Communication Algorithms and Boundary Conditions in SAMRAI

Introduction

The purpose of this document is to provide a brief description of parallel AMR data communication support in the SAMRAI library. It provides an outline of the main software abstractions involved, where they can be found in the library, and how one may use them in the context of an AMR application. Questions and requests for additional information should be sent via email to `samrai@llnl.gov`.

Structured Adaptive Mesh Refinement

The goal of adaptive mesh refinement (AMR) is to focus computational resources in a numerical simulation by dynamically placing a finer spatial and/or temporal mesh near solution features requiring higher resolution. Structured adaptive mesh refinement (SAMR) is a particular AMR methodology in which the computational mesh is implemented as a collection of structured mesh components. The mesh consists of a hierarchy of levels, each of which corresponds to a single uniform degree of mesh spacing. See Figure 1. Also, the levels are nested; that is, the coarsest level covers the entire computational domain and each successively finer level covers a portion of the interior of the next coarser level. Computational cells on each level are clustered to form a set of logically rectangular regions, or *patches*.

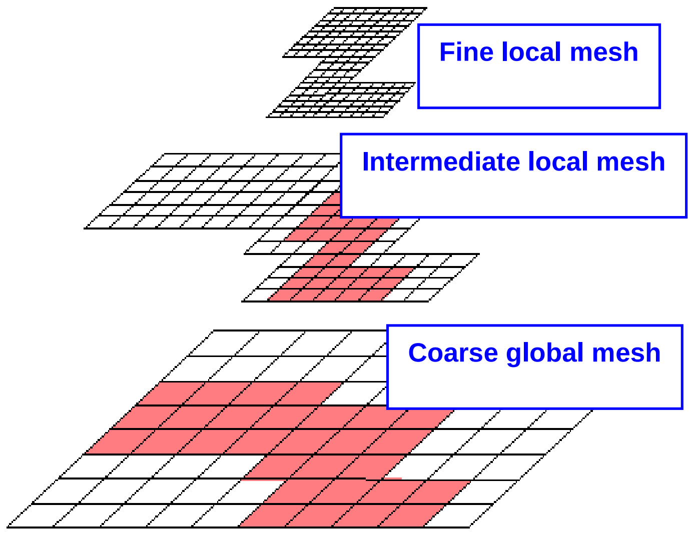

Figure 1. Simple SAMR hierarchy with two levels of local refinement. On a level, cells are clustered to form logically rectangular "patches".

SAMR solution methods share certain characteristics with parallel uniform structured grid methods. In either case, simulation data are represented by arrays (typically) that represent the numerical solution on logically rectangular regions. A computation is usually comprised of a set of numerical routines that operate on those data regions and communication operations that pass information between regions; for example, to fill *ghost cells*. Additional considerations arise when a mesh is refined locally so that SAMR solution methods can be substantially more complicated than those for uniform meshes. In particular, the solution must be constructed on multiple levels of mesh resolution, where the domain of each level is described by a collection of rectangular regions whose union may be complicated and which may change often. Correct solution algorithms must treat internal mesh boundaries between coarse and fine levels properly to maintain a consistent solution state.

A general observation about SAMR applications is that most parallelism occurs on individual refinement levels. Patches are distributed to different processors and once boundary data is exchanged among them, patches are essentially uncoupled for parallel processing. Thus, the basic ingredients for the vast majority of SAMR applications, beyond a proper numerical problem formulation for an SAMR hierarchy, are numerical routines for patches and inter-patch data transfer operations. Essentially, communication operations involve data copying, data refining, and data coarsening. For example, data may be copied from patch interiors to neighboring patch ghost regions on a single mesh level to uncouple the patches so that they may be processed in parallel. Also, data may be spatially interpolated from coarser levels into ghost regions around a finer level to supply boundary data before integrating that fine level. Lastly, solution values may be coarsened from a fine level into the interior of a coarser level so that the coarser level sees the more accurate solution. This document describes how to perform inter-patch data transfer operations like these using SAMRAI.

SAMRAI Parallel Communication Classes

Before we delve into examples that demonstrate how one moves data around on an AMR patch hierarchy in SAMRAI, we present some basic concepts and terminology that appear in the following discussion. The framework in SAMRAI for transferring data between patches in an SAMR patch hierarchy revolves around three basic abstractions: *a communication algorithm*, a *communication schedule*, and a *patch strategy*. Each of these abstractions appears in one of two forms; they are distinguished by they way in which they are used in hierarchy data communication operations. These and other related classes reside in the *Transfer* package. There is a *coarsen* algorithm, a *coarsen* schedule, and a *coarsen* patch strategy. These classes manage data coarsening from a finer mesh level to some coarser level. There is also a *refine* algorithm, a *refine* schedule, and a *refine* patch strategy. These classes manage data transfers among patches on the same hierarchy level (a special case of refinement) and data refinement from coarser meshes to some finer level.

A **communication algorithm** supports an algorithm-level description of a data transfer phase in a computation. For example, an algorithm may represent the filling of ghost cell data for a collection of variables before integrating the numerical solution on a mesh level. An algorithm depends on variables and operators that interpolate those quantities in space and time on an SAMR mesh. An algorithm may be used to manage data movement for any number of variables on any subset of levels in an AMR patch hierarchy. The state of a communication algorithm object is defined *after it is created* through a set of *registration* operations. Each registration specifies source and destination patch data information and associated spatial refinement and temporal interpolation operators needed to move the data. It is important to note that each algorithm object is independent of the SAMR patch hierarchy configuration. Usually, the state of an algorithm is established once (via the registration process) during a problem setup phase and is maintained throughout an entire computation.

A **communication schedule** manages the set of data transfer operations, specific to a particular configuration of an SAMR patch hierarchy, required to perform the data communication described by an algorithm. Thus, a schedule depends on an algorithm and a particular layout of patches and levels in a hierarchy. An algorithm is used to build a different schedule for each instantiation of the communication pattern that it describes. For example, a single algorithm may be used to interpolate a set of variable quantities between all consecutive pairs of mesh levels in the hierarchy. But, a separate schedule is needed to move the data between each pair of levels. When a schedule is constructed, descriptions of data send and receive sets are computed and stored in the schedule. As a result, a schedule may be invoked to communicate data any number of times on the same patch hierarchy configuration. However, when the patch configuration changes (e.g., via mesh adaptation), the schedule is no longer valid and must be regenerated.

A **patch strategy** is an abstract class defining an interface for user-defined routines invoked during data communication procedures. Specifically, the interface is used to provide data refining or coarsening operations that are problem-specific and to supply boundary conditions for the physical domain.

Generality, flexibility, and efficiency motivate the decomposition of the SAMRAI communication framework into the three abstractions described above. Separating the algorithm from a schedule allows one to describe a variety of AMR communication scenarios involving the same set of variables independent of the patch hierarchy configuration. That is, one may invoke the communication operations in different ways on the same hierarchy. Separating the schedule creation from the actual data communication allows one to amortize the cost of creating send/receive data dependency information over multiple communication cycles. Finally, the patch strategy provides a fairly straightforward mechanism for users to customize communication operations while relieving them of complex implementation details associated with data movement on an AMR mesh.

SAMR Hierarchy and Data Objects in SAMRAI

Before we provide examples of SAMRAI data communication operations, we recall some SAMRAI objects that are central to our discussion. The `Box`, `PatchHierarchy`, `PatchLevel`, and `Patch` classes are found in the *Hierarchy* package. A `Box` object represents a logically rectangular region of index space. Essentially all operations in an SAMR application depend on index space concepts and box calculus operations. A `PatchHierarchy` object maintains the collection of patch levels in an SAMR patch hierarchy. A `PatchLevel` object is used to manage the patches comprising a single level of mesh resolution in the hierarchy. Finally, a `Patch` is a container holding all simulation data residing over a single mesh region defined by a `Box`. See Figure 2.

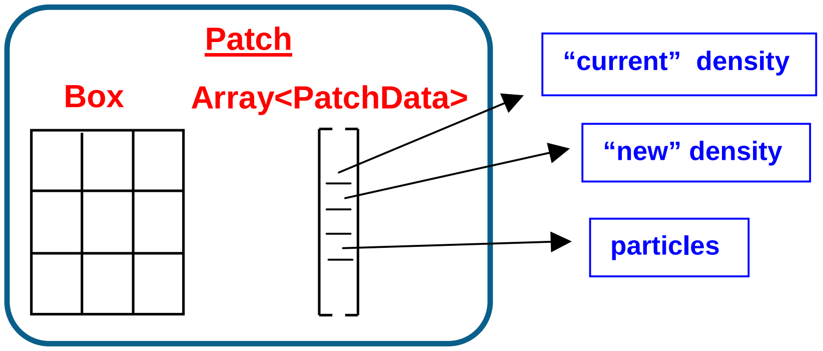

Figure 2. SAMRAI "patches" are containers providing access to all data created over a box region. Data objects, such as old and new density and particles are stored in an array on each patch.

Each patch stores an array of data objects. These objects may be accessed via array indices. However, the preferred method is to use `Variable` and `VariableContext` objects. That is, SAMRAI provides mechanisms that make managing patch data quantities on an SAMR hierarchy straightforward, especially for applications that use many variables. Figure 2 shows two storage locations on a patch for the variable "density". One can specify this either using two different *variables*, "current density" and "new density", or using a single variable, "density", and two different *contexts*, "current" and new". The `VariableDatabase` class in the *Hierarchy* package manages the mapping between variables and contexts and indices in the data array on each patch in the hierarchy. The database is a singleton object that is conveniently accessible from anywhere in the application source code. The variable, variable context, variable database concepts may seem idiosyncratic at first inspection, but once understood, it should be clear that SAMRAI allows great flexibility for managing variable quantities in a complicated application code.

SAMRAI Communication Examples

In the next several sections, we describe various ways in which to use communication algorithms and schedules in SAMRAI. For completeness and consistency, we begin with the process of creating variable and variable context objects, and registering them with the variable database.

Suppose that we are building an application in which we have two variable quantities, cell-centered density and particles. These variables will appear in the examples that follow. We first create density and particle variables and the variable contexts we will use:

```
VariableDatabase* vdb = VariableDatabase::getDatabase();

Pointer< CellVariable<double> > density_var =
   new CellVariable<double>("density", 1);
Pointer<ParticleVariable> particle_var = new ParticleVariable(…);

Pointer<VariableContext> cur_ctxt = vdb->getContext("CURRENT");
Pointer<VariableContext> new_ctxt = vdb->getContext("NEW");
```

The pointer “vdb” to the singleton variable database object is used to manage the mapping between the variables and the associated patch data storage. We have created two variables. Density is a cell-centered scalar (signified by the “1” in the constructor argument list) of type double. The particle variable is an application-specific particle object, left unspecified. We assume that we have available some particle data type that we can link with SAMRAI. Construction of user-defined data types will not be discussed in this document. We include this notion here solely to illustrate that, once such a user-defined type is built, it may be used with SAMRAI communication routines just as any standard array-based type found in the library. Finally, we create two variable contexts for managing “current” and “new” data.

Next, we establish the variable storage indices on each patch by registering variable, context, and ghost cell width information with the variable database:

```
int cur_density_data_id = vdb->registerVariableAndContext(
   density_var, cur_ctxt, IntVector(0));
int new_density_id = vdb-> registerVariableAndContext(
   density_var, new_ctxt, IntVector(1));

int particle_data_id = vdb->registerVariableAndContext(
   particle_var, cur_ctxt, IntVector(1));
```

The integer value returned by each registration call is the index of the data array location for that data object on each patch in the hierarchy. In this example, there will be one copy of particle data on each patch associated with the “current” context and it will have a ghost cell layer one cell wide. There will be two copies of density data on each patch, one for “current” values and one for “new” values with ghost cell widths of zero cells and one cell, respectively. We emphasize that *variable objects are distinct from patch data objects* that store data values. The reason for this is a variable tends to persist, representing the same quantity for the duration of a computation. Patch data objects are created and destroyed as the adaptive mesh moves and also to accommodate the storage needs of our computation. Also, variables know nothing about ghost cell width information; a single variable may be associated with multiple storage locations on a patch, each of which has a different number of ghost cells. An important function of the variable database is to maintain consistency of this information. For example, it is impossible to register a variable-context pair with the database more than once with different ghost cell information.

Data Refinement Operations

In this section, we describe SAMRAI communication operations for data refinement using the quantities defined above. Typical usage of refine algorithms and schedules involves four steps:

1. Create a `RefineAlgorithm` object.
2. Register operations with the refine algorithm to provide source and destination patch data information, as well as space refinement and time interpolation operators. The `RefineAlgorithm` class has two registration methods; one supports time interpolation, one does not.
3. After all operations are registered, create a `RefineSchedule` object using the refine algorithm. There are several methods in the `RefineAlgorithm` class for this purpose distinguished by the resulting data communication pattern. A concrete `RefinePatchStrategy` object may be required when the schedule is created to supply physical boundary conditions and user-defined spatial data refine operations.
4. Invoke the `RefineSchedule` to perform the data transfers.

The following sequence of operations implements the first two steps of this prescription:

```
RefineAlgorithm fill_ghost_data;

Pointer<RefineOperator> density_refine_op =
   new CartesianCellDoubleLinearRefine();

fill_ghost_data.registerRefine(new_density_id,  // destination
                               cur_density_id,  // source
                               new_density_id,  // scratch
                               density_refine_op);

fill_ghost_data.registerRefine(particle_data_id,  // destination
                               particle_data_id,  // source
                               particle_data_id,  // scratch
                               NULL);
```

Note that the registration function `registerRefine()` accepts source and destination patch data index and operator information. The first three arguments in the calls above identify destination data, source data on the destination level (and potentially coarser levels), and “scratch” data, respectively. Scratch data is used internally within SAMRAI to perform copy and refinement operations. It must have a sufficient ghost cell width to support the stencil of the refine operator specified. Generally, destination and scratch data may be the same; user data values *will not* be overwritten. The “null” refine operator specified for the particles indicates that particle data will not be refined, but only copied into ghost regions from neighboring patches.

The refine algorithm "fill_ghost data" may be used to fill density and particle ghost cell data on any level in an SAMR patch hierarchy, and in more than one way depending on how the algorithm is used to produce a refine schedule object. Suppose that we have the simple patch

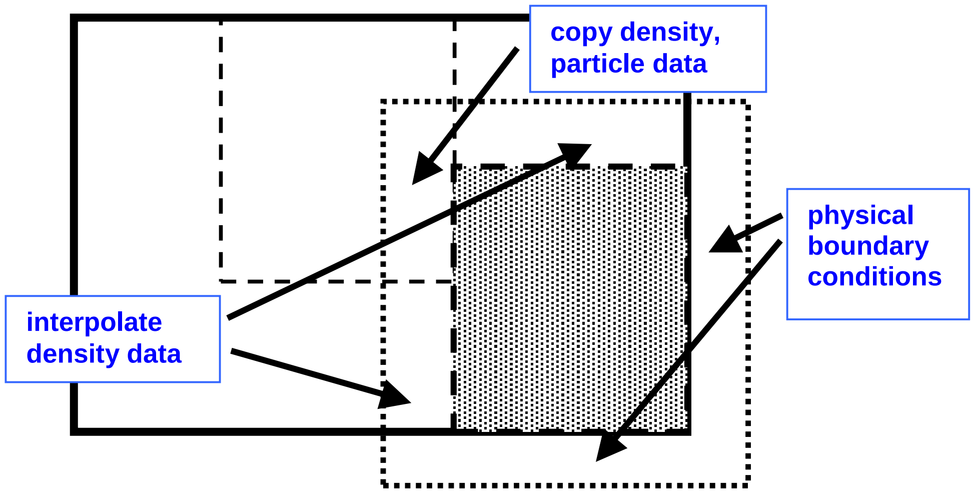

Figure 3. This simple SAMR patch hierarchy configuration illustrates potential sources of ghost cell data when the refine algorithm "fill_ghost_data" is used to communicate data. The hierarchy has two levels; fine patches are outlined with dashes. Part of the ghost region around the shaded fine patch will receive density and particle data from the adjacent fine patch interior. Other ghost regions in the domain interior, but not intersecting other fine patches, will receive refined values from the coarse level. Regions outside the physical domain will get boundary values set by user-defined routines.

hierarchy configuration illustrated in Figure 3. We describe two communication patterns that may be generated using the "fill_ghost_data" algorithm.

First, we fill ghost regions on the fine level patches from sources identified in Figure 3. Note that on fine patch interior regions "current" density values will be copied to "new" data as prescribed by the source and destination information given. Following the last two steps of the prescription above we accomplish this:

```
Pointer<PatchHierarchy> hierarchy = . . .;
RefinePatchStrategy* my_patch_strategy = . . .;

double fill_time = . . .;

Pointer<PatchLevel> fine_level = hierarchy->getPatchLevel(1);

Pointer<RefineSchedule> fill_ghost_schedule =
   fill_ghost_data.createSchedule(fine_level,
                                  0,   // next coarser level index
                                  hierarchy,
                                  my_patch_strategy);

fill_ghost_schedule->fillData(fill_time);
```

We assume that there is some `PatchHierarchy` object available and that we have implemented a concrete `RefinePatchStrategy` object to provide physical boundary routines. The `createSchedule()` function produces a `RefineSchedule` object that computes data transactions needed to perform the data movement. We identify the fine patch level (i.e., level "1") as the destination level and the hierarchy and next coarser level number (i.e., "0") as data sources for refine operations. The next coarser level number indicates the relationship between the hierarchy and the level to be filled. The patch strategy provides user-defined data refinement and boundary routines. If a “null” strategy is given, no such operations will be invoked. Finally, the `fillData()` function performs the data communication.

As a second example, we fill the fine level using only the fine level as a data source:

```
Pointer<RefineSchedule> fill_fine_ghost_schedule =
   fill_ghost_data.createSchedule(fine_level, fine_level, NULL);

fill_ghost_schedule->fillData(fill_time);
```

This communication may be needed, for example, to synchronize the fine patches without upsetting the ghost values set from coarser levels and physical boundary conditions. Notice that the same algorithm creates this schedule using a different `createSchedule()` method. By passing a “null” patch strategy, we short circuit calls to the physical boundary routines. If we need to reset those values, we would pass the pointer “my_patch_strategy”.

The two preceding examples perform data communication without time interpolation. As noted earlier, the `RefineAlgorithm` class has a `registerRefine()` function to specify operations involving time interpolation. To set density values in fine ghost regions based on time interpolation between current and new values on the coarse level, for example, we would perform the following sequence of operations:

```
RefineAlgorithm fill_time_ghost_data;

Pointer<TimeInterpolateOperator> density_time_op =
   new CellDoubleLinearTimeInterpolate();

fill_time_ghost_data.registerRefine(
                     new_density_id,  // destination
                     cur_density_id,  // source
                     cur_density_id,  // old on coarse
                     new_density_id,  // new on coarse
                     new_density_id,  // scratch
                     density_refine_op,
                     density_time_op);

fill_time_ghost_data.registerRefine(
                     particle_data_id,  // destination
                     particle_data_id,  // source
                     particle_data_id,  // scratch
                     NULL);

Pointer<RefineSchedule> fill_time_ghost_schedule =
   fill_ghost_data.createSchedule(hierarchy, 0,
                                  fine_level,
                                  my_patch_strategy);

fill_time_ghost_schedule->fillData(fill_time);
```

To avoid confusion, we use a different `RefineAlgorithm` object that the earlier examples. However, operations involving time refinement and operations that do not can both be registered with the same algorithm. Note that this sequence of operations resembles very closely those operations in the previous examples. The major difference lies in the specification of the refine operations for the density variable. Here, we specify the “old” and ”new” time source data on the coarse level in addition to the other sources. We also provide a time interpolation operator. Specification of particle communication is the same as before; there will be neither time nor space interpolation of the particle data.

Remarks:

- `RefineAlgorithm` and `RefineSchedule` objects are used to perform SAMR data communication operations involving spatial data refinement between levels and data copying between patches on the same level.
- The `RefineAlgorithm` class has two `registerRefine()` functions with which to register refine operations; one supports time refinement, the other does not.
- The `RefineAlgorithm` class has several `createSchedule()` functions with which to create a refine schedule. These methods are distinguished by the sort of communication pattern they produce; e.g., inter-patch communication on a single level, communication of data between two different levels at the same grid resolution, or refinement of data between different SAMR hierarchy levels. An important use of this routine that was not described above involves setting data on a new level after a remeshing operation. In this case, data is copied to the new level where it intersects the old level before the old level is destroyed.
- User-defined data refine operations and physical boundary routines are provided by creating a `RefinePatchStrategy` object and passing it to the `createSchedule()` function.

Data Coarsening Operations

SAMRAI communication operations for data coarsening on an SAMR hierarchy are similar to those for data refinement. Typical usage of coarsen algorithms and schedules involves four steps:

1. Create a `CoarsenAlgorithm` object.
2. Register operations with the coarsen algorithm to provide source and destination patch data information, as well as coarsening operators. Note that there is only one such registration function in the `CoarsenAlgorithm` class.
3. After all operations are registered, create a `CoarsenSchedule` object using the coarsen algorithm. There is only one method in the `CoarsenAlgorithm` class for this purpose. If user-defined data coarsen operators are needed, a concrete `CoarsenPatchStrategy` object is passed to the coaren schedule at this time.
4. Invoke the `CoarsenSchedule` to perform the data transfers.

To illustrate the data coarsening process, we will coarsen the "current" density values from fine patches to corresponding regions on coarse patch interiors. This is done as follows:

```
CoarsenAlgorithm coarsen_data;

Pointer<RefineOperator> density_coarsen_op =
   new CartesianCellDoubleWeightedAverage();

coarsen_data.registerCoarsen(cur_density_id,  // destination
                             cur_density_id,  // source
                             cur_density_id,  // scratch
                             density_coarsen_op);

double coarsen_time = . . .;

Pointer<PatchLevel> coarse_level = hierarchy->getPatchLevel(0);
Pointer<PatchLevel> fine_level = hierarchy->getPatchLevel(1);

Pointer<RefineSchedule> coarsen_schedule =
   coarsen_data.createSchedule(coarse_level,
                               fine_level,
                               my_patch_strategy);

coarsen_schedule->coarsenData(coarsen_time);
```

The major differences between this coarsening example and the preceding refinement example are that the `createSchedule()` function accepts only the source and destination level and that the patch strategy is passed to the schedule when the communication is invoked rather than when the schedule is created.

Remarks:

- `CoarsenAlgorithm` and `CoarsenSchedule` objects are used to perform SAMR data communication operations involving spatial data coarsening. Unlike, refine operations, SAMRAI coarsen operations never involve time interpolation nor boundary conditions.
- The `CoarsenAlgorithm` class has one `registerCoarsen()` function with which to register coarsen operations.
- The `CoarsenAlgorithm` class has one `createSchedule()` function with which to create a coarsen schedule (recall that `RefineAlgorithm` has several such functions). No more than two levels are ever involved in a single data coarsen phase since SAMR patch hierarchy levels are nested. Thus, coarsen schedule creation only requires a coarse (destination) and a fine (source) level.
- User-defined data coarsen operations are set by creating a `CoarsenPatchStrategy` object passing it to the `createSchedule()` function.

User-defined Data Refine and Coarsen Operators

In many cases, users do not need special spatial coarsen or refine operators for their applications. The needed operators can be found in the SAMRAI library in the *Geometry* package. If a new operation is needed, one has a choice for implementing it. An operator class may be derived from either the `CoarsenOperator` or `RefineOperator` interface found in SAMRAI *Transfer* package. This works whenever the operation is a function of a single variable and the spatial mesh coordinates. However, when an operation is a function of more than one variable, either the `CoarsenPatchStrategy` or the `RefinePatchStrategy` interface must be used. These interfaces allow one to provide very general routines to the communication schedules so that they may be invoked during data refinement or data coarsening communication operations.

As an example, suppose that two variables, density and velocity, are used in an application. To refine or coarsen density conservatively, all that is needed are the density values and the mesh coordinates. Thus, these operations may be found in the SAMRAI library or, if not, may be implemented as an operator object that the schedule can invoke. In contrast, refinement or coarsening of velocity in so as to conserve momentum requires one to operate on the product of density and velocity conservatively and then divide the result by the refined or coarsened density. This operation is not a function of a single variable and the SAMRAI library does not understand the physical relationship between density and velocity which is defined at the application level.

The `CoarsenPatchStrategy` and `RefinePatchStrategy` classes each declares two functions, `preprocessCoarsen/Refine()` and `postprocessCoaren/Refine(),` which provide interfaces to user-supplied coarsen and refine operations. When a patch strategy object is passed to a communication schedule, these operations will be invoked at every data coarsen/refine step along with the standard operator objects passed into the communication registration routines. In the schedule, the `preprocessCoarsen/Refine()` function that the user supplies is called first. Then, each of the standard operators is called. Lastly, the user `postprocessCoaren/Refine()` function is called. Using this scheme, an application developer may combine standard operations with special operations for his problem.

Setting Physical Boundary Conditions

Implementing physical boundary conditions is a fundamental task when using SAMRAI to build an SAMR application. This is done using the same concrete `RefinePatchStrategy` object that provides special spatial data refinement operations to the refine schedule. This strategy interface has a virtual function `setPhysicalBoundaryConditions()` for setting values at the boundary of the computational domain. In this section, we describe this function, how one uses `BoundaryBox` objects to determine the relationship between a patch and the domain boundary, and special issues that arise for periodic boundary conditions.

The signature of the function for setting boundary values is

```
void RefinePatchStrategy::setPhysicalBoundaryConditions(
    Patch& patch,
    const double fill_time,
    const IntVector& ghost_width_to_fill);
```

When a concrete refine patch strategy object is passed to the refine algorithm function `createSchedule()`, the routine for filling boundary conditions will be called whenever it is needed to properly perform all the data communication operations described by the schedule. Within this method, a user must fill all physical boundary values for the given patch at the given time in a ghost region whose width is indicated. The routine must be implemented so that it may be used on any arbitrary patch in the hierarchy. To do this properly, one must know which patch data to set and how to determine where the patch sits relative to the physical boundary. The emphasis of this section is on the second point. However, before we begin we note two things. First, the data for which a user will set boundary values is that which corresponds to the “scratch” data indices given when operations are registered with the refine algorithm. Second, the physical boundary routine will always be the last operation invoked when filling data on a patch; thus, all patch interior data and all other ghost region values are available when boundary values are set.

SAMR computations give rise to complicated patch configurations. Each SAMRAI `Patch` owns a `PatchGeometry` object that has information on where the patch resides in the domain. In addition to real space coordinates and other mesh information, the patch geometry object provides information about where a patch lives in terms of the underlying index space. When a SAMRAI `Patch` object is created, an array of `Boundary Box` objects is constructed for each spatial dimension and stored in the `PatchGeometry` object owned by the patch. The `BoundaryBox` objects in each array define the relationship between the patch and the domain boundary for a single intersection type (e.g., face, edge, node). Without delving too deeply into implementation details, each `BoundaryBox` has a `Box` and two integer data members. The `Box` describes a ghost region outside the physical domain and the integers identify the *boundary type* and *location index*. One does not need to fully understand the manner in which the box describes a boundary region to use a `BoundaryBox` object effectively. However, this may be needed in some circumstances, so we note that the box data member resides immediately outside of the boundary shared by the Patch and the physical domain and is one cell wide in the direction normal to the physical boundary.

The *boundary type* identifier indicates whether the patch boundary intersects the physical boundary on a 2-dimensional surface (face), a 1-dimensional line (edge), or at a single point (node). The following table describes the integer boundary types for each spatial dimension:

Boundary type	**1-d hierarchy**	**2-d hierarchy**	**3-d hierarchy**
node	1	2	3
edge	N/A	1	2
face	N/A	N/A	1

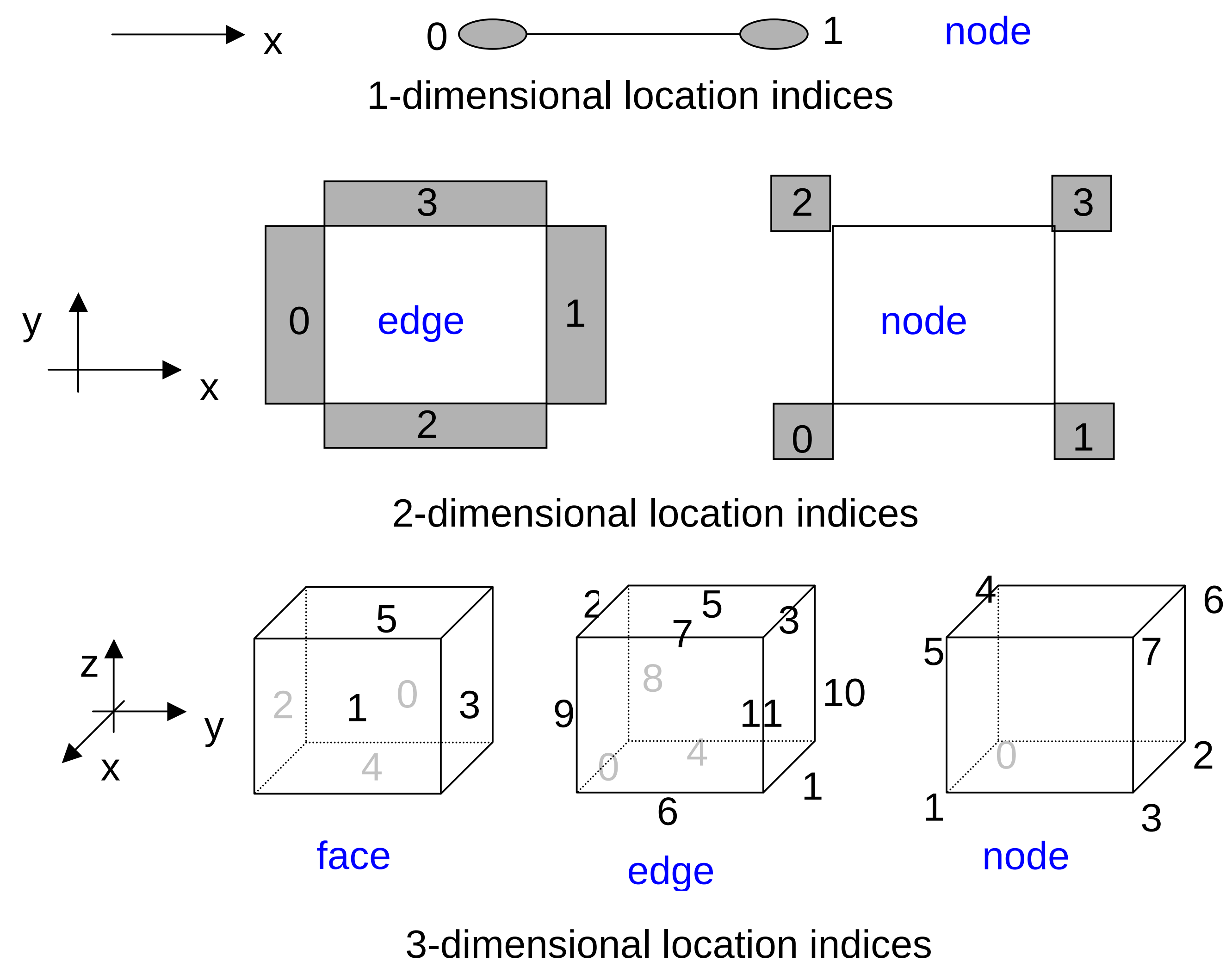
x
0
1
node
1-dimensional location indices
y
x
3
0
edge
1
2
2
3
node
0
1
2-dimensional location indices
z
y
x
5
2
1
0
3
4
face
2
5
7
3
8
9
11
10
0
4
1
6
edge
4
6
5
7
0
2
1
3
node
3-dimensional location indices

Figure 4. BoundaryBox location index schemes for 1-, 2-, and 3-dimensional boxes. This location index information indicates how a patch boundary intersects the physical domain boundary

The *location index* indicates where a patch sits relative to the boundary. For example, in a 2-dimensional rectangular domain, a patch may touch the boundary along an *edge* on the top, bottom, left, or right, or some combination of these. For each spatial dimension and boundary type, an indexing scheme for boundary regions helps one to resolve situations such as this. Figure 4 illustrates the indexing scheme for the location of each boundary type.

The essential task of the `setPhysicalBoudaryConditions()` routine is to set boundary values for a patch that are appropriate for the given problem. Recall that arrays of `BoundaryBox` objects owned by the patch geometry describe the relationship between the patch and the domain boundary. After retrieving the arrays, one should iterate through them and set values based on the description of a boundary region found in each `BoundaryBox` object. Often, the most straightforward way to set boundary data is to write general routines to fill boundary values in a box region given information about the relationship between that region and the patch (e.g., whether it lies above, below, to the left or right of the patch). Then, those routines can be used for any patch in any SAMR patch configuration. For example, one may implement a routine to fill patch boundary values in a 3-dimensional problem as follows:

```
void MyPatchStrategy::setPhysicalBoundaryConditions(
    Patch& patch,
    const double fill_time,
    const IntVector& ghost_width_to_fill)
{

   Pointer<PatchGeometry> pgeom = patch.getPatchGeometry();

   const Array<BoundaryBox> face_bdry = pgeom->getFaceBoundary();
   const Array<BoundaryBox> edge_bdry = pgeom->getEdgeBoundary();
   const Array<BoundaryBox> node_bdry = pgeom->getNodeBoundary();

   Box interior(patch.getBox());

   for (int i = 0; i < face_bdry->getSize(); i++) {

      int bdry_type = face_bdry[i].getBoundaryType();
      int bdry_loc = face_bdry[i].getLocationIndex();

      Box fill_box =
         pgeom->getBoundaryFillBox(face_bdry[i],
                                   interior,
                                   ghost_width_to_fill);

      setMyBoundaryRegion(. . .);

   }

   . . .
}
```

First, we obtain arrays for face, edge, and node boundary regions from the patch geometry. Then, we iterate through the face boundary regions, setting values for each. Note that the function `getBoundaryFillBox()` in the `PatchGeometry` class is used to get a box describing the boundary region to be filled for a given `BoundaryBox`, a box for the patch interior, and a fill ghost width. The routine `setMyBoundaryRegion()`is a user-defined function that fills regions with the appropriate boundary values. A user must dimension the data arrays and access the array entries properly for each patch and boundary value setting scenario. In an actual application code, we would most likely process the edge and node boundaries in a similar way.

Before we conclude, we comment on periodic boundary conditions. Essentially, a user needs only to write boundary routines for non-periodic boundary regions. `BoundaryBox` objects are created only for patches with boundaries that intersect non-periodic physical boundaries. All periodic boundary regions are filled automatically during execution of a communication schedule. They are treated as though they are part of the domain interior. However, the presence of periodic boundaries may influence how a user may process a non-periodic boundary region.

When BoundaryBox objects are created in a problem that has both periodic and non-periodic boundaries, periodic boundary regions are treated as though they are part of the domain interior. For example, a boundary region that may be described with an edge boundary and a node boundary when there are no periodic boundaries may instead be described using only an edge boundary. Figure 5 illustrates a situation of this sort in 2 dimensions. In 3-dimensional problems, similar situations may arise involving face and edge boundaries. User-supplied boundary routines should be aware of the periodic nature of the problem domain if this will influence the way that boundary values are set. In Figure 5, the boundary region in the upper-right corner should have the same values that appear in the boundary region near the upper-left corner. Note that when these values depend on values in the domain interior there is no problem; values in periodic boundary regions are already filled before the user boundary routine is called.

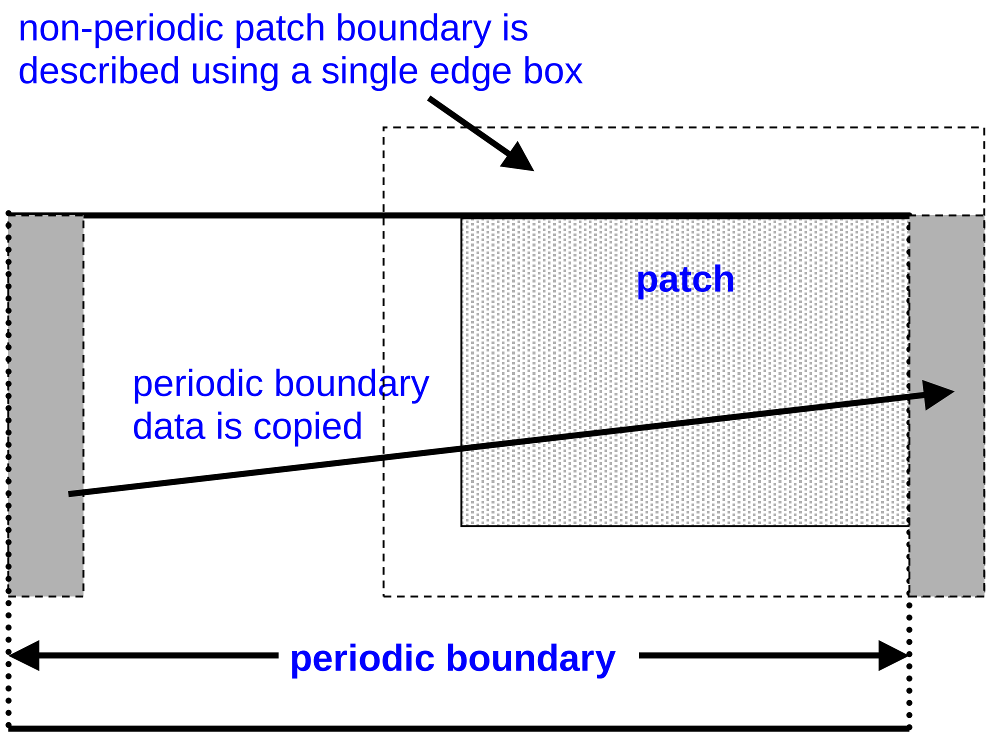
non-periodic patch boundary is
described using a single edge box
patch
periodic boundary
data is copied
periodic boundary

Figure 5. Example of a patch located along a periodic boundary in 2-dimensions. In this case, periodic boundary data is copied into the right boundary region from the left side of the domain. The user sees a single "edge" boundary box along the top of the patch to fill with non-periodic boundary data. If the domain were not periodic, the user would see an edge box on top, an edge box on the right, and a node box at the upper right corner.

One final item to note is that, during inter-level data transfers, temporary Patch objects may be created that extend past the edge of a periodic boundary. Thus, user-code may be asked to set boundary values for a patch whose interior partially resides outside the index space of the computational domain. However, such a patch is physically reasonable due to periodicity of the domain. Figure 6 illustrates this situation. When this occurs, the boundary region to fill maintained by the BoundaryBox object will extend along the non-periodic boundary to the end of the patch ghost region. User boundary routines that rely on physical space coordinates must account for such possibilities.

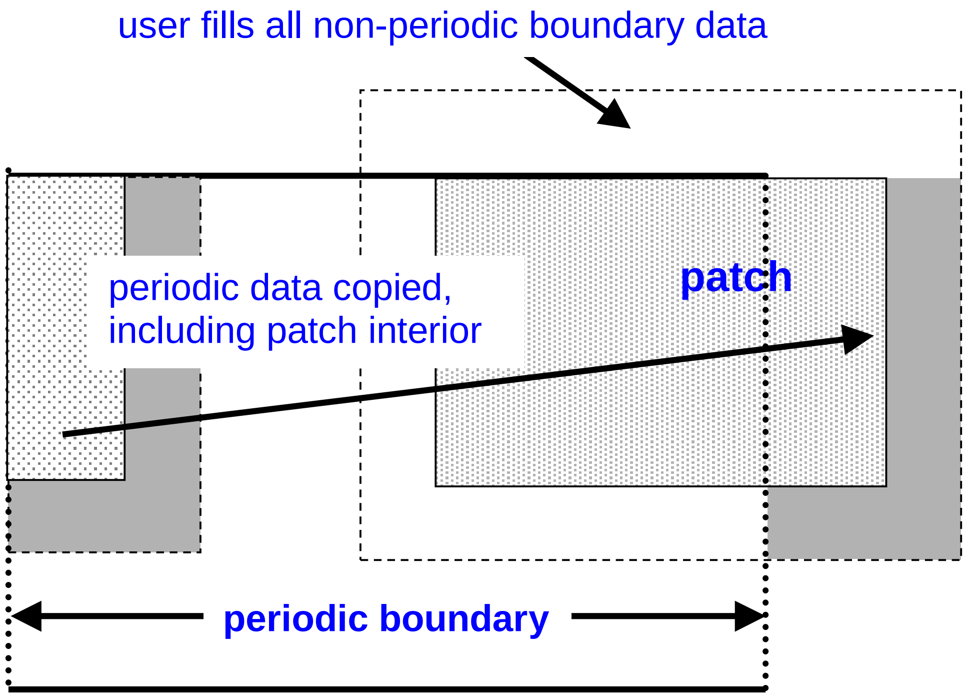
user fills all non-periodic boundary data
periodic data copied, including patch interior
patch
periodic boundary

Figure 6. When using periodic domains, it is likely that boundary data must be set for a patch that seemingly lies outside the index space of the domain. The patch is really within the domain due to periodicity. Thus, user routines must be able to treat this case.

Remarks:

- Data in physical boundary regions are set by the user-supplied routine `setPhysicalBoundaryConditions()` in the `RefinePatchStrategy` interface.
- The relationship between a patch boundary and the domain boundary is expressed using arrays of `BoundaryBox` objects that are obtained from the `PatchGeometry` object.
- Each `BoundaryBox` object has a *boundary type* and a *boundary location index.*
- The function `getBoundaryFillBox()` in the `PatchGeometry` class is used to obtain a box describing a boundary region to be filled for given patch, boundary box, and ghost width.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Document UCRL-TM-202188.