::initialize();

   // 2:  Startup SAMRAI
   tbox::SAMRAIManager::startup();

   // 3:  Create input database and parse input file
   boost::shared_ptr< tbox::InputDatabase > input_db(
          new tbox::InputDatabase("input_db") );
   tbox::InputManager::getManager()->parseInputFile(filename, input_db);

   // 4:  Access "Main" input database and create problem dimension
   boost::shared_ptr<tbox::Database> main_db(input_db->getDatabase("Main"));
   const tbox::Dimension dim(
         static_cast<unsigned short>( main_db->getInteger("dim")) );
```

```
// 5:  Create and compose main algorithmic objects for the simulation

// 5a:  Grid geometry and patch hierarchy
boost::shared_ptr<geom::CartesianGridGeometry> grid_geometry(
   new geom::CartesianGridGeometry(
      dim,
      "CartesianGeometry",
      input_db->getDatabase("CartesianGeometry")) );

boost::shared_ptr<hier::PatchHierarchy> patch_hierarchy(
   new hier::PatchHierarchy(
      "PatchHierarchy",
      grid_geometry,
      input_db->getDatabase("PatchHierarchy")) );

// 5b:  HyperbolicPatchStrategy object with Linear advection routines
LinAdv* linear_advection_model = new LinAdv(
      "LinAdv",
      dim,
      input_db->getDatabase("LinAdv"),
      grid_geometry );

// 5c:  Level integrator for hyperbolic equations
boost::shared_ptr<algs::HyperbolicLevelIntegrator> hyp_level_integrator(
   new algs::HyperbolicLevelIntegrator(
      "HyperbolicLevelIntegrator",
      input_db->getDatabase("HyperbolicLevelIntegrator"),
      linear_advection_model,
      ...) );

// 5d:  Cell tagging algorithm
boost::shared_ptr<mesh::StandardTagAndInitialize> error_detector(
   new mesh::StandardTagAndInitialize(
      dim,
      "StandardTagAndInitialize",
      hyp_level_integrator.get(),
      input_db->getDatabase("StandardTagAndInitialize")) );

// 5e:  Cell clustering, load balancing, and gridding algorithm
boost::shared_ptr<mesh::BergerRigoutsos> box_generator(
   new mesh::BergerRigoutsos(
      dim,
      input_db->getDatabaseWithDefault(
      "BergerRigoutsos",
      boost::shared_ptr<tbox::Database>())) );

boost::shared_ptr<mesh::TreeLoadBalancer> load_balancer(
   new mesh::TreeLoadBalancer(
      dim,
      "LoadBalancer",
      input_db->getDatabase("LoadBalancer")) );

boost::shared_ptr<mesh::GriddingAlgorithm> gridding_algorithm(
   new mesh::GriddingAlgorithm(
      patch_hierarchy,
      "GriddingAlgorithm",
```

```
      input_db->getDatabase("GriddingAlgorithm"),
      error_detector,
      box_generator,
      load_balancer) );

// 5f:  Hierarchy time integration algorithm
boost::shared_ptr<algs::TimeRefinementIntegrator> time_integrator(
   new algs::TimeRefinementIntegrator(
      "TimeRefinementIntegrator",
      input_db->getDatabase("TimeRefinementIntegrator"),
      patch_hierarchy,
      hyp_level_integrator,
      gridding_algorithm));


// 6:  Initialize hierarchy and get initial time step and
//     simulation time info from integrator
double dt_now = time_integrator->initializeHierarchy();

double loop_time = time_integrator->getIntegratorTime();
double loop_time_end = time_integrator->getEndTime();

// 7:  Iterate over time steps until done
while ( (loop_time < loop_time_end) &&
        time_integrator->stepsRemaining() ) {

   // 7a:  Advance solution to the new time
   double dt_new = time_integrator->advanceHierarchy(dt_now);

   loop_time += dt_now;
   dt_now = dt_new;

   // 7b:  Write restart and viz files if desired

}

// 8:  Destroy objects used in simulation
time_integrator.reset();
gridding_algorithm.reset();
load_balancer.reset();
box_generator.reset();
error_detector.reset();
hyp_level_integrator.reset();
if (linear_advection_model) delete linear_advection_model;
patch_hierarchy.reset();
grid_geometry.reset();
input_db.reset();
main_db.reset();
```

```
   // 9:  Shutdown SAMRAI
   tbox::SAMRAIManager::shutdown();

   // 10:  Finalize SAMRAI and MPI environments
   tbox::SAMRAIManager::finalize();
   tbox::SAMRAI_MPI::finalize();

   return 0;
}
```

A summary of the major object associations created by the calls to the object constructors is shown in Figure 2

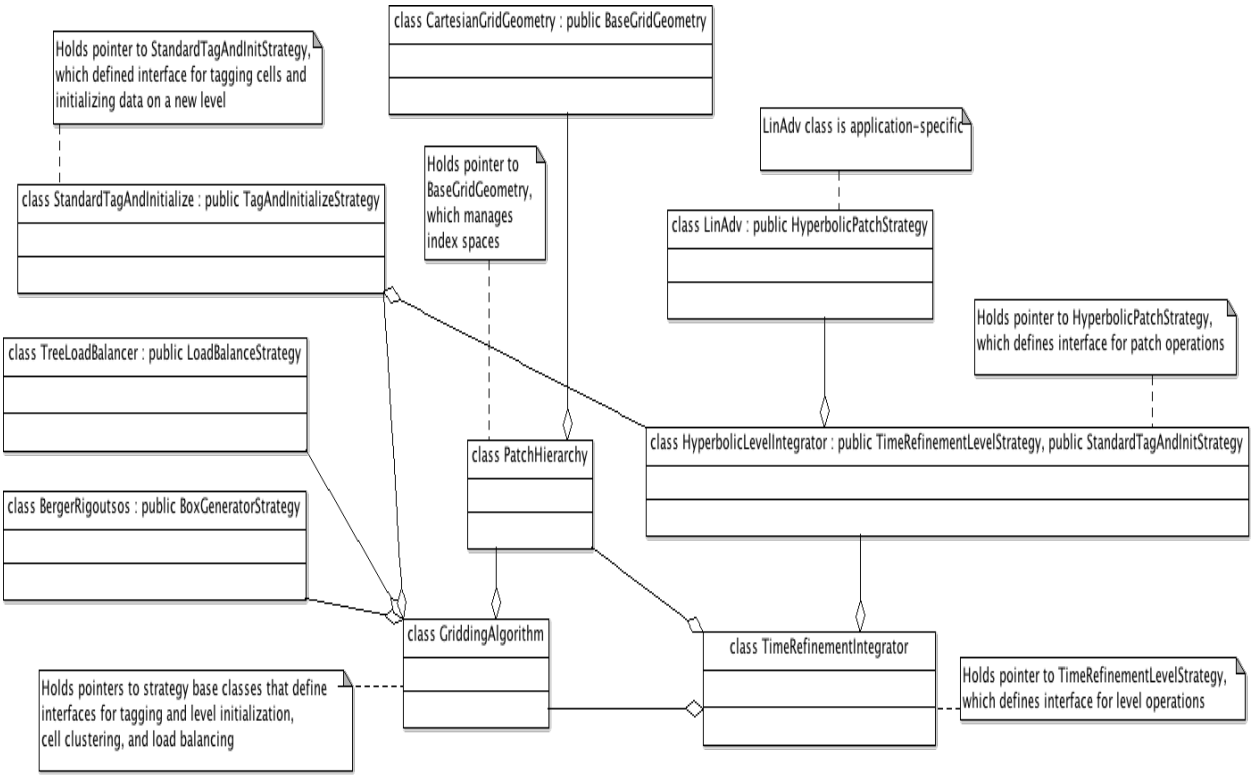


Figure 2: Major object associations created by constructor calls in linear advection example.

Next, we elaborate on some details of this code. First, note that each SAMRAI class name is qualified with the name of the SAMRAI package in which it resides. The numbers in the following list correspond to the numbered comments in the code above.

0. Before the main routine, we include header files containing definitions of the SAMRAI classes and other entities used in the routine
1. Initialize the MPI and SAMRAI environments. Note that MPI must be initialized first. The SAMRAIManager::initialize( ) method initializes SAMRAI assertion handlers, I/O, and other

SAMRAI internals. It must be called before any SAMRAI objects are used and must be called exactly once.

2. Startup certain SAMRAI classes by calling SAMRAIManager::startup( ). This method, and the corresponding SAMRAIManager::shutdown( ) method, may be called multiple times to execute multiple SAMRAI problems successively within a single program execution.
3. Create an input database and parse the contents of an input file into that database. Typically, the name of the input file is given as a command line argument. Such details are omitted here.
4. Many SAMRAI objects require a Dimension object when constructed. Usually, the spatial problem dimension is specified as an integral value in the input file and this is used to create the Dimension object. This is illustrated here.
5. Build a complete SAMRAI simulation algorithm by creating and composing its constituent parts. A common SAMRAI pattern for this is to pass objects representing parts of an algorithm into other objects that represent a larger algorithm. Note that each of these classes also takes the following two arguments: a name string identifier (used to distinguish multiple objects of the same type in error/warning messages, for restart, etc.), an input database (representing the portion of the input file holding data used to initialize the object). This constructor argument pattern is common in SAMRAI.
    a. Create grid geometry and patch hierarchy objects. The patch hierarchy object takes the grid geometry object, which defines the mesh index space for the problem and coordinate system; here we employ a Cartesian mesh. Note that the grid geometry also takes the Dimension object that defines the spatial dimension of the mesh.
    b. Create the LinAdv object, which defines the variables used in the problem and provides the numerical routines that operate on those variables on each patch. In this example, this is the only object that is specific to the simulation problem we will run.
    c. Create a HyperbolicLevelIntegrator object that drives the patch integration routines defined by the LinAdv object.
    d. Create a StandardTagAndInitialize object that coordinates the routines that tag cells for refinement and initialize data on new patches with the adaptive meshing operations. It takes the level integrator object.
    e. Build the adaptive gridding algorithm by composing the StandardTagAndInitialize, BergerRigoutsos (cell clustering), and TreeLoadBalancer objects. These objects each define a piece of the adaptive meshing process and are passed to the GriddingAlgorithm class constructor, which coordinates the operations provided by each part.
    f. Lastly, construct the TimeRefinementIntegrator object that orchestrates integration of levels in the hierarchy with adaptive meshing operations. Its constructor takes the patch hierarchy, level integrator, and gridding algorithm objects.
6. At this point, we have composed the entire structure of the simulation components we need to run our problem. We start by telling the integrator to build the initial patch hierarchy configuration by calling its initializeHierarchy( ) method. Using all the algorithmic pieces we have provided, the integrator builds the coarsest hierarchy level and initializes data on it. Then, it tags cells for refinement (using routines provided by the LinAdv object) and builds the next finer level

based on those tags. This process repeats until some maximum number of levels is created or no more cells are tagged for refinement. Finally, we are ready to begin the time integration process. Note that the time integrator owns the simulation time information.

7. The time integration process is comprised of a loop over time increments in which we:
    a. Advance the solution to the new time, and
    b. Write out visualization and restart files, as needed.
8. When we have reached the final simulation time and have exited the time step loop, we begin the memory cleanup process. The first part of this is to destroy the objects we created earlier.
9. Shutdown SAMRAI classes by calling SAMRAIManager::shutdown( ). This method may be followed by a call to SAMRAIManager::startup( ) to run another problem in the same program.
10. Finalize the SAMRAI and MPI environments. Note that MPI must be finalized last.

5 Index spaces and boxes

A central concept in SAMR is the *index space*, which defines the structure of the computational mesh for a simulation problem. Nearly all operations that interact with an SAMR mesh hierarchy and simulation data on it involve index spaces. As mentioned in Section 3.1, a *box* is the central concept used to describe the extent of logical coordinates on a portion of an SAMR mesh.

In SAMRAI, the computational domain on each hierarchy level is represented as a collection of boxes, each of which defines a multidimensional interval in the mesh index space associated with that level. SAMRAI employs the convention that all boxes describing an SAMR mesh hierarchy are *cell-centered*. That is, a box is defined by two cell indices, lower and upper, which define the bounds of the multidimensional interval in index space covered by the box. Boxes on different levels are related by a *refinement ratio* that describes how the index space on one level is a coarsening or refinement of the mesh index space on another level.

5.1 Illustration of key index space concepts

Figure 3 illustrates essential index space concepts needed to describe an SAMR mesh using a simple two-dimensional, three level example. The coarsest level is represented by a 5-by-4 mesh, which is defined by the problem *index space* for that level. A box with lower and upper cell indices (0, 0) and (4, 3), respectively, defines this mesh (cell coordinate numbers 0 – 4 and 0 – 3 are shown in black along the bottom and left). SAMRAI uses the notation [(0,0),(4,3)] to describe such a box. Patches on the coarsest level always cover the entire problem domain, and so we use the same box to define a patch covering the level in this case. The next finer (intermediate) level relates to the coarsest level by a refinement ratio of (4, 2). That is, eight intermediate level cells (4-by-2) cover each refined coarse level cell. The extent of the problem index space for the intermediate level is described by a box with lower and upper cell indices (0, 0) and (19, 7), respectively (cell coordinates are shown in blue along the bottom and right). A single patch, defined by the box [(8, 0),(19, 5)] is shown. Note that this patch represents a refinement of the box [(2, 0),(4, 2)] on the coarsest level. The finest level relates to the intermediate level by a refinement ratio of (1, 2); i.e., two finest level cells (1-by-2) cover each refined intermediate level cell. Thus, the extent of the problem index space for the finest level is [(0, 0),(19, 15)] (cell coordinates are shown in red along the bottom and right). Lastly, one patch, defined by the box [(12, 4),(15, 7)], is shown on the finest level. This patch represents a refinement of the box [(12, 2),(15, 3)] on the intermediate level.

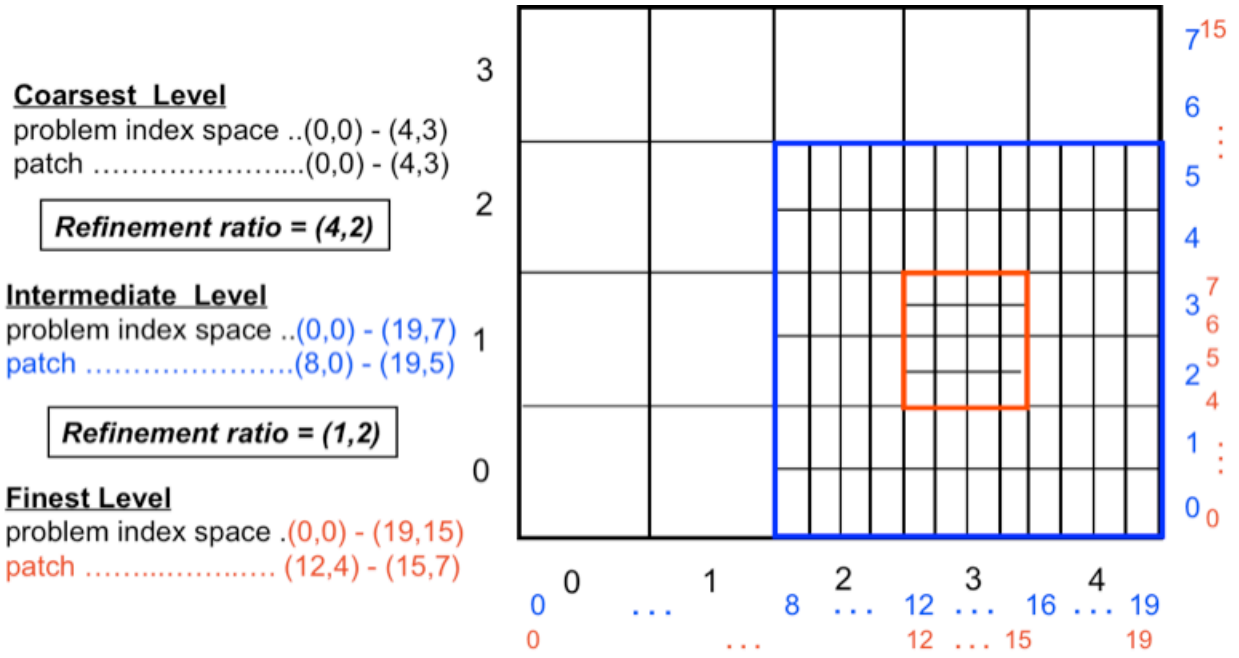


Figure 3. This simple example shows a two-dimensional, three level mesh to illustrate the main concepts used to describe an SAMR patch hierarchy. The (cell-centered) problem index space for each level defines the maximum logical extent of the computational mesh on the level. Index spaces for two different levels are related by a refinement ratio. A patch is defined by a box that covers some multidimensional interval of the index space on the level on which the patch resides.

5.2 Key classes and their associations

The notion of a *box* is a central concept used to define mesh index spaces associated with SAMR patch hierarchies and operations on them. Figure 4 show basic associations among key SAMRAI classes: Box, Index, IntVector, and Dimension. A Box object represents a multi-dimensional interval in index space and is defined by its lower and upper Index members. It provides access to these indices as well as many simple box calculus operations, such as growing/shrinking, shifting, coarsening/refining, intersection with another box, etc. An Index object represents a single point in some index space. The Index class is a subclass of IntVector, which implements an integer vector and provides a variety of arithmetic and other operations on such vectors. The dimensionality of Index and IntVector objects is set when they are constructed by providing a Dimension object. A Dimension object holds a scalar integral type that defines the dimensionality.

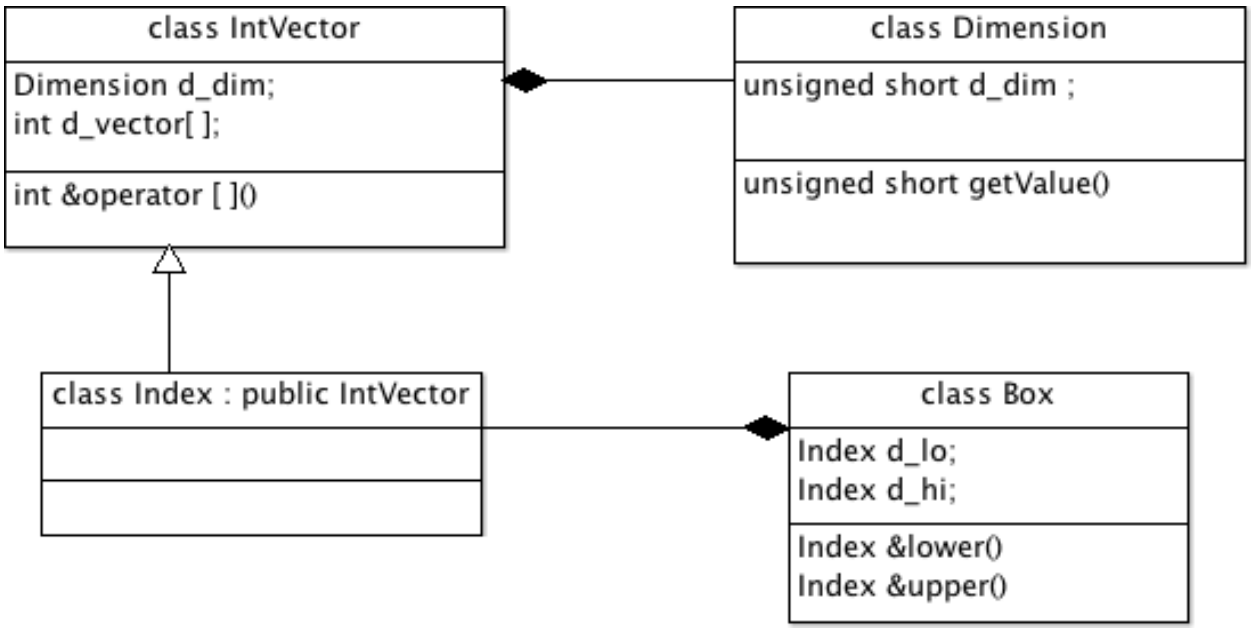


Figure 4. The key classes used to represent index space concepts are: Box, Index, IntVector, and Dimension.

5.3 Simple Index and Box usage example

This section illustrates in code some basic usage of Index and Box types by showing various operations to create the Boxes describing the patches discussed in Section 5.1. It is important to note that there are many ways to generate the boxes "box_0", "box_1", and "box_2" in this example. The intent here is to illustrate some basic Index, IntVector, and Box manipulations.

```
#include "SAMRAI/tbox/Dimension.h"
#include "SAMRAI/hier/BlockId.h"
#include "SAMRAI/hier/Index.h"
#include "SAMRAI/hier/Box.h"

const tbox::Dimension dim( 2 );  // Define index space dimension

hier::BloxkId blockid( 0 );      // Define block id

// Create level zero box by specifying its lower, upper indices.
Index lo( 0, 0 );
Index hi( 4, 3 );
Box box_0( lo, hi, blockid );

// Create level one box as a refinement of a level zero box.
lo( 0 ) = 2;
hi( 1 ) = 2;
Box box_1( lo, hi, blockid );
IntVector ratio( dim ); ratio( 0 ) = 4; ratio( 1 ) = 2;
box_1.refine( ratio );
```

```
// Create level two box by making a copy of level one box, shrinking it, and
// refining the result.
Box box_2( box_1 );
IntVector shrink( dim, ); shrink( 0 ) = -4; shrink( 1 ) = -2;
box_2.grow( shrink );
ratio( 0 ) = 1;
box_2.refine( ratio );
```

In this example code, we first include the SAMRAI header files needed to make the code work. Then, we define an object representing our two-dimensional index space and a BlockId. The Dimension object is required to create IntVector objects. The BlockId is not germane to our discussion here and is only shown for completeness (each Box must be constructed with a block id); the concept of mesh blocks will be introduced in Section 8.

To create "box_0", the box defining the patch on the coarsest level, we construct its lower and upper indices and pass these to the Box constructor. We generate the intermediate level patch box, "box_1", by refining a box in the index space of the coarsest level. First, we modify the existing lower and upper Index objects and use these to create the box [(2, 0),(4, 2)]. Second, we create the IntVector object (4, 2), which describes the refinement ratio between the coarsest and intermediate levels. Third, we refine the box, which yields the box [(8, 0),(19, 5)] on the intermediate level. Lastly, we create the finest level box. First, we make a copy of the intermediate level box and shrink it by 4 cells in the first coordinate direction and 2 cells in the second direction by calling the box grow() method passing the IntVector (-4, -2). This yields the box [(12,2),(15,3)]. Then, we adjust the refinement ratio vector so that it represents the refinement ratio between the intermediate and fines levels (i.e., (1, 2)), and refine the box to give the box [(12, 4),(15, 7)].

5.4 BoxContainer

Nearly all SAMR computations require creation, management, and manipulation of collections of box objects. The best specific data structure for the collection, e.g., a list or a set, depends on how it will be used. For example, it may be beneficial to create an ordering for the boxes to enable more efficient searching over the collection. At other times fast insertion may be the key consideration for our collection, so a simple vector or list is a better choice. Moreover, sometimes we have a single collection that we would like to behave at times like a list and at times like a set. The BoxContainer class addresses these issues by providing a single object that can be used in more than one way, depending on the requirements at hand. It also helps eliminate redundant copies of box collections.

BoxContainer has two modes: the "unordered" and "ordered" modes. These can also be thought of as "list mode" and "set mode" respectively. Ordered here means that the order that iterators will return the box objects does not depend on how the boxes were inserted, i.e., they have an intrinsic ordering. In the unordered case, the order that the iterators will return the boxes depends on how they were added to the collection.

The BoxContainer class allows different manipulations to be performed on a single container object by managing multiple representations as necessary. Regardless of the number of internal representations a BoxContainer object maintains, only one instance of each Box in a collection exists. It is essential to know that a BoxContainer object must be in the proper state required for any manipulation that will be performed. For example, intersectBoxes() methods in the BoxContainer class require the container to be *unordered*. If one has an *ordered* container, its state must be set to *unordered* prior to calling one of these methods. For example:

```
BoxContainer bc;
Box b;
…
bc.unorder();
bc.intersectBoxes(b);
```

A developer does not need to be concerned with changes to the internal container representation that result from the call to the unorder() method. It is sufficient to know that calling the unorder() method puts the object into an unordered state, which allows the intersectBoxes() operation to be efficient; e.g., without constructing excessive copies of Boxes in the container. Furthermore, changing the container's internal representation does not cause Boxes in the container to be replicated. At any given time the container holds exactly one instance of each Box held by it.

We have intentionally chosen to require that applications explicitly call order() or unorder() rather than letting BoxContainer methods make that transformation behind the scenes. The reason is that there is a finite cost associated with changing the mode of the BoxContainer and it is therefore better to structure an algorithm so that it does not make unnecessary changes to the mode a BoxContainer's mode. Requiring applications to make this mode change serves as a reminder that some overhead cost is involved.

The documentation for the BoxContainer class describes ordering and other requirements of each of its methods. In particular, it indicates which methods work on ordered and unordered containers; methods with the same requirements are grouped together.

5.4.1 Box Tree

A BoxContainer option exists to create an internal search tree representation based on the spatial coordinates of the Boxes in the container. This option can be used to reduce the cost of searching operations in the methods removeIntersections(), intersectBoxes(), findOverlaps(), and hasOverlap(). This option is invoked by calling the BoxContainer method makeTree(). This option should only be used in cases where the listed search methods will be called multiple times on the same unchanging BoxContainer. The cost of building the tree representation is O(N(log(N))), while the tree reduces the cost of the search operations to O(log(N)) from O(N). Thus it is advisable to only use the tree representation when the reduction in search cost is expected to offset the cost of building the tree.

Constructing the tree representation via makeTree() will change nothing about the Boxes stored in the container, nor will it change the ordered/unordered state of the container.

5.5 Box Ordering

In this section, we briefly describe what ordering of Boxes in a BoxContainer means. Each box has a BoxId object, which has a GlobalId and a PeriodicId. The PeriodicId is only relevant to periodic domains and is not essential to this box ordering discussion. Therefore, it will not be discussed further here.

The GlobalId indicates the rank of an MPI process that is considered to own the Box. A GlobalId also has a LocalId that is the local identifier of the Box on the owning process. Note that the local identifier for a Box is not unique since different Boxes owned by different processes may have the same local identifier.

So, a Box has two basic pieces of identifying information for our purposes here: its owning process rank, and its local process ID,. Boxes are ordered based on the BoxId class less-than comparison operator. The operator compares owning process ranks and local IDs in this order. Thus, a Box owned by a lower rank process is "less than" a Box on a higher rank process. If two Boxes reside on the same process, the Box with the lower local ID is "less than" the one with the higher local ID. More details about these concepts and how they are used are provided in Section 6 and Section 8.

6 SAMR patch hierarchy

The class PatchHierarchy is used to represent the entire hierarchy of nested levels that make up an SAMR mesh.

6.1 Key concepts

A PatchHierarchy represents the entire SAMR hierarchy, and it consists of the PatchLevels each of which represents a single level of resolution. A PatchLevel holds the distributed Patches, which in turn hold the data representing the state of the problem.

6.2 Major classes and their associations

PatchHierarchy HAS-A: vector of PatchLevel, PatchDescriptor (manages data allocation on all Patches in the hierarchy), BaseGridGeometry (describes the computational domain), and parameters that describe features of the hierarchy, such as the maximum number of levels allowed and the refinement ratios between adjacent levels.

PatchLevel HAS-A: BoxLevel, PatchContainer (map<BoxId, Patch*>), BoxContainer (describes the physical domain using the mesh resolution of the level), a collection of local Patches, BaseGridGeometry, PatchDescriptor, and IntVector refinement ratios describing the relationship to the next coarser level and level zero.

BoxLevel describes a distributed (in the MPI sense) collection of Boxes. Within the PatchLevel, the BoxLevel is used to describe distributed portion of the level's mesh that exists on each processor. BoxLevel is explained in greater detail in Section 7.

Patch is the distributed object in a PatchHierarchy. For any MPI process, the Patches held by a PatchLevel are those that are owned by that process, and the PatchLevel has no built-in means to access the Patches from other processes. PatchLevel's iterator iterates over the Patches that exist on the local process.

Patch HAS-A: Box, PatchDescriptor (shared with owning PatchHierarchy/Level), PatchGeometry (manages relationship between the Patch's index space and location in physical coordinates), vector of PatchData.

6.3 Example

This code example shows nested loops that iterate over all of the levels and patches of a PatchHierarchy.

```
boost::shared_ptr<hier::PatchHierarchy> hierarchy;
…
int num_levels = hierarchy->getNumberOfLevels();
for (int ln = 0; ln < num_levels; ++ln) {
   boost::shared_ptr<hier::PatchLevel> level = hierarchy->getPatchLevel(ln);
```

```
   for (hier::PatchLevel::iterator p(level->begin());
        p != level->end(); ++p) {
      boost::shared_ptr<hier::Patch> patch = *p;
      // put operations on patch here
   }
}
```

7 GridGeometry, PatchGeometry, and specialized geometry classes

SAMRAI geometry classes describe and manage attributes of index spaces used in SAMR hierarchies.

7.1 Key concepts

All grid geometry classes must inherit from BaseGridGeometry, which is an abstract base class but implements much of the general functionality for all grid geometries. A BaseGridGeometry object has a BoxContainer whose Boxes define the full extent of the computational mesh at the coarsest level of resolution for the problem. The maximum allowable extent of the mesh index space on any finer level is represented as a refinement of these Boxes. The BaseGridGeometry class also has methods used during construction of a PatchLevel to determine how each Patch relates to the physical domain boundary. The most basic concrete implementation of BaseGridGeometry is GridGeometry.

Each Patch has a PatchGeometry object that describes the relation of the Patch to the problem boundary. PatchGeometry methods can be queried to determine whether a patch touches a problem boundary, either physical or periodic. Containers of BoundaryBox objects that provide specific information about which Patch cells touch a physical boundary can be retrieved from the PatchGeometry object owned by a Patch as well.

GridGeometry and PatchGeometry have virtual methods but are not abstract base classes. They provide sufficient functionality to understand how Patches relate to each other and problem boundaries as we have just described. In addition, it is often useful to specialize these classes via class inheritance derive to describe the physical coordinates of the mesh. SAMRAI provides CartesianGridGeometry and CartesianPatchGeometry classes derived from these base classes that may be used in problems that employ simple Cartesian meshes. Users that require similar mesh geometry specialization should follow this pattern of creating subclasses of GridGeometry and PatchGeometry.

During construction of a PatchLevel, the PatchGeometry is constructed by the GridGeometry and then set on the Patch using Patch::setGridGeometry(). This functionality should be replicated in any user-defined specialized grid geometry and patch geometry classes.

GridGeometry is also used to manage multiblock metadata, but discussion of that functionality is contained in the Multiblock section.

7.2 Major classes and their associations

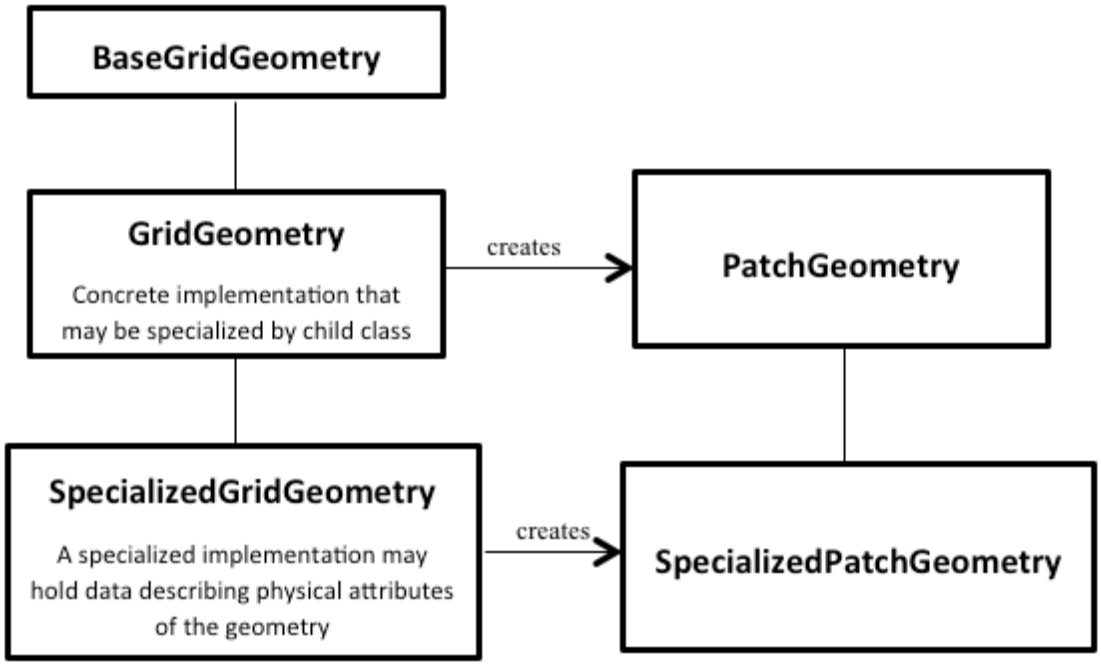


Figure 5: Geometry class associations

BaseGridGeometry HAS-A: Dimension, BoxContainer (boxes describing extent of index space on level zero), IntVector (describing the directions with periodic boundaries, if any). A BaseGridGeometry is passed into the constructor of PatchHierarchy and is constant for the life of that hierarchy.

GridGeometry IS-A: BaseGridGeometry via class inheritance. It contains no additional state data, allowing for the most basic concrete instantiation of a grid geometry.

A SpecializedGridGeometry object (e.g., CartesianGridGeometry) IS-A: GridGeometry via class inheritance. HAS-A: Data relating index space to physical coordinates.

PatchGeometry HAS-A: Dimension, PatchBoundaries (contains an array of BoundaryBoxes), bools telling whether the Patch touches a regular or periodic boundary. A PatchGeometry is created for every Patch and is constant for the life of that Patch.

A SpecializedPatchGeometry object (e.g., CartesianPatchGeometry) IS-A: PatchGeometry via class inheritance. HAS-A: Data relating index space of the Patch to physical coordinates.

7.3 GridGeometry and PatchGeometry object usage

BaseGridGeometry and related classes are important objects for the infrastructure of SAMRAI, but typically there is not expected to be much direct use of the grid geometry in user code. The most

common usage is to ask a GridGeometry object for a container of Boxes that describe the physical problem domain using the methods getPhysicalDomain() or computePhysicalDomain().

The main use of PatchGeometry in user code is to get information about how a Patch touches boundaries of the mesh. The boolean query methods getTouchesRegularBoundary() and getTouchesPeriodicBoundary() are used to indicate whether a Patch touches a certain type of boundary at all. For Patches that touch physical boundaries, the PatchGeometry holds BoundaryBox objects that provide information about how the Patch touches the boundary.

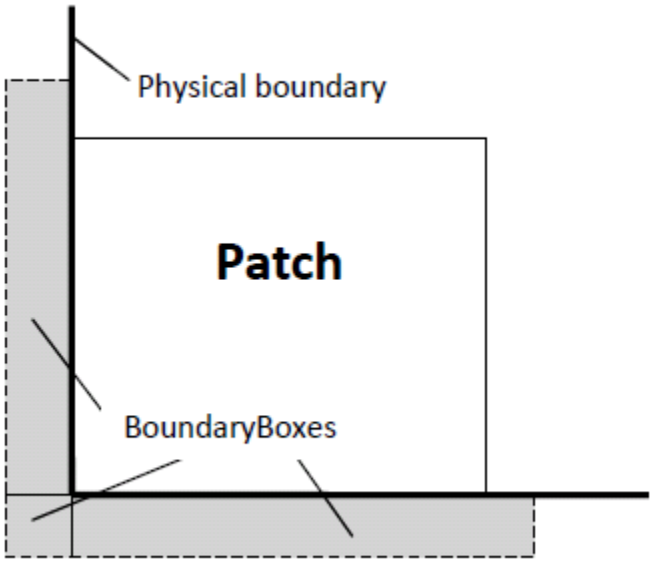


Figure 6: PatchGeometry holds BoundaryBox objects that lie across a physical boundary from a Patch

The BoundaryBox class contains information on how the Patch touches the boundary (along a node, an edge, or a face) and whether the boundary is at the upper or lower extents of the Patch in each coordinate direction. The containers of BoundaryBoxes held by PatchGeometry can be used when setting boundary values for simulation data:

```
const Dimension& dim = patch.getDim();
boost::shared_ptr<PatchGeometry> patch_geom( patch.getPatchGeometry() );

for (int codim = 1; codim <= dim.getValue(); ++codim) {

   const Array< BoundaryBox >& boundary_boxes =
      patch_geom->getCodimensionBoundaries( codim );

   // Code to set values at boundaries with given codimension.

}
```

7.4 Specialization for Cartesian geometries

SAMRAI provides geometry classes that are specializations of the GridGeometry and PatchGeometry to represent Cartesian meshes. These specializations add data for the physical coordinates of the mesh and query functions to retrieve the physical coordinates of the lower and upper corner of a Patch and the mesh spacing increment in each coordinate direction. These coordinates can then be used to compute the physical coordinates of any given point in the mesh. When a PatchHierarchy is constructed with a CartesianGridGeometry object, the geometry object will create a CartesianPatchGeometry object on each Patch with these data initialized properly.

8 Patch distribution and neighbor relationships

This section discusses how SAMRAI represents distributed Patches and provides Patch neighbor and overlap information to application developers. We call this data, and others that describe the mesh, the mesh metadata.

8.1 Key concepts

Recall that each Patch and its data are assigned to a single process in an MPI Communicator. For scalability to large numbers of processes, no single process owns a complete description of the Patch distribution for an AMR hierarchy, typically. By default, SAMRAI only stores the neighbor and overlap Patch relationship information on each process that is required to support parallel inter-patch data communication and adaptive meshing. However, SAMRAI does provide ways to generate a globalized view of this information on each process if needed. This is useful for debugging and when working with applications and other libraries that require each process to have a global view.

8.2 BoxLevel and Connector classes

Two important SAMRAI classes that maintain and provide access to distributed Patch relationships are BoxLevel and Connector.

A BoxLevel object stores a set of boxes, each of which is owned by a process analogous to the way a PatchLevel stores patches. The BoxLevel is the metadata for the PatchLevel. It describes the boxes and the processes they are assigned to. Each Box in a BoxLevel object has a BoxId, which is unique across all processes over which the Boxes are distributed. Each BoxId holds a (MPI) rank and a LocalId. The rank is the owning process of the Box and the LocalId is an identifier that is unique on the local process for the BoxLevel.

We use Connectors to store overlap relationships between two boxes in a BoxLevel or in two different BoxLevels. Overlap relationships are derived from metadata and are considered part of the metadata. A Connector can maintain overlap relationships between Boxes in two BoxLevels, referred to as the "base" and the "head". Currently, a Connector describes only relationships from the base to the head. That is, for each Box in the base, the Connector knows which boxes in the head relate to it. A Connector may hold a pointer to its transpose, the Connector going the other direction. Constructing another Connector with the base and head switched creates relationships going in the other direction. A Connector whose base and head are the same BoxLevel stores relationships between boxes in that BoxLevel.

The overlap relationships described by a Connector are defined by the width of the Connector. The width is an IntVector describing how much a base box has to grow to have an overlap relationship with a head box. Thus, we consider two boxes overlapping if they are within the Connector width of each other. A width of zero means we consider the boxes overlapping only if they actually intersect. A width of one means we consider the boxes overlapping if they merely touch each other. We always specify the width in the index space of the base BoxLevel.

While many Connectors are generated and used within SAMRAI, the Connector objects most important to application developers are those describing overlaps between Patches an AMR hierarchy. When a PatchLevel is built, SAMRAI generates Connectors between its BoxLevel and BoxLevels associated with adjacent levels in the hierarchy. It generates overlaps with widths big enough to support all components that require overlap data to function. This width is a function of the refinement ratio between the levels, the stencil widths of interlevel transfer operators, ghost width of data on the Patches, etc.

Overlap data for each BoxLevel is cached in its PersistentOverlapConnectors object. To access the overlap with another BoxLevel, provide the PersistentOverlapConnector with that BoxLevel and the width defining the overlaps you want.

BoxLevel and Connector objects may be in globalized mode, where each process contains a full representation of the data on all other processes, or distributed mode where each process holds only data associated with the process and what it needs to communicate with other processes. The distributed mode is used most commonly by SAMRAI because constructing a distributed Connector is a scalable operation.

Writer of simple applications similar to the Euler, LinAdv and ConvDiff examples do not need to directly access any metadata. Metadata is used by mesh management code such as GriddingAlgorithm, RefineSchedule and CoarsenSchedule. More advanced applications might have to access BoxLevels and overlap Connectors in the hierarchy.

Developers seeking to duplicate or modify SAMRAI's mesh management code should be familiar the way Connectors are used. For details, see the classes OverlapConnectorAlgorithm and MappingConnectorAlgorithm.

8.3 Basic usage examples

These examples show how to access Boxes in a BoxLevel and overlap data.

How to get the overlap Connector in a PatchHierarchy:

```
const BoxLevel &l0 = *hierarchy.getPatchLevel(0)->getBoxLevel();
const BoxLevel &l1 = *hierarchy.getPatchLevel(1)->getBoxLevel();
const IntVector width( l0.getDim(), 1 );
const Connector &l0_to_l1 = l0.getPersistentOverlapConnectors().
   findConnector( l1, width );
```

How to loop through neighbors in the overlap Connector:

```
for ( Connector::NeighborhoodIterator nhi = l0_to_l1.begin();
      nhi != l0_to_l1.end(); ++nhi ) {
   const Box &base_box = *l0.getBox(*nhi);
```

```
   const BoxContainer neighbors;
   l0_to_l1.getNeigbhorBoxes( base_box.getBoxId(), neighbors );
   // neighbors contains the neighbors of base_box.
}
```

How to access neighbors data for boxes in a BoxLevel. This loop is similar to the previous, but it goes through all base Boxes, whether or not they have neighborhoods in the Connector. The previous loop only sees boxes with neighborhoods.

```
const BoxContainer &base_boxes = l0_to_l1.getBase().getBoxes();
for ( BoxContainer::ConstIterator bi = base_boxes.begin();
      bi != base_boxes.end(); ++bi ) {
   plog << "Base box " << *bi;
   if ( l0_to_l1.hasNeighborSet( bi->getBoxId() ) {
      const BoxContainer neighbors;
      l0_to_l1.getNeigbhorBoxes( bi->getBoxId(), neighbors );
      // neighbors contains the neighbors of *bi.
   }
}
```

9 Variables and patch data

In SAMRAI the concept of variable and patch data are closely related, but have distinct roles within the library, which we describe in this section.

9.1 Key concepts

A Variable is an object that abstractly represents a quantity that can be instantiated on the mesh hierarchy, such as “density” or “flux”. Each variable has a name, and the variable objects tend to be persistent and static through a simulation. Instantiations of Variable subclasses define datatypes and centering. Variable objects do not contain storage for data on the mesh.

This is in contrast to PatchData, which represent the instantiations of data on a particular hierarchy. PatchData objects are dynamic throughout the simulation as the solution and mesh change. The PatchData class, like the Variable class, is as an abstract base class, which provides the interface which define how a particular type of data is to be manipulated on the hierarchy, for example how it can be copied and communicated. In fact, the PatchData interface is the complete collection of methods that SAMRAI uses to manipulate data on the hierarchy. If you can implement this interface, SAMRAI can interoperate with your data. The implementations for such manipulation will depend on, for example, the centering of the data on the mesh, whether it is multi-valued, and the data types used such as integer or floating-point representations.

As a convenience, SAMRAI comes with a number of commonly used Variable and PatchData types already implemented, such as CellData and NodeData. However, it is a key feature of the library that the user can implement any data type they wish, so long as they implement the PatchData interface. Some examples of possible custom user-defined Variable/PatchData types might be a data type to represent a field of particles, or a quaternion-valued field. Note that many users of the library who are not doing anything particularly esoteric will not need to implement any custom data types in this manner. However, if a user is using SAMRAI to retrofit an existing simulation code, it will be common that they will want to create their own custom PatchData/Variable data types to match the way data is organized in the simulation code.

9.2 Major classes and their associations

Variable, PatchDataFactory, and PatchData are all abstract base classes that must have corresponding concrete subclasses. A Variable subclass, e.g. CellVariable, creates the corresponding PatchDataFactory subclass, such as a CellPatchDataFactory. The role of a subclass of PatchDataFactory is to create instances of a particular type of PatchData, in this case the CellData class. Similarly related triples of classes exist for each of the SAMRAI supplied data types, for example: NodeData, EdgeData, etc.

Variable (base class) HAS-A: dimension, name, instance id, PatchDataFactory.

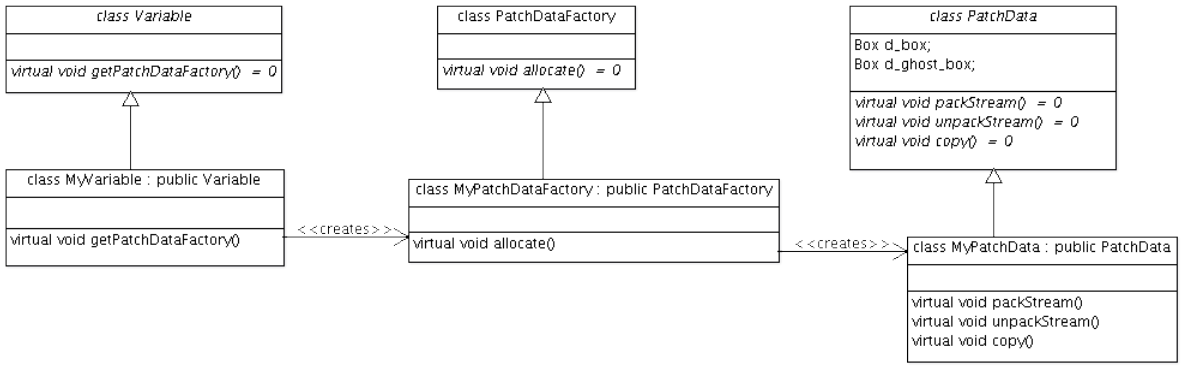
NOTE: Variable has-a dimension implies that if one uses multiple hierarchies with different dimensions, then different (dimension-dependent) variables are needed.

PatchDataFactory (base class) HAS-A: IntVector (ghost width)

PatchData (base class) HAS-A: Box (box and ghost box), IntVector (ghost width), double (timestamp)

NOTE: That a variable, by itself, is insufficient to allocate a PatchData object. The ghost width is needed which is provided by a Factory (refer to VariableDatabase discussion below).

The following diagram sketches the relationships between the three base classes, and the three required concrete subclasses for a fictional user implemented datatype MyPatchData associated with a variable MyVariable. This is only a sketch that highlights the most conceptually important relationships, methods, and data members between these triples of classes.



9.3 Variable context, variable database, and data id

The concept of a "variable context" is very important, although not entirely self-describing and often unclear to a new user of the library. A variable context is essentially a named copy of storage associated with a particular variable that is usually reserved for a particular use or function in the simulation. For example, one might have two copies of a CellData field, one which represents the data at the current simulation time $t_{n,}$ and one which represents the values at an advanced simulation time t_{n+1}. In this case the user could use two variable contexts and name them (for example) "current" and "new". It is also important to note that two different contexts of the same variable may have differing ghost cell widths (including zero or no ghost cells).

The VariableDatabase is a convenient utility that allows the user to maintain and access a collection of variables and associated contexts. You can think of the VariableDatabase as being a table, with named variables along one axis and named contexts along the other axis.

VariableDatabase	"current"	"new"	"scratch"
"density"	1	2	3
"pressure"	4	5	6
"velocity"	7	8	

For each combination of variable and context, there is a unique "data id" which identifies that particular instance of data. These data id's are created and maintained internally by the VariableDatabase as variable-context pairs are registered with the database. These data ids can be used to fetch particular instances of PatchData from a patch using the Patch::getPatchData(int data_id) method.

Note that not all variable-context pairs need to be registered or have data ids associated with them. In the example table above, there happens to be no need for a "scratch" context for the variable "velocity".

9.4 Basic usage example

```
/*
 * This example shows how Variable, VariableContext, the
 * VariableDatabase, and a Patch work together to define and
 * instantiate PatchData.
 */

#include "SAMRAI/hier/VariableDatabase.h"
#include "SAMRAI/pdat/CellVariable.h"
#include "SAMRAI/pdat/CellData.h"
#include "SAMRAI/pdat/NodeVariable.h"
#include "SAMRAI/tbox/SAMRAIManager.h"
#include "SAMRAI/tbox/SAMRAI_MPI.h"
using namespace SAMRAI;

#include <boost/shared_ptr.hpp>
using namespace boost;

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
   tbox::SAMRAI_MPI::init(&argc, &argv);

   const tbox::Dimension dim(2);

   /*
    * Create two variables, one scalar with cell centering, and one
    * vector with node centering
    */

   shared_ptr< pdat::CellVariable<double> > density(
      new pdat::CellVariable<double>(dim, "density") );

   shared_ptr< pdat::NodeVariable<double> > velocity(
      new pdat::NodeVariable<double>(dim, "velocity", 2) );

   // Create three contexts

   shared_ptr< hier::VariableContext > current(
```

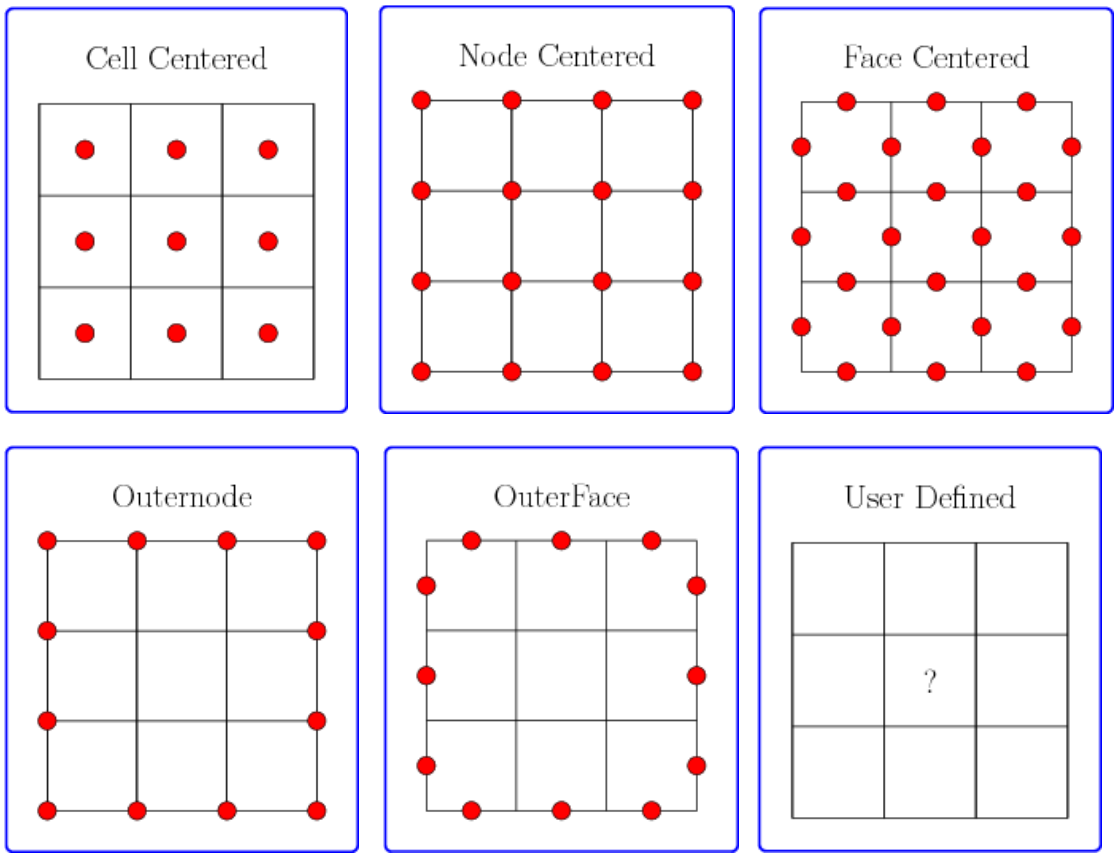
```
   hier::VariableDatabase::getDatabase()->getContext("current") );

shared_ptr< hier::VariableContext > scratch(
   hier::VariableDatabase::getDatabase()->getContext("scratch") );

shared_ptr< hier::VariableContext > special(
   hier::VariableDatabase::getDatabase()->getContext("special") );

/*
 * Create a variable database, and register some variable-context
 * pairs with it.
 */

hier::VariableDatabase* var_db = hier::VariableDatabase::getDatabase();

hier::IntVector gw0 = hier::IntVector::getZero(dim);
hier::IntVector gw1 = hier::IntVector::getOne(dim);

int d_cur_id = var_db->registerVariableAndContext(density, current, gw0);
int v_cur_id = var_db->registerVariableAndContext(velocity, current, gw0);

int d_scr_id = var_db->registerVariableAndContext(density, scratch, gw1);
int v_scr_id = var_db->registerVariableAndContext(velocity, scratch, gw1);

int v_sp_id = var_db->registerVariableAndContext(velocity, special, gw1);

/*
 * In order to instantiate data, we need a patch to instantiate it
 * on.  In a more realistic application, we would typically obtain
 * the Patch from a PatchLevel which in turn would be obtained from
 * a PatchHierarchy.  In order to limit the scope of this example,
 * we manually create an isolated patch here that is not on a
 * hierarchy.
 */

/*
 * In order to create a patch, we need a box and a PatchDescriptor.
 * The PatchDescriptor describes the types of data that can be
 * instantiated on the patch; it is essentially a collection of
 * PatchDataFactories.  This is information that we have supplied
 * to the variable database through our variable-context pair
 * registration calls.
 */

hier::BlockId block_id(0);
const hier::Box box(hier::Index(0,0), hier::Index(3,3), block_id);

hier::Patch patch(box, var_db->getPatchDescriptor());

/*
 * Once we have a patch, we can allocate our PatchData on it.  Note
 * that in the usage pattern shown here, the use of the
 * PatchDataFactory is hidden from the user, as the
 * PatchDataFactory objects are stored with the patch inside the
 * PatchDescriptor.  Internally, allocatePatchData() uses the
 * PatchDescriptor to look up the correct factory corresponding to
```

```
     * the supplied data_id, and uses it to do the PatchData
     * allocation.
     */

    patch.allocatePatchData(d_cur_id);
    patch.allocatePatchData(v_cur_id);

    patch.allocatePatchData(d_scr_id);
    patch.allocatePatchData(v_scr_id);

    patch.allocatePatchData(v_sp_id);

    /*
     * The newly allocated PatchData can now be pulled off of our patch
     * to work on.  We can identify the PatchData we want through the
     * data_id assigned by the variable database.
     */

    shared_ptr< hier::PatchData > dendata(
       patch.getPatchData(d_cur_id) );

    shared_ptr< hier::PatchData > veldata(
       patch.getPatchData(v_cur_id) );

    // Work with the patch data...

    tbox::SAMRAIManager::shutdown();
}
```

9.5 Variable and PatchData types in SAMRAI

SAMRAI supplies a number of Variable and PatchData types that are ready to use. They are:

- CellData
- NodeData, OuternodeData
- FaceData, OuterfaceData
- EdgeData, Outeredge Data
- SideData, OutersideData
- SparseData



The "outer" versions of data do not instantiate data on the interior of patches. These would typically be used for managing the interaction of data between a coarser and a finer level, such as the handling of flux mismatches. FaceData and SideData are similar, only differing in the ordering of the data layout.

Each PatchData type supplied with the library contains a corresponding iterator class that can be used to traverse the data in a uniform way. The conventions employed are designed to be reasonably similar to those used within the STL. An example iterator usage is:

```
for (pdat::CellData<double>::iterator ic(pdat::CellGeometry::begin(box));
     ic != pdat::CellGeometry::end(box); ++ic) {
  data(*ic) = 1.0;
}
```

9.6 User defined patch data types

Implementation of a custom patch data type requires creating a number of derived classes, each of which implement abstract operations that SAMRAI will call in order to do its work with your data on an AMR hierarchy. The classes that need to be implemented that we have already discussed are:

- Variable
- PatchDataFactory
- PatchData

In addition, a custom variable with a custom centering or other non-standard feature may also need a custom way of describing how the data overlaps when patches it resides on intersect. The interfaces that may need to be implemented are:

- BoxOverlap
- VariableFillPattern

VariableFillPattern subclasses create BoxOverlap subclasses that describe the details of what data needs to be copied or communicated when two patches intersect. For additional details regarding the use of overlaps and fill patterns, refer to Chapter 10, in particular Section 10.4.

10 Patch hierarchy construction and adaptive meshing

The hierarchy is a container of levels, with methods for inserting and deleting levels. One can set up the hierarchy manually using these methods, but it is tedious to do for all but the simplest of configurations.

Most users would use the GriddingAlgorithm to set up and adapt the mesh hierarchy. The GriddingAlgorithm efficiently sets up the hierarchy while ensuring its has the configurations appropriate for working with other SAMRAI components (such as the transfer schedules).

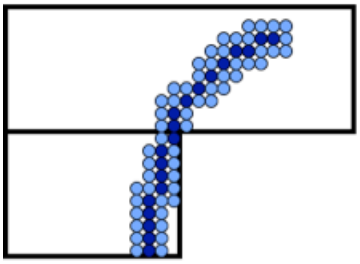
This section covers the primitive methods for building the hierarchy but assumes the user will use the GriddingAlgorithm for mesh generation and adaptive meshing.

10.1 Key concepts

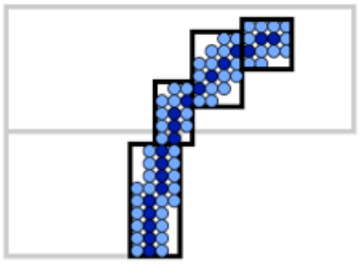
Mesh generation consists of several major steps:

1. Tagging: identify cells that should be refined
2. Clustering: compute a set of boxes containing those cells
3. Box adjustments: make slight adjustments to ensure nesting and other requirements
4. Partitioning: partition the boxes to achieve balanced loads in parallel.
5. Construction: construct the new level
6. Initializing: populate the new level with solver data.

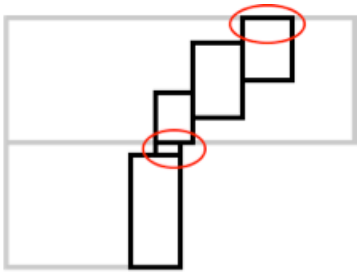
The first four steps are illustrated below.



1. Tag cells



2. Cluster



3. Box adjustments

4. Partition

Mesh adaptation is very similar to mesh generation in that a new level is generated to replace the current level. In adaptation, we use solver data from the current level as well as coarser levels to initialize the new level. In mesh generation, there is no current level at the same resolution, so all solver data comes from the coarser levels.

The GriddingAlgorithm uses some abstract interfaces to interact with interchangeable components: tagging, clustering, partitioning and initializing level data.

10.2 Major classes and their associations

Major classes are the PatchHierarchy, GriddingAlgorithm and a set of strategies used by GriddingAlgorithm.

We construct the PatchHierarchy one level at a time, using the method makeNewPatchLevel(). This method takes a BoxLevel (metadata describing the boxes in the new level) and constructs a new PatchLevel in the hierarchy. To remove a level from the hierarchy, use the method removePatchLevel().

The GriddingAlgorithm provides higher-level interfaces for generating and adapting the hierarchy. GriddingAlgorithm's main interfaces are:

1. makeCoarsestLevel(): make the coarsest level
2. makeFinerLevel(): make the next finer level in the hierarchy, increasing the number of levels by one.
3. regridAllFinerLevels(): regrid all levels above a given level.

These methods not only calls makeNewPatchLevel() but performs all the mesh generation and adaptation steps described above. Moreover, they ensure that the hierarchy configuration satisfies proper requirements for simulations.

Each GriddingAlgorithm objects holds a pointer to the hierarchy it operates on and pointers to implementations of strategies used by the mesh generation steps.

<UML showing GrididngAlgorithm, PatchHierarchy, BoxGeneratorStrategy, LoadBalanceStrategy and TagAndInitializeStrategy>

The roles of the strategies are:

1. BoxGeneratorStrategy: Defines interface for clustering tags into non-overlapping boxes.
2. LoadBalanceStrategy: Defines interface to redistribute a set of boxes among the MPI tasks to achieve a balanced load. The boxes may be cut into smaller boxes in the process.
3. TagAndInitializeStrategy: Defines interface for tagging cells for refinement, initializing data on a new level.

The strategy implementations are used in mesh generation. First, the TagAndInitializeStrategy identifies the cells that the application wants to be refined. The tag data is a cell-centered integer PatchData and

living on the next coarser level. The BoxGeneratorStrategy generates a set of non-overlapping boxes containing the tagged cells and as few untagged cells as feasible. The GriddingAlgorithm makes small adjustments to the evolving boxes to ensure they are suitable for simulation. For example, they must follow proper nesting rules. Now, we have a BoxLevel defining exactly the extent of the new level. The LoadBalanceStrategy redistributes the boxes, possibly cutting up some boxes to a portion of their work to other processes. After load balancing, we have the box configuration for the new level. The GriddingAlgorithm uses makeNewPatchLevel() to generate the level. The TagAndInitializeStrategy initializes the level data.

10.3 Basic usage examples

To manually create a PatchHierarchy with one level:

```
boost::shared_ptr<BaseGridGeometry> base_grid_geometry =
   new BaseGridGeometry( dim,
                         “My geometry”,
                         geometry_input_database );
PatchHierarchy hierarchy(“My hierarchy”,
                         base_grid_geometry,
                         hierarchy_input_database);
BoxLevel box_level( ratio,
                    base_grid_geometry,
                    SAMRAI_MPI::getSAMRAIWorld() );

hierarchy.makeNewPatchLevel( 0, box_level );
```

To use GriddingAlgorithm to build a PatchHierarchy with the maximum number of levels:

```
boost::shared_ptr<BaseGridGeometry> base_grid_geometry =
   new BaseGridGeometry( dim,
   “My geometry”,
   geometry_input_database );
boost::shared_ptr<PatchHierarchy> hierarchy =
   new PatchHierarchy( “My hierarchy”,
                       base_grid_geometry,
                       hierarchy_input_database );

boost::shared_ptr<TagAndInitializeStrategy> level_strategy = ...;
boost::shared_ptr<BoxGeneratorStrategy> generator = ...;
boost::shared_ptr<LoadBalancerStrategy> balancer = ...;

GriddingAlgorithm gridding_algorithm(
   hierarchy,
   “My gridding algorithm”,
   gridding_algorithm_database,
   level_strategy,
   generator,
   balancer );
```

```
Array<int> tag_buffer( hierarchy->getMaxNumberOfLevels(), 2 );
gridding_algorithm.makeCoarsestLevel( 0.0 );
while ( hierarchy->getNumberOfLevels() <
        hierarchy->getMaxNumberOfLevels() ) {
   gridding_algorithm.makeFinerLevel(
      0.0,
      true,
      tag_buffer[hierarchy->getFinestLevelNumber()] );
}
```

To adapt the above hierarchy after doing some time integration:

```
// Integrate a few time steps.
gridding_algorithm.regridAllfinerLevels( 0, regrid_time, tag_buffer );
```

10.4 Customizing algorithmic features

Of the three strategies used by GriddingAlgorithm, SAMRAI provides off-the-shelf implementations for BoxGeneratorStrategy and LoadBalancerStrategy. Only TagAndInitializeStrategy requires user customization. At the very least, users need to implement TagAndInitializeStrategy::tagCellsForRefinement() and TagAndInitializeStrategy::initializeLevelData() because there is no generic implementations for those. Method tagCellsForRefinement() implements a user-defined algorithm to determine which cells on a given level should be refined. Method initializeLevelData() is a call-back that SAMRAI uses whenever a level is created either during the initial hierarchy construction or while regridding. It should allocate and populate patch data on the level. On fine levels, this is usually done with a RefineSchedule.

11 Patch data communication conceptual overview

11.1 Key concepts

Data communication is handled through the infrastructure for refining and coarsening. Both refining and coarsening are managed through a set of classes that apply largely the same concepts to each case.

- **Algorithm** – An Algorithm class is used to describe and manage the communication that is to be done at a specific point during the execution of the application. It knows about which data are to be communicated and what operations will be executed, but it is independent of the particular state of the hierarchy.
- **Schedule** – A Schedule class is used to execute the communication between two specific PatchLevels. It is constructed by the Algorithm and can exist only as long as its source and destination levels exist.
- **Operator –** Operator classes are used by the Schedules to move data between meshes of different resolutions. They are written to execute a specific numerical method of coarsening or refinement, such as averaging for coarsening or linear interpolation for refinement.
- **PatchStrategy –** PatchStrategy abstract base classes are provided as a means for the user to optionally add some application-specific operations to be called during the execution of a Schedule.
- **Fill patterns –** Optional tools that can be used to enforce restrictions on where data is communicated onto a destination level

These concepts all exist for both refining and coarsening under a parallel structure of classes. There are RefineAlgorithm, CoarsenAlgorithm, RefineSchedule, CoarsenSchedule, etc.

11.2 RefineAlgorithm and CoarsenAlgorithm

The Algorithm objects are used to describe at a high level the operations that will be used in communication at specific points in the execution of an application and are independent of the specific layout of any PatchHierarchy. Algorithm objects are usually created during setup of the application, and registration methods are called to register the data that will be communicated and the operators that will be used to refine or coarsen.

```
xfer::RefineAlgorithm my_algorithm;
my_algorithm.registerRefine(dst_id,     // patch data id for destination data
                            src_id,     // patch data id for source data
                            scratch_id, // patch data id for scratch data
                            refine_op); // operator for refinement
```

registerRefine() (or the corresponding method registerCoarsen()) can be called any number of times on a single Algorithm object.

Note that the refining-related methods presented in the code examples in this chapter are single examples of overloaded methods that have multiple usage options. Please see the class documentation for all possible options and to see the analogous methods in the coarsening-related classes.

11.3 RefineSchedule and CoarsenSchedule

While the Algorithms are used to describe the properties of the communication operations that are independent of the mesh configuration, the Schedule objects manage and execute the communication between two PatchLevels that exist at a specific point in the run of an application. Schedules are built using the createSchedule() methods in the Algorithm classes.

```
boost::shared_ptr<xfer::RefineSchedule> my_schedule =
   my_algorithm.createSchedule(dst_level,           // destination level
                               src_level,           // source level
                               next_coarser_level,  // int level number of
                                                    // next coarser level
                               hierarchy,           // hierarchy where
                                                    // destination exists
                               patch_strategy);     // strategy for user
                                                    // defined operations
```

In this example the destination and source levels are of the same resolution (such as when moving data from an old level to a new level when regridding). The schedule will set up transaction to copy data from the source to the destination wherever they overlap, and any parts of the destination that do not overlap the source will be filled from the next coarser level of the hierarchy using the refinement operators that were registered with the Algorithm object.

Once a Schedule is created, the communication is invoked using the fillData() method for RefineSchedule or the coarsenData() method for CoarsenSchedule. These communication methods can be called any number of times as long as the Schedules are valid. The Schedules become invalid when either their source or destination levels are destroyed.

11.4 RefineOperator and CoarsenOperator

RefineOperator and CoarsenOperator are abstract base classes that define the interface for the refinement or coarsening of data. Operator objects are passed into the registration methods of the Algorithm classes, providing the Operator that will be used on the particular data being registered. Each concrete implementation of RefineOperator or CoarsenOperator is an implementation of a numerical algorithm for refining or coarsening a specific type of PatchData. SAMRAI provides a number of implementations of the Operator classes, and users may also write their own.

11.5 RefinePatchStrategy and CoarsenPatchStrategy

These strategy classes are abstract base classes that provide an interface for user-defined operations to occur during the execution of a communication. The main pure virtual methods provided are preprocess and postprocess methods that are called on each patch before and after the calls to the Operators' refine() or coarsen() methods. These allow users to implement any application-specific

manipulation of internal data that may be needed before and after the inter-level operations of the Operators. RefinePatchStrategy also provides an interface that is used for the filling of ghost data around physical boundaries, which is described in more detail in the chapter on refinement.

11.6 Fill patterns

The use of fill patterns is an optional way to add some restrictions that limit which parts of the destination level receive data during communication. There are two types of fill patterns, each of which is defined by an abstract base class, PatchLevelFillPattern and VariableFillPattern.

A PatchLevelFillPattern, used only in refinement communication, is given to a RefineSchedule at its construction and limits to communication of all data to a specific portion of the destination level. For example, it may be desired to only communicate data onto ghosts at coarse-fine boundaries; PatchLevelBorderFillPattern is an implementation of PatchLevelFillPattern that enforces this restriction.

A VariableFillPattern, which can be used both in refining and coarsening, is associated with each data item that is registered with the Algorithm. (Thus each data item can have a different VariableFillPattern, or none at all.) VariableFillPatterns are used to restrict the filling on the destination patches to a specific part of each patch. A user may want to fill a certain NodeData object only on the nodes on the patch boundary, or may want to fill a certain CellData object only in the first layer of ghost cells. Implementations of VariableFillPattern usually are specialized for a specific data centering

12 Data refinement communication

12.1 Key concepts

Refinement communication is typically used for two primary purposes: for filling ghost regions on a level, and for regridding. Refinement communication is conceptually split into three main parts, called the algorithm, the schedule, and the patch strategy. These concepts were described in Section 11.1.

12.2 Major classes and their associations

The major classes associated with refinement are RefineAlgorithm, RefineSchedule, RefineOperator, RefinePatchStrategy, PatchLevelFillPattern, and VariableFillPattern. These classes were discussed in the prior Section 11.2.

12.3 Basic usage example

A common use case for refinement is to fill the ghost cells on a finer level, using data from the same level and data interpolated from coarser levels at coarse fine boundaries. In this snippet we're showing how to fill the variable "u" on level 1 using data interpolated from level 0 using a built-in refine operator CartesianCellDoubleLinearRefineOperator. In this example, the situation is XXX, but YYY. We are assuming level 0 is at the same simulation time as level 1, and are therefore not invoking time refinement. First we set up the variable, contexts, and data_ids. Then, we show how to use registerRefine to set up the algorithm, createSchedule to construct the schedule for filling the destination level, and finally we call fillData() to execute the schedule and fill the ghost cells.

```
/*
 * Create a scalar variable with cell centering.
 */
shared_ptr< pdat::CellVariable<double> > u(
   new pdat::CellVariable<double>(dim, "u") );

/*
 * Create two contexts, named cur (current) and scr (scratch)
 */
shared_ptr< hier::VariableContext > cur(
   hier::VariableDatabase::getDatabase()->getContext("cur") );

shared_ptr< hier::VariableContext > scr(
   hier::VariableDatabase::getDatabase()->getContext("scr") );

/*
 * Create a variable database, and register some variable-context
 * pairs with it.
 */
hier::VariableDatabase* var_db = hier::VariableDatabase::getDatabase();

hier::IntVector gw0 = hier::IntVector::getZero(dim);
hier::IntVector gw1 = hier::IntVector::getOne(dim);

int cur_id = var_db->registerVariableAndContext(u, cur, gw0);
```

```
int scr_id = var_db->registerVariableAndContext(u, scr, gw1);
…
// Note that for brevity we are not showing some steps like the creation of
// the hierarchy and the allocation of data.

shared_ptr<hier::RefineOperator> refine_op(
   grid_geom->lookupRefineOperator(u, "LINEAR_REFINE") );

xfer::RefineAlgorithm fill_ghosts_alg;

fill_ghosts_alg.registerRefine(scr_id,     // destination data
                               cur_id,     // source data
                               scr_id,     // scratch space
                               refine_op); // refine operator

shared_ptr<xfer::RefineSchedule> fill_ghosts_L1;

int fill_level_num = 1;
int next_coarser_level_num = 0;
fill_ghosts_L1 = fill_ghosts_alg.createSchedule(
hierarchy->getPatchLevel(fill_level_num),
   next_coarser_level_num,
   hierarchy,
   patch_strategy);

See example RefineCommunication.

CALL FILLDATA
```

12.4 Fill patterns found in SAMRAI

Fill patterns were discussed in Section 11.6. The following figures illustrate the destination of the built-in variable fill patterns if they were used on a patch of 2x2 cells, and a destination that has a ghost width of two.

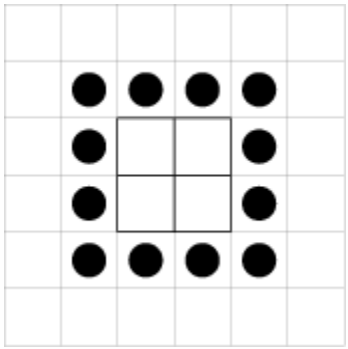


Figure 7: FirstLayerCellVariableFillPattern

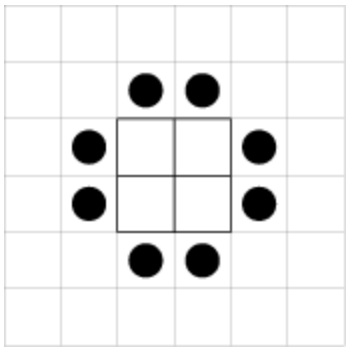


Figure 8: FirstLayerCellNoCornersVariableFillPattern

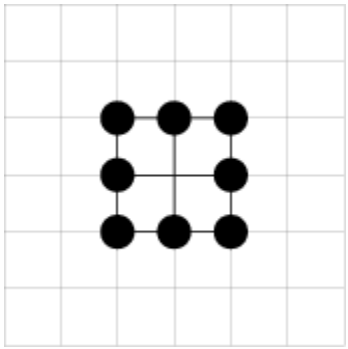


Figure 9: FirstLayerNodeVariableFillPattern

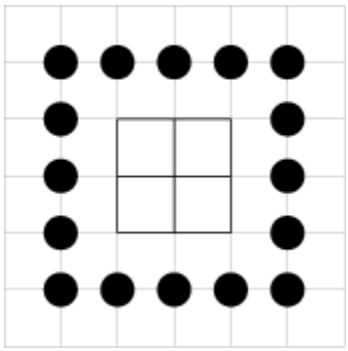


Figure 10: SecondLayerNodeVariableFillPattern

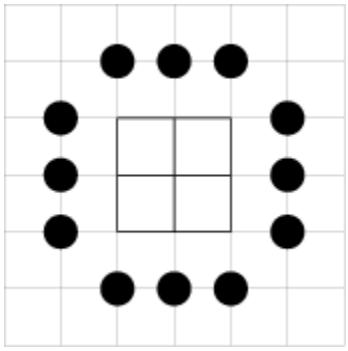


Figure 11: SecondLayerNodeNoCornersVariableFillPattern

12.5 Refine operators found in SAMRAI

The built-in operators in SAMRAI can be divided into two categories: those that are associated with the CartesianGridGeometry and those that are generic with respect to the GridGeometry (MENTION COORDINATE SYSTEM). In addition to these two categories, the operators are associated with a centering (Cell, Node, etc.), a datatype (double, float, complex) and a refinement method (constant, linear, conservative, etc.).

Operators that don't depend on the coordinate system:

Where |Centering| is one of Cell, Node, Edge, Side, Face, and <Type> is one of Float, Double, Complex, or Integer:

|Centering||Type|ConstantRefine: Implements a constant refinement, where interpolated values are equal to the coarse value.

Operators that work on Cartesian grids:

Cartesian|Centering||Type|LinearRefine: Implements a linear refinement, where interpolated values are derived from the coarse value and a linear slope determined from coarse data.

Cartesian|Centering||Type|ConservativeLinearRefine: Implements a conservative linear refinement, where interpolated values are derived from the coarse value and a linear slope determined from coarse data.

13 Data coarsening communication

13.1 Key concepts

As the simulation results for any location on an AMR mesh have the greatest accuracy on the finest level representing that location, it is necessary to coarsen these accurate results to the less accurate coarser levels. The key concepts concerning data coarsening communication have already been described in Chapter 11, Patch data communication.

13.2 Major classes and their associations

The major data coarsening communication classes are CoarsenAlgorithm, CoarsenSchedule, CoarsenOperator, CoarsenPatchStrategy, and VariableFillPattern. They have all been discussed in Chapter 11. Details about the CoarsenOperators will be supplied in Section 13.4 below.

13.3 Basic usage examples

```
boost::shared_ptr<hier::PatchHierarchy> hierarchy;
boost::shared_ptr<hier::Variable> var;
boost::shared_ptr<hier::VariableContext> ctx;
tbox::Dimension dim;
…

/*
 * Register the given variable, var, with context, ctx, obtaining the patch
 * data index.
 */
hier::VariableDatabase* variable_db = hier::VariableDatabase::getDatabase();
const hier::IntVector zero_ghosts(dim, 0);
int new_id = variable_db->registerVariableAndContext(var, ctx, zero_ghosts);

/*
 * Create a CoarsenAlgorithm and register this data for coarsening.
 */
boost::shared_ptr<hier::CoarsenOperator> coarsen_op =
   hierarchy->getGridGeometry()->lookupCoarsenOperator(var,
      "CONSERVATIVE_COARSEN");
xfer::CoarsenAlgorithm coarsen_alg(dim);
coarsen_alg.registerCoarsen(new_id, new_id, coarsen_op);
…

/*
 * Coarsen data from finest level with coarser levels.  Create a
 * CoarsenSchedule for a level and its immediately coarser level and coarsen
 *the data.
 */
for (int fine_ln = finest_ln; fine_ln > coarsest_ln; --fine_ln) {
   const int coarse_ln = fine_ln - 1;
```

```
   boost::shared_ptr<hier::PatchLevel> fine_level(
      hierarchy->getPatchLevel(fine_ln);
   boost::shared_ptr<hier::PatchLevel> coarse_level(
      hierarchy->getPatchLevel(coarse_ln);

   boost::shared_ptr<xfer::CoarsenSchedule> coarsen_sched =
      coarsen_alg.createSchedule(coarse_level, fine_level);
   coarsen_sched->coarsenData();
}
```

13.4 Coarsen operators found in SAMRAI

Each implementation of the CoarsenOperator interface provided by the library is described below. Applications are free to create their own by implementations by creating a class derived from CoarsenOperator and implementing that base class' pure virtual methods.

geom::CartesianCell[Complex, Double, Float]WeightedAverage: conservative cell-weighted averaging for cell-centered [complex, double, float] data defined on a Cartesian mesh

geom::CartesianEdge[Complex, Double, Float]WeightedAverage: conservative edge-weighted averaging for edge-centered [complex, double, float] data defined on a Cartesian mesh

geom:: CartesianFace[Complex, Double, Float]WeightedAverage: conservative face-weighted averaging for face-centered [complex, double, float] data defined on a Cartesian mesh

geom:: CartesianOuterface[Complex, Double, Float]WeightedAverage: conservative face-weighted averaging for outerface [complex, double, float] data defined on a Cartesian mesh

geom:: CartesianOuterSideDoubleWeightedAverage: conservative side-weighted averaging for outerside double data defined on a Cartesian mesh

geom::CartesianSide[Complex, Double, Float]WeightedAverage: conservative side-weighted averaging for side-centered [complex, double, float] data defined on a Cartesian mesh

pdat::Node[Complex, Double, Float, Integer]Injection: constant injection for node-centered [complex, double, float, integer] data

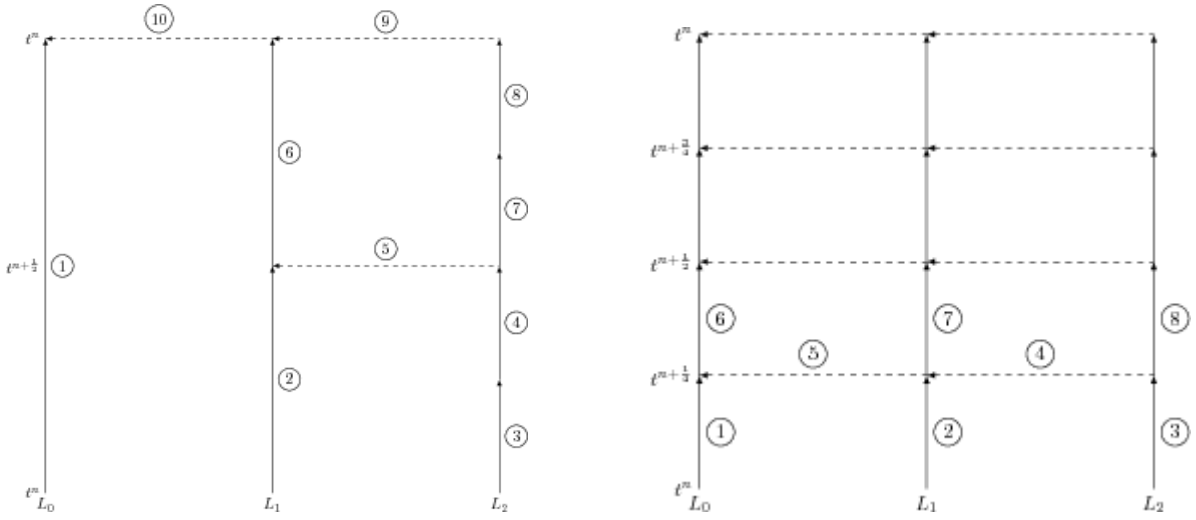
pdat::OuternodeDoubleInjection: constant injection for outernode-centered double data

14 Time integration and solvers

SAMRAI provides a number of classes that implement some common AMR algorithms, including time integration and solvers on an AMR hierarchy. While the framework supplies implementations of these algorithms for common use case scenarios as a convenience, the developer is free to implement their own algorithm and is not constrained to use those provided.

14.1 Time integration methods

Perhaps the class most likely to be used by an application code that uses explicit time integration is the TimeRefinementIntegrator. It should be noted that the TimeRefinementIntegrator is more flexible than its name implies, in that it can operate both in a time-refined mode, where finer grids take smaller time steps that coarser grids, and also in a synchronized mode, where time steps are matched at all levels in the hierarchy. The difference between these two modes is illustrated here:



An application code can use TimeRefinementIntegrator by implementing the strategy class TimeRefinementLevelStrategy, which contains the following pure virtual methods: initializeLevelIntegrator, getLevelDt, getMaxFinerLevelDt, advanceLevel, standardLevelSynchronization, synchronizeNewLevels, resetTimeDependentData, resetDataToPreadvancedState, and usingRefinedTimeStepping.

Note that this class manages the sequencing of steps on the various AMR levels, but is agnostic to any particular integration scheme or process. This provides the flexibility needed for use with a wide variety of application codes.

An implementation of TimeRefinementLevelStrategy is provided via the HyperbolicLevelIntegrator, which in turn relies on a user-supplied implementation of HyperbolicPatchStrategy to provide the implementations for the pure virtual methods registerModelVariables, initializeDataOnPatch, computeStableDtOnPatch, computeFluxesOnPatch, conservativeDifferenceOnPatch, and setPhysicalBoundaryConditions.

The HyperbolicLevelIntegrator and the TimeRefinementIntegrator taken together implement the classical Berger, Colella, Oliger AMR integration method.

The MethodOfLinesIntegrator, by contrast, is an implementation of a specific time integration scheme for a system of ODEs, in particular, the Strong Stability Preserving (SSP) Runge-Kutta schemes. An application can use this algorithm by implementing the MethodOfLinesPatchStrategy, which contains the following pure virtual functions: registerModelVariables, initializeDataOnPatch, computeStableDtOnPatch, singleStep, tagGradientDetectorCells, and setPhysicalBoundaryConditions.

14.1.1 External packages

SAMRAI also provides interfaces to the CVODE external package (a component of the Sundials suite) for performing time integration on a system of ODEs. In order to use the CVODE solver, the user can provide an implementation of the CVODEAbstractFunctions interface, namely the pure virtual methods evaluateRHSFunction, CVSpgmrPrecondSet, and CVSpgmrPrecondSolve. See the CVODESolver.h header for additional details.

14.2 Basic usage examples

The usage of these algorithm classes is best explained by studying simple but functional examples.

The linear advection example LinAdv is a good example of the use of TimeRefinementAlgorithm in conjunction with HyperbolicLevelIntegrator. A more complex example of using a system of equations can be found in the Euler example.

The MethodOfLinesIntegrator algorithm has an example implementation in the convection-diffusion example ConvDiff.

The CVODE package interface has an example implementation in the sundials/ directory, namely CVODEModel.

14.3 Linear and nonlinear solvers

SAMRAI provides a number of classes for performing linear and nonlinear solves on AMR hierarchies. Some of these classes are interfaces to external packages, and some implement solution methods directly.

14.3.1 Linear solvers

HYPRE is an external library for solving linear systems in parallel. It is also used internally to some other solver implementations such as the FAC solver and the CellPoissonHypreSolver.

FACPreconditioner is an implementation of the FAC iterative solution procedure for a solution of a linear system over an AMR hierarchy. This class uses the FACOperatorStrategy interface, for which users must implement a number of pure virtual methods. See FACOperatorStrategy.h for details.

14.3.2 Nonlinear Solvers

Sundials is an external suite of tools for solving nonlinear and differential/algebraic systems. CVODE, a package for integration of systems of ODEs, was already mentioned in the Section 14.1.1. SAMRAI also provides wrappers for the KINSOL package, for solving nonlinear algebraic systems, through the class KINSOLSolver. The user must supply implementations for the interface KINSOLAbstractFunctions. SundialsAbstractVector is an interface that allows the user to provide custom vector kernel operations. See the associated header files for details.

PETSc is a more general scientific computation toolkit for which SAMRAI provides an interface for a user-supplied vector class that can be used with the PETSc solver framework. In addition, SAMRAI provides PETSc_SAMRAIVectorReal, which is an implementation of PETScAbstractVectorReal which wraps a SAMRAIVectorReal, which allows a collection of real-valued patch data types to be manipulated as though they are all part of a single vector.

14.4 Basic usage examples

The linear and nonlinear solver facilities are best understood by studying small but functional examples, which can be found in the following locations:

Solver Package	Location of Example Code
HYPRE	test/hypre
CVODE	test/sundials
KINSOL	test/nonlinear
PETSc vectors	test/vector
PETSc solvers	test/nonlinear

15 Multiblock meshes

SAMRAI allows for the problem domain to be constructed as a multiblock mesh, with the mesh being composed of a number of logically rectangular blocks that when combined may not be logically rectangular on the whole.

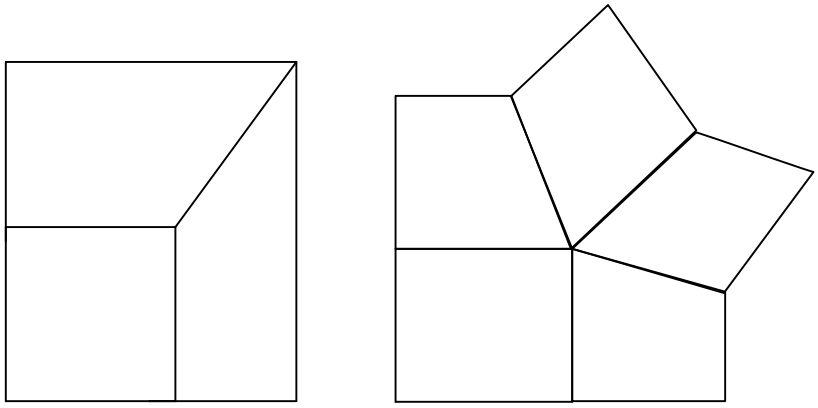


Figure 12: Multiblock meshes may have blocks that meet at singularity points (or lines in 3D), where nodes are surrounded by more or fewer cells than in a normal rectangular mesh.

15.1 Key concepts

The blocks of a multiblock mesh are each assigned an integer ID, ordered from 0 to N-1, with N being the number of blocks. Blocks that touch each other in any way are considered neighbors, and data can be communicated between neighbors across block boundaries. Each block has its own index space and its own alignment of its coordinate axes, while the Transformation class can be used to map between the index spaces of neighboring blocks.

Singularity points or lines must be specifically identified in the input that describes the multiblock mesh.

In a multiblock hierarchy, a PatchLevel can consist of Patches that lie on one or more blocks. The coarsest level will cover the entire multiblock domain, while finer levels may have patches on any number of blocks. The interior of each Patch must be fully inside one block, but Patches that touch block boundaries may have ghost regions that overlap the index space of another block.

15.2 SAMRAI classes used in multiblock applications

The same SAMRAI classes that are used to create a single-block application contain the internal capabilities to also work on multiblock applications. So there is little to no difference in the usage of classes such as PatchHierarchy, RefineSchedule, GriddingAlgorithm, or most of the other classes detailed earlier in this document. In this section we consider some classes and concepts that a user will likely encounter specifically when working with a multiblock mesh.

BaseGridGeometry and GridGeometry – In addition to the grid geometry functionality described earlier BaseGridGeometry holds all of the metadata that describes the block structure of the multiblock mesh, and provides interfaces for users to retrieve specific information about that structure. As BaseGridGeometry is an abstract base class, geom::GridGeometry is a concrete implementation available to be used as a grid geometry for a multiblock application. geom::GridGeometry is a general implementation of a grid geometry that manages an index space without being tied to a specific type of physical coordinate coordinate system. For a specific problem, the user provides the description of the coarse-level multiblock mesh in the grid geometry input.

BaseGridGeometry::Neighbor – A nested struct defined within BaseGridGeometry, Neighbor holds the data describing the relationship between two neighboring blocks.

BlockId – A class used to identify the block number for a block within the multiblock mesh. This class exists for type safety instead of a native integer type. BlockId can be used to identify a block as a whole, and it can also be used to associate an object with the block where it is located. For example, each Box holds a BlockId, which indicates that it is located on a specific block, and that its indices are defined in terms of the index space of that block.

Transformation – The Transformation class can be used for mapping between the index spaces of neighboring blocks. It is mainly used to operate on a Box by transforming a Box from one block's index space to another's. When transformed, a Box represents the same location on the mesh as before, but its BlockId is changed and its high and low indices are defined in terms of the index space associated with the new BlockId.

SingularityPatchStrategy – Usually applications need to do something problem-specific to handle the filling of ghost data around multiblock singularities. SingularityPatchStrategy contains a pure virtual interface called fillSingularityBoundaryConditions that will be called from RefineSchedule during the filling of data, where the user code can implement whatever needs to be done at those ghost regions. It is recommended that the same class that inherits from RefinePatchStrategy also inherit from SingularityPatchStrategy.

15.3 Defining a multiblock mesh

The multiblock mesh for a particular problem must be defined via the input for GridGeometry. The required input consists of 4 components: The number of blocks, boxes describing the index space of each block, descriptions of the neighbor relationship between all pairs of neighboring blocks, and the description of any singularity points or lines. A document describing the needed input in greater detail is provided at `docs/userdocs/Multiblock.pdf` in the SAMRAI distribution.

16 Boundary boxes and CoarseFineBoundary

SAMRAI can compute level boundaries and provides facilities for an application to access this data.

16.1 Key concepts

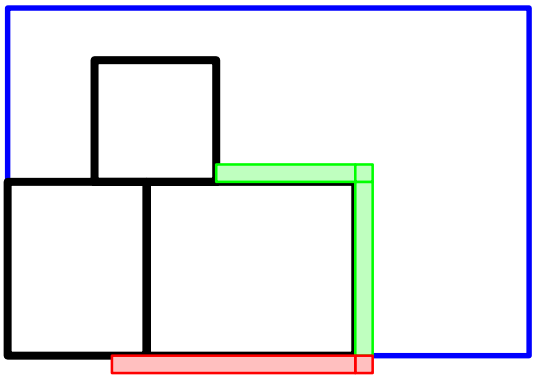
Applications often require the boundaries of a PatchLevel to perform special operations such as setting physical boundary conditions. SAMRAI represents level and patch boundaries with thin (one-cell wide) boxes lying just outside the boundary. Physical boundaries are boundaries that coincide with physical (not periodic) domain boundaries. They are automatically built for each PatchLevel. Coarse-fine boundaries can be constructed as needed.

16.2 BoundaryBox

A BoundaryBox describes a portion of a boundary next to a patch and is associated with its patch. Each box describes the extent, type (face, edge, node, etc.) and where it is in relation to its patch.

Each BoundaryBox has a Box to describe the extent of the portion of the boundary, an integer type (face, edge, node, etc.) and an integer location id to indicate where it is in relation to the patch. In directions parallel to the boundary, the Box extends the length of its portion of the boundary. In directions perpendicular to the boundary, the Box is one-cell wide.

The next figure illustrates some BoundaryBoxes for a patch in a 2D PatchLevel. The blue box represents the level 0 and also the domain. Black boxes represent level 1 patches. BoundaryBoxes for the big patch on level 1 are shown shaded. Red boxes represent physical boundaries. Green boxes are coarse-fine boundaries (explained below).



In 2D, there are 2 types of boundary boxes. Edge boundaries lie against an edge of a PatchLevel. These are the long boundary boxes in the figure. Node boundaries lie at convex corners of a PatchLevel. These are the single-cell boxes in the figure. The integer boundary type is the codimension of the boundary box. The figure shows one edge boundary box extending past the patch's corner. The reason there is no node boundary here is that it is not a convex corner of the PatchLevel. (The presence of a patch boundary does not change the type of the level boundary.) The extension past the corner of the

patch is equal to the maximum ghost width of registered data.

The location index of a boundary box indicates where the boundary box is in relation to its patch, i.e., the upper or lower side of the patch in a given index direction. See BoundaryBox::getLocationIndex() for details.

In 3D, there is one more BoundaryBox type: the face boundary type. Face type BoundaryBoxes lie against the 2D face of a 3D box.

16.3 How to access boundaries

SAMRAI computes physical boundaries for all Patches in a PatchLevel. The boundaries at a patch is available from a Patch's PatchGeometry, through the methods getNodeBoundaries(), getEdgeBoundaries(), getFaceBoundaries(), or the more general getCodimensionBoundaries().

The following simple example shows how one might set some quantity's ghost cell data to zero at physical edge boundaries:

```
void MyPatchStrategy::setPhysicalBoundaryConditions(
   hier::Patch& patch,
   const double fill_time,
   const hier::IntVector& ghost_width_to_fill)
{
   boost::shared_ptr<pdat::CellData<double> > uval_data(
      patch.getPatchData(d_uval_data_id, getDataContext()),
      BOOST_CAST_TAG);
   const boost::shared_ptr<geom::CartesianPatchGeometry> pgeom(
      patch.getPatchGeometry(),
      BOOST_CAST_TAG);
   hier::IntVector ghost_cells(uval_data->getGhostCellWidth());

   // Set boundary conditions for cells corresponding to patch edges.
   const std::vector<hier::BoundaryBox>& edge_bdry =
      pgeom->getCodimensionBoundaries(Bdry::EDGE2D);
   for (int i = 0; i < static_cast<int>(edge_bdry.size()); i++) {
      // Boundary box.
      const hier::BoundaryBox &bbox = edge_bdry[i];
      // Utility for working with boundary boxes.
      hier::BoundaryBoxUtil bbox_util(bbox);
      // Compute box corresponding to bbox, but with correct width.
      hier::Box working_box(bbox.getDim());
      bbox_util.stretchBoxToGhostWidth(working_box,
                                       ghost_width_to_fill);
      // Fill data in the working box.
      mesh_data->fill( 0.0, working_box );
   }
}
```

16.4 Coarse-fine boundaries

Coarse-fine boundaries refer to the boundaries between a refined region and an un-refined region, as shown by the green boxes in the above figure. Knowing these boundaries can simplify special operations around them. Like physical boundaries, they are also represented by BoundaryBoxes. They are not precomputed or held by the patches. The patches do not have inherent knowledge to compute them. They can be computed by constructing a CoarseFineBoundary object. These boundaries are accessed from this object using accessors of the same name.

17 SAMRAI database interfaces

All data read or generated by SAMRAI conforms to the class tbox::Database. This data specifically includes input parameters defining a problem, restart dumps, and plot dumps. Such data is generically called a SAMRAI database. Class tbox::Database defines the interface to SAMRAI databases. In particular it defines the types of data that may be read from or placed into a database. This data is quite generic and application independent. The following types of data may be read from or placed into a SAMRAI database:

- Databases
- Single bools, std::vectors of bools, and C-style arrays of bools.
- Single DatabaseBoxes, std::vectors of DatabaseBoxes, and C-style arrays of DatabaseBoxes.
- Single chars, std::vectors of chars, and C-style arrays of chars.
- Single complex values, std::vectors of complex values, and C-style arrays of complex values.
- Single double values, std::vectors of double values, and C-style arrays of double values.
- Single float values, std::vectors of float values, and C-style arrays of float values.
- Single integer values, std::vectors of integer values, and C-style arrays of integer values.
- Single std::strings, std::vectors of std::strings, and C-style arrays of std::strings.
- std::vectors of any object which is derived from class tbox::Serializable.

Each database entry is named. When an entry is created it is given a name unique to the database in which it resides. Retrieval of an entry is achieved through its name. Retrieval methods that take a default value are provided. These methods return the default value if the named entry does not exist. Retrieval methods not taking a default value will result in an unrecoverable error if the named entry does not exist.

17.1 Key concepts

Class tbox::Database only defines an interface. It describes WHAT a SAMRAI database may contain. It does not say anything about HOW the database is implemented. The interface also does not say anything about what the bools, ints, etc. in a database represent. That is entirely up to the entity reading from or creating the database. Similarly, the interface does not say anything about the organization of the database. That is also up to the entity reading or creating the database.

Another key point is that SAMRAI databases contain other SAMRAI databases. Therefore, SAMRAI databases are hierarchical. This provides a way to group related pieces of data and to separate unrelated pieces of data. A database embedded in another database is still a first class SAMRAI database as defined by the tbox::Database interface. One important implication of this hierarchical structure is that database entries with the same name may exist in separate yet embedded databases in the hierarchy.

17.2 Major classes and their associations

As stated above, class tbox::Database defines the interface to SAMRAI databases. It is an abstract base class. Applications are free to implement their own database classes by inheriting from tbox::Database and implementing its pure virtual methods. The SAMRAI library provides several database implementations that applications may choose to use. These are:

- MemoryDatabase—a class that is used for storing a database in memory. This can be used to create a database programmatically or to read an input file into memory. The typedef, InputDatabase, is a synonym.
- HDFDatabase—a class that implements SAMRAI databases as HDF5 files.
- SiloDatabase—a class that implements SAMRAI databases as Silo files.

Parallel to this class hierarchy is the DatabaseFactory hierarchy. Class tbox::DatabaseFactory is an abstract base class containing only one pure virtual method, allocate. It is used by tbox::RestartManager to construct the restart database. Applications may register the DatabaseFactory of their choice with the RestartManager in order to specify the type of restart database that they generate. An application that implements its own Database class should also implement a corresponding DatabaseFactory class. SAMRAI provides MemoryDatabaseFactory, HDFDatabaseFactory, and SiloDatabaseFactory.

The last class related to SAMRAI databases is tbox::DatabaseBox. It exists for two purposes. First, it essentially creates a POD (plain old data) representation of class hier::Box. This is necessary for certain database implementations such as HDFDatabase. Second, it breaks the dependency between the tbox::Database and hier::Box classes.

17.3 Basic usage examples

The following is an example of how to read an input database:

```
// Construct an input (memory) database and read the input file into it.
boost::shared_ptr<tbox::InputDatabase> input_db(
   new tbox::InputDatabase("input_db"));
tbox::InputManager::getManager()->parseInputFile(input_filename, input_db);

// Pass the database containing the TimerManager's inputs to the TimerManager
// constructor.
tbox::TimerManager::createManager(input_db->getDatabase("TimerManager"));

// Get the database containing information read by the main program and read
// parameters from it.
boost::shared_ptr<tbox::Database> main_db(input_db->getDatabase("Main"));
const tbox::Dimension dim(
   static_cast<unsigned_short>(main_db->getInteger("dim")));
string base_name = "unnamed";
base_name = main_db->getStringWithDefault("base_name", base_name);
const string log_file_name = base_name + ".log";
```

```
bool log_all_nodes = false;
log_all_nodes = main_db->getBoolWithDefault("log_all_nodes", log_all_nodes);
if (log_all_nodes) {
   tbox::PIO::logAllNodes(log_file_name);
}
else {
   tbox::PIO::logOnlyNodeZero(log_file_name);
}
…

// Construct various classes used by the application by passing the database
// containing each class' input parameters to its constructor.
boost::shared_ptr<geom::CartesianGridGeometry> grid_geom(
   new geom::CartesianGridGeometry(
      dim,
      "CartesianGeometry",
      input_db->getDatabase("CartesianGeometry")));
…
```

The following is an example of how a class would put its state into a restart database via the putToRestart method. Note that this code is independent of the specific implementation of restart_db.

```
void Euler::putToRestart(
   const boost::shared_ptr<tbox::Database>& restart_db) const
{
   restart_db->putDouble("d_gamma", d_gamma);
   restart_db->putString("d_riemann_solve", d_riemann_solve);
   restart_db->putInteger("d_godunov_order", d_godunov_order);
   restart_db->putString("d_corner_transport", d_corner_transport);
   restart_db->putIntegerArray("d_nghosts", &d_nghosts[0], d_dim.getValue());
   restart_db->putIntegerArray("d_fluxghosts",
      &d_fluxghosts[0],
      d_dim.getValue());
…
}
```

The following is an example of how a class would read its input parameters from a restart database via the getFromRestart method. Note that this code must agree completely with the corresponding putToRestart().

```
void Euler::getFromRestart()
{
   boost::shared_ptr<tbox::Database> root_db(
      tbox::RestartManager::getManager()->getRootDatabase());

   if (!root_db->isDatabase(d_object_name)) {
      TBOX_ERROR("Restart database corresponding to "
         << d_object_name << " not found in restart file." << endl);
   }
```

```
   boost::shared_ptr<tbox::Database> db(root_db->getDatabase(d_object_name));

   d_gamma = db->getDouble("d_gamma");
   d_riemann_solve = db->getString("d_riemann_solve");
   d_godunov_order = db->getInteger("d_godunov_order");
   d_corner_transport = db->getString("d_corner_transport");
   db->getIntegerArray("d_nghosts", &d_nghosts[0], d_dim.getValue());
   db->getIntegerArray("d_fluxghosts",
      &d_fluxghosts[0],
      d_dim.getValue());
…
}
```

18 Input and restart

Problems may be run either from their beginning or from some time step resulting from an earlier run. In either case it is necessary for SAMRAI to obtain information about the problem. When run from the beginning, the problem is entirely defined by an input database that must be supplied to SAMRAI. When run from a time step of an earlier run it is necessary for the earlier run to have produced a restart database defining the state of the problem. This restart database must then be supplied to SAMRAI along with an input database.

Each of SAMRAI's classes defines the parameters that it needs. Some of these parameters are required to be supplied. For others, defaults exist so these parameters need not be supplied. When a problem is initially run, the input database must define all required parameters for the SAMRAI classes relevant to the problem. The input database must also define any parameters with defaults that the user wishes to override. When run from a previously generated time step it is expected that the restart database will hold these parameters.

SAMRAI's classes all describe their parameters in their header file doxygen comments. This information may be either read directly from the header files or, more conveniently, by browsing the SAMRAI documentation of the classes of interest.

18.1 Key concepts

Any SAMRAI classes that have input parameters read their input and restart data when they are constructed. These classes expect the restart database passed into their constructors to the database for that class, not an enclosing database. The general pattern used by SAMRAI's classes to read input and restart data is to first check if the problem is being run from a restart database. If it is, then the class reads its parameters from the restart database. Required parameters not existing in the restart database will trigger an unrecoverable error. An input database is always required and parameters are read from it only after the restart database is read (if it exists). This allows a user to modify parameters on restart. Any given SAMRAI class will read parameters from an input database on restart only if the "read_on_restart" parameter is set to true for that class. If the problem is being run from the beginning then all required parameters must exist in the input database or an unrecoverable error will occur.

There are limitations on which restart parameters may be modified by an input database on restart. For example, it is perfectly acceptable to change any parameter related to the GriddingAlgoroithm. However, it is not allowed to change any parameter related to the GridGeometry.

There are two important classes related to input and restart—the InputManager and the RestartManager. The InputManager parses the input database and constructs an in memory version of it for use throughout an application. The RestartManager is the means by which restart databases are not only accessed but also produced. The more important of these is the simple interface that it provides to produce a restart database.

Each SAMRAI or application class that needs to save its state to a restart database must be derived from class tbox::Serializable and implement that base class' pure virtual putToRestart method. Each instance of these classes must register themselves with the RestartManager at the time of their construction. This allows the RestartManager to produce a restart database by simply invoking each registered object's putToRestart. The application simply invokes the RestartManager's writeRestartFile() method and all required restart data is saved to the restart database.

18.2 Major classes and their associations

The major classes relating to input and restart have already been mentioned. They will be summarized here for convenience:

- tbox::Serializable—the base class for any SAMRAI or application class which needs to save its state to a restart database. It contains one pure virtual method, putToRestart, which each derived class must implement.
- tbox::InputManager—a singleton class that provides convenience methods for reading input databases. This class hides the details of opening the input database, constructing the parser, and populating the in memory database with the input database's values.

tbox::RestartManager—a singleton class that provides convenience methods for reading and constructing restart databases.

18.3 Basic usage examples

In order to create an input database from an input file:

```
boost::shared_ptr<tbox::InputDatabase> input_db(
   new tbox::InputDatabase("input_db"));
tbox::InputManager::getManager()->parseInputFile(input_filename, input_db);
```

In order to open and read from a restart database:

```
// Open restart database.
tbox::RestartManager* restart_manager = tbox::RestartManager::getManager();
if (is_from_restart) {
   restart_manager->openRestartFile(
      restart_read_dirname,
      restore_num,
      mpi.getSize());
}
// Now that the restart database has been opened classes may access it via
// the tbox::RestartManager::getRootDatabase() method.
…
// Close the restart database.
restart_manager->closeRestartFile();
```

In order to generate a restart database at the current time step:

tbox::RestartManager::getManager()->writeRestartFile(restart_write_dirname, iteration_num);

18.4 User-defined input and restart

User defined classes must implement the means to read input parameters from both input and restart databases at the time of their construction. Users are free to choose how to do this. However it is done, the code reading input parameters from input or restart must know which parameters are required to exist in the database and which are optional. For optional parameters the code must know what the default values are. If the class will read from restart then the code must implement a policy covering which database parameter, input or restart, takes precedence.

Although you are free to choose your method of implementation, the pattern used by SAMRAI classes is as follows for your reference:

- Each SAMRAI class reading input parameters from an input database has a method, getFromInput(const boost::shared_ptr<tbox::Database>& input_db). This method knows which parameters are required to exist, which are optional, and any applicable defaults.
- If the class is also capable of reading from a restart database then getFromInput takes an argument, bool is_from_restart, so that the method may take the appropriate action should the user attempt to change a parameter read from the restart database that may not be modified.
- If the class is capable of reading from a restart database then it has the method, getFromRestart().
- The two methods, getFromRestart and getFromInput are called from each constructor in the following manner:

```
bool is_from_restart =
   tbox::RestartManager::getManager()->isFromRestart();
if (is_from_restart) {
   getFromRestart();
}
getFromInput(input_db, is_from_restart);
```

In order for a class to write its input parameters to an restart database it must be derived from class tbox::Serializable and implement that class' pure virtual method putToRestart(). In each of the class' constructors there must be a call to register the class with the RestartManager, tbox::RestartManager::getManager()->registerRestartItem(object_name, this);. The putToRestart method will decide how to organize this class' input parameters in the restart database. In particular, it will construct a database to hold these parameters in the root database. It is essential that putToRestart be in complete agreement with the code that reads a restart database. In the case of SAMRAI classes this is the method getFromRestart. In particular, there must be agreement on the name of the database that will hold the parameters and the names and types of the parameters.

It is particularly easy to map hierarchical AMR class structures onto a SAMRAI database, which is inherently hierarchical. Multiple instances of a class may be registered with the RestartManager provided each instance has a different name. Classes may be written to a restart database either

through explicit registration with the RestartManager or via delegation. When using delegation, some root class is registered for restart. This root class' putToRestart method traverses its aggregated classes and invokes methods in these classes to write to the restart database. The aggregated classes are not registered with the RestartManager. This delegation may recur arbitrarily many times. It may be instructive to look at how PatchData delegates to Patch, Patch delegates to PatchLevel, PatchLevel delegates to PatchHierarchy, and PatchHierarchy is registered with the RestartManager. This demonstrates not only delegation but also how classes that are hierarchically related are mapped to the restart database.

19 Visualization

The library provides support for writing data that is compatible with the VisIt visualization tool (http://wci.llnl.gov/codes/visit). Use of this capability requires compiling with the HDF5 library.

19.1 Key concepts

The VisItDataWriter directly supports the writing of cell and node data of double, float, and integer types, for scalar, vector, or tensor quantities. It also supports the notion of "material" and "species" data in cells. Material data is associated with multiple materials in a zone. These materials are immiscible and occupy some fraction of the zone volume. VisIt has the capability to internally perform a material interface reconstruction procedure to show the boundaries of materials across multi-material zones using the material volume fraction information. Species data can be associated with any material, whether there is one material or multiple materials in a zone. For example, a material "air" may have species such as N_2 and O_2. Each material may have its own set of species. Species are considered fully mixed with other species in the same material.

Data that does not exist on the hierarchy directly as CellData or NodeData, but is derived from data on the hierarchy by some calculation, can also be written. Such calculated data is called "derived" data and involves use of the VisDerivedDataStrategy class.

In parallel, by default one file per processor will be written. However, it is possible to specify the number of procs writing to each file when constructing the VisItDataWriter.

Note that all real data, whether of type double or float, is written out as float data. Therefore some caution is advised regarding the limitations of double to float conversions that occur implicitly during the writing process.

For data types which are not node or cell based, it is possible to process the data to be cell or node centered so that it can be written out for VisIt.

While Cartesian grids are assumed by default, it is possible to write out non-Cartesian general structured grids using an extra step where node coordinates are registered to be written out along with the field data.

19.2 Major classes and their associations

VisItDataWriter is the main class that manages the data writing. This class can use implementations of the interfaces defined by the VisDerivedDataStrategy and the VisMaterialsDataStrategy classes to perform user-customizable operations associated with derived quantities and materials and species specifications.

The basic procedure for use of the classes is to create the VisItDataWriter object, register plot quantities using the registration calls registerPlotQuantity() and/or registerDerivedPlotQuantity(), and then perform the write by calling writePlotData().

The Euler application example demonstrates the basic usage pattern. First, in main.C, the VisItDataWriter object is created. Then in Euler.C, in the registerModelVariables() method, the plot quantities are registered. Two derived quantities, total energy and momentum, are also registered. Note that the Euler class derives from VisDerivedDataStrategy, so that it can implement the required method to compute the derived quantities, namely packDerivedDataIntoDoubleBuffer(), which takes a string argument to specify which variable is to be written upon callback. Finally, the method writePlotData() is invoked on the VisItDataWriter to write out the current state of the hierarchy. This occurs in the main simulation loop in main.C.

19.3 Writing multi-material and species data

The concepts of "multi-material" and "species" information for zonal quantities are supported. In a multi-material zone, the components are considered spatially distinct and confined to a specific fraction of the zone's volume. By contrast, species data implies the components are well mixed, and are specified by a given fraction of the zone's mass.

An example demonstrating the writing of multi-material data can be found in the source/tests/viz/VisMaterials directory. There are two formats for writing this data, the sparse format and the non-sparse format. This choice is made by either registering the variable names to registerMaterialNames() or via registerSparseMaterialNames(). You must choose one format or the other; they cannot be intermingled. Depending on which of these two methods is called, the interface appu::VisMaterialsDataStrategy must be implemented via either the packMaterialFractionsIntoDoubleBuffer() method or the packMaterialFractionsIntoSparseBuffers() method. The method packMaterialFractionsIntoDoubleBuffer() will be called for each material; packMaterialFractionsIntoSparseBuffers() will be called only once to handle all materials. Refer to the example for details on how to implement these methods to provide the required information.

If there are multiple species for a given material, use registerSpeciesNames() after registerMaterialNames(), and implement the method packSpeciesFractionsIntoDoubleBuffer().

19.4 Writing derived data

Sometimes it can be desirable to write data for visualization which is not directly associated with a SAMRAI PatchData component, but is derived from one or more PatchData components. A simple example of this might be the writing of momentum data in a simulation which stores densities and energies rather than momenta.

To write derived data, a user must do two things. First, subclass appu::VisDerivedDataStrategy in order to provide an implementation for packDerivedDataIntoDoubleBuffer(). This method provides an allocated buffer, into which the user must write his derived data. Then, this method is linked to a plot quantity through the VisItDataWriter method registerDerivedPlotQuantity().

An example of this can be found in source/test/applications/Euler.C, which contains both the registration call and the implementation of packDerivedDataIntoDoubleBuffer().

20 Further reading

Berger, M. J., and P. Colella, Local Adaptive Mesh Refinement for Shock Hydrodynamics. Journal of Computational Physics 82:64-84 (1989).

Hornung, R. D., and S. R. Kohn, Managing application complexity in the SAMRAI object-oriented framework. Concurrency and Computation: Practice and Experience 14:347-368 (2002).

SAMRAI Project Site. URL http://www.llnl.gov/CASC/SAMRAI/ Center for Applied Scientific Computing. Lawrence Livermore Laboratory, Livermore, CA.

Berger, M., and I. Rigoutsos, An Algorithm for Point Clustering and Grid Generation. IEEE Transactions on Systems, Man and Cybernetics 21:1278-1286 (1991).

Gunney, B. T. N., A. M. Wissink, and, D. A. Hysom Parallel Clustering Algorithms for Structured AMR. Journal of Parallel and Distributed Computing 66:1419-1430 (2006).