A SAMRAI Primer
(Version 0.2)

Kevin T. Chu
Department of Mechanical and Aerospace Engineering
Princeton University

August 1, 2007

Acknowledgements

I'd like to thank the SAMRAI development team for their assistance with using SAMRAI over the past several years. Without their help, it would have been impossible to learn enough about SAMRAI to be able to write this primer. I'd also like to think Brian Taylor, Adele Lim, and Zi Chen for many helpful suggestions/comments/corrections of this primer.

Contents

Chapter 1

Overview

The **S**tructured **A**daptive **M**esh **R**efinement **A**pplications **I**nfrastructure (SAMRAI) [1] is a large C++ software library developed in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). It is designed to support large-scale, parallel, structured adaptive mesh refinement (SAMR) simulations without requiring application developers to implement low-level parallel and/or SAMR algorithms. For example, SAMRAI shields application developers from the low-level programming details involved with management of bookkeeping for SAMR data structures and parallel communication. In addition, SAMRAI is designed to make it relatively easy to migrate from serial codes written to solve problems on simple, uniform grids to parallel and/or SAMR numerical simulations.

It is important to emphasize that SAMRAI does *not* take away the application developer's control over the numerics. Implementation of the core numerical kernels is still the responsibility of the application developer. SAMRAI just manages the complexities associated with SAMR data structures and programming issues.

1.1 Fundamentals of Structured Adaptive Mesh Refinement

The basic idea underlying all adaptive mesh refinement methods is that computational resources can be more efficiently utilized by carrying out a numerical simulation on a non-uniform grid where the grid resolution is higher only in localized regions of the domain that require it (see Figure ??) [2, 3, 4, 5]. For calculations on structured meshes, the computational grid is typically represented as a hierarchy of refinement levels each of which is a collection of grid cells on a structured mesh with the same resolution. Grid refinement is achieved by tagging grid cells to refine and constructing a refinement level that contains the tagged cells. While cell-based refinement[1] is one possible approach to grid refinement, SAMRAI adopts a patch-based approach, which generates refinement levels that are unions of logically rectangular blocks (known as *patches*). Because SAMRAI uses a patch-based approach to structured adaptive mesh refinement, in this primer, we shall focus solely on this approach to SAMR.

Numerical algorithms for computations on SAMR grids are not fundamentally different than those for computations on uniform grids. The most important difference is the need for special numerical schemes to handle the interfaces between refinement levels. Computations on SAMR grids require that two important numerical issues be addressed: (1) refinement (also known as interpolation or restriction) of data from coarse to fine grids and (2) coarsening (also known as prolongation) of data from fine to coarse grids. Both of these operations are important because they make it possible to couple calculations on different refinement levels. For example, refinement of coarse data at coarse-fine interfaces may be thought of as a means of providing boundary data for the fine level from the coarse level. Coarsening of fine data at the interfaces of different refinement levels can be viewed in the opposite manner – as a way to provide boundary data for the coarse level.

For patch-based SAMR methods, once data has been refined/coarsened at the interfaces between refinement levels, many numerical calculations on individual patches in the SAMR grid proceed as they would for

[1] In cell-based grid refinement, grid cells are individually refined and refinement levels are simply collections of refined cells. Quad- and oct-tree data structures are a common way to store the simulation data associated with cell-based SAMR calculations.

a uniform grid computation. For example, suppose we wanted to solve the heat equation on a SAMR grid:

$$T_t = k\nabla^2 T \tag{1.1}$$

where k is thermal conductivity. A simple numerical method to solve this equation on a SAMR grid would be to take a method of lines approach and use a forward Euler scheme to carry out the time integration. At each time step, we could use the following procedure to advance the solution in time:

1. Loop from the finest level to the coarsest level and do the following.
   (a) Fill boundary data by refining data from the next coarser grid.
   (b) Compute the numerical approximation to the spatial derivative term on all patches using standard discretizations on uniform grids.
   (c) Advance the solution in time using a forward Euler scheme.
2. Loop from the finest level to the coarsest level and coarsen the updated solution from each level to the next coarser level to obtain a consistent solution on the entire SAMR grid.

Note that the last step of this procedure serves two important purposes for the next time step: (1) it provides boundary data for coarse grids and (2) it provides data for use in the refinement stage of the computation. It is in this sense that the solution on overshadowed portions of coarse levels is considered to be consistent with the solution on finer grids. For elliptic problems or implicit time integration schemes, solution of the discretized problem can often obtained via iterative schemes based on fast adaptive composite (FAC) methods [**?**] where the procedure during each iteration has a similar flavor to the procedure described above.

1.2 Basic Structure of SAMRAI Applications

The basic structure of most SAMRAI simulations follows the same standard structure of many scientific computing applications:

1. Set up data structures for variables.
2. Carry out computations on variables (possibly writing out data, visualization, and/or restart files in the process).
3. Clean up memory and exit.

However, because SAMRAI is designed to be very general and support difficult to program features (*e.g.,* SAMR and high-performance parallel simulation), there are several SAMRAI *idioms* that must be learned in order to successfully build SAMRAI-based applications. For instance, access to data in SAMRAI is a bit indirect but follows a fairly standard set of operations. While we are often accustomed to having explicit control over memory allocation for variables during the set up phase and and having direct access to variables during the computation phase, this level of control is really only possible for calculations with simple data structures (*e.g.* serial, uniform grid calculations on simple domains). Because the data structures required for SAMR and parallel computations are relatively complicated and tedious to program, SAMRAI sacrifices the convenience of direct access to data variables in order to: (1) achieve good computational performance and/or (2) hide the complexity from the application developer. The goal is to allow application developers to focus on their particular scientific application rather than on having to deal with low-level programming issues.

Structure of Typical SAMRAI `main` Programs

Before getting in to the details of SAMRAI, let's take a look at the structure of the `main` program for a typical SAMRAI application. The following steps show how the generic procedure for scientific computations described above is expressed within the SAMRAI framework (see Appendix A for a sample `main` program). It should be noted that the following procedure is specific to simulations on Cartesian grids and takes advantage of several built-in SAMRAI classes. SAMRAI is capable of supporting more general computations, but a discussion of the advanced SAMRAI features is beyond the scope of this primer.

1. Initialize the MPI and SAMRAI environments.

2. Process command-line arguments. Set up input and restart databases. Set up logging.

3. Create objects required for computation

   (a) Set up the geometry for the problem, by creating a `CartesianGridGeometry` object.

   (b) Create `PatchHierarchy` object. Initially, the `PatchHierarchy` is empty (*i.e.,* contains no refinement levels). The individual levels of the SAMR hierarchy are added later.

   (c) Create instances of application specific subclasses of the SAMRAI strategy classes that are needed for the simulation (see Section 5.4 or [6] for more information about the *strategy* design pattern).

   (d) Create the grid management objects.

      i. Create the three objects that are required by the `GriddingAlgorithm` object:

         - a `StandardTagAndInitialize` object, which handles initializing data on the `PatchHierarchy` and tagging grid cells for refinement,
         - a `BergerRigoutsos` object, which generates boxes for new refinement levels, and
         - a `LoadBalancer` object, which distributes patches across the processors allocated for the computation.

      ii. Create the `GriddingAlgorithm` object.

4. Set up visualization. VisIt [7] is commonly used to visualize the results of SAMRAI simulations.

5. Use the `GriddingAlgorithm` object to initialize the `PatchHierarchy`. Initialization of the `PatchHierarchy` includes both creating refinement levels in the hierarchy as well as allocating and initializing the data on those levels.

   NOTE: depending on how the simulation algorithm has been decomposed into C++ classes, the `GriddingAlgorithm` may not be the object explicitly used to initialize the `PatchHierarchy` in the `main` program.

6. Carry out the main time-stepping or solver routine. It is common for data, visualization, and restart files to be written out during this stage.

7. Clean up memory allocated for the computation.

8. Shutdown the MPI and SAMRAI environments.

It is worth mentioning that several important SAMRAI objects are configured using parameters from the input file. For example, parameters from the input file are used in the construction of the `CartesianGridGeometry`, `GriddingAlgorithm`, `LoadBalancer`, and `StandardTagAndInitialize` objects.

1.3 Non-SAMR Use of SAMRAI

While SAMRAI was originally developed to support SAMR simulations, it provides several features that are useful for general scientific computing applications. These features by themselves can justify the use of SAMRAI even when SAMR is not required for the computation.

- **Parallel Computing**. SAMRAI provides strong support for parallel computing, including automatic domain decomposition and management of data communication between processors. Moreover, the parallel computing facilities in SAMRAI are specifically designed to be scalable to huge (>10K) number of processors. These features make it possible to develop high-performance parallel applications without requiring application developers to deal with the low-level details of MPI programming.

- **Restart Capabilities**. SAMRAI provides extensive check-point and restart capabilities. All major SAMRAI objects involved in computations are instrumented to support serialization/deserialization (also known as deflation/inflation). Reading to and writing from restart files are managed by the `RestartManager` class. To take advantage of check-pointing and restart, the application developer need only implement serialization/deserialization for application specific classes. For most applications, this work should be minimal and only requires knowledge of the interface provided by the SAMRAI `Database` class.
- **Input Database**. SAMRAI provides a convenient and general mechanism for processing input files through its `InputDatabase` class. Input files are organized in a hierarchical format using C/C++ syntax (including comments), which is an intuitive and natural format for complex simulations.

1.4 Scope of This Primer

This primer is intended to help a new SAMRAI application developer get familiarized with basic SAMRAI constructs and quickly develop a applications based on SAMRAI. It is *not* intended to be comprehensive or be a reference manual for the SAMRAI library. Instead, the primer takes a very practical approach to SAMRAI. The flexibility and extensibility of the SAMRAI library makes it possible to do many things in multiple ways. In this primer, I have chosen to illustrate methods that should be relatively intuitive to a scientific programmer comfortable with writing serial numerical simulations. This choice may not always lead to the most elegant way to express ideas in SAMRAI, but hopefully it will give new users a solid foundation from which to start learning the intricacies of SAMRAI on their own.

Two-dimensional Example Code

Many of the C++ classes in the SAMRAI library are templated on the number of spatial dimensions of the problem[2]. In this primer, we provide all examples in two-dimensions rather than giving examples for a general number of spatial dimensions. What this means is that the template variable for the spatial dimension will always be set to 2 instead of `DIM` in code snippets throughout the primer. For instance, `PatchHierarchy<2>` as opposed to `PatchHierarchy<DIM>` will appear in sample code.

[2]While the C++ classes support an arbitrary number of spatial dimensions, the numerical kernels for refining and coarsening data are only implemented for 1, 2, and 3 spatial dimensions.

Chapter 2

Getting Started with SAMRAI: Installation and Documentation

2.1 Obtaining SAMRAI

SAMRAI is freely available and may be downloaded from LLNL by following the "Download SAMRAI Software" link at `http://www.llnl.gov/CASC/SAMRAI/software/software.html` after filling out a short registration form.

2.2 Installing SAMRAI

The build procedure for SAMRAI follows the standard `configure; make; make install` paradigm. The only strict software dependency for SAMRAI (beyond having C, C++, and Fortran compilers) is the Hierarchical Data Format library (HDF5) developed at the University of Illinois at Urbana-Champaign [8]. HDF5 is required for the check-point/restart functionality provided by SAMRAI and visualization using VisIt, a visualization software developed at LLNL [7]. Parallel SAMRAI applications require an implementation of MPI (Message Passing Interface) to be installed before building SAMRAI. Other external libraries (*e.g.,* PETSc, Hypre, SUNDIALS, etc.) may be needed to support certain SAMRAI features, but they are not necessary for most basic SAMRAI applications.

SAMRAI is intended to be built in a directory that is *different* than the source directory. For a basic SAMRAI installation, the following procedure (with minor variations) may be used to build and install SAMRAI:

```
>CC=gcc
>CXX=g++
>F77=gfortran
>export CC CXX F77
>mkdir ${OBJ_DIR}
>cd ${OBJ_DIR}
>${SAMRAI_SRC_DIR}/configure --prefix=${SAMRAI_INSTALL_DIR}            \
                            --enable-opt=-O3                          \
                            --with-x                                  \
                            --with-hdf5=/home/ktchu/opt/hdf5          \
                            --without-petsc
>make library
>make install
```

It is worth mentioning that the SAMRAI library can take up to several hours to build depending on the platform and the compiler options that are enabled.

2.3 SAMRAI Doxygen Documentation

The entire SAMRAI library is documented using the Doxygen system. The HTML documentation provided is very convenient for quickly finding detailed information about classes and their methods. The Doxygen documentation can either be generated from the source code, downloaded directly from LLNL, or viewed online at `http://www.llnl.gov/CASC/SAMRAI/software/html/main.html`.

Chapter 3

SAMRAI Essentials: Grids, Geometry, and Simulation Data

Computational grids, physical geometry, and simulation data form the foundation of many numerical simulations. For small-scale simulations, we have the luxury of choosing not to use complicated data structures to manage these important constructs. In fact, we can often get away with just storing all relevant information in a few variables and not using data structures at all. Unfortunately, as simulations grow in size and complexity, we are forced to adopt at least a few basic software engineering techniques to keep our programs from growing out of control.

Parallel, SAMR computations are no exception to this rule. SAMRAI's goal of shielding the application developer from the programming complexities needed to support parallel, SAMR computations comes at the cost of requiring application developers to learn how to access and manipulate grid, geometry, and simulation data using a collection of SAMRAI classes. In this chapter, we highlight several of the SAMRAI classes that manage the essential components of any numerical simulation.

3.1 SAMR Grids

Hierarchy of Refinement Levels: The `PatchHierarchy`, `PatchLevel`, and `Patch` Classes

The fundamental data structure for patch-based SAMR calculations is a hierarchy of refinement levels each of which is a collection of *patches*. Each patch covers a logically rectangular block of grid cells with uniform spacing in each coordinate direction[1]. The location of the simulation data within each grid cell is flexible (*e.g.,* cell-, face-, and node-centered) and depends on the nature of the numerical algorithm.

In SAMRAI this data structure is represented by the following three classes: `PatchHierarchy`, `PatchLevel`, and `Patch`. As their names suggest, the `PatchHierarchy` manages a collection of `PatchLevel`s each of which, in turn, contains a collection of `Patches` (see Figure ??). The `PatchHierarchy` and `PatchLevel` are essentially container classes for `PatchLevel`s and `Patches`, respectively. The `Patch` class, however, holds several pieces of information involved in many numerical calculations in patch-based SAMR applications. We will discuss the `Patch` class in more detail a bit later in this chapter after we have introduced a few important ideas. In general, application developers are *not* responsible for explicitly constructing the entire patch hierarchy. This task is typically handled by a collection of classes that SAMRAI provides to manage grid generation (see Section 4.1).

In SAMRAI, simulation data (and therefore numerical computations) are directly associated with `Patches`. As a result, the following SAMRAI idiom is often seen in SAMRAI applications:

```
// loop over PatchHierarchy
const int num_levels = patch_hierarchy->getNumberLevels();
```

[1]The grid spacing need *not* be the same in all coordinate directions. For example, dx does not need to be the same as dy.

```
for ( int ln=0 ; ln < num_levels; ln++ ) {
  Pointer< PatchLevel<2> > level = patch_hierarchy->getPatchLevel(ln);

  // loop over patches
  typename PatchLevel<2>::Iterator patch_iterator;
  for (patch_iterator.initialize(level); patch_iterator; patch_iterator++) {

    // do some work ...

  }
}
```

Note that `PatchLevel`s are numbered starting at zero with higher level numbers corresponding to higher levels of grid refinement. Note also that the `Iterator` object only loops over `Patches` that reside on the local processor. Looping over the `Patches` using the patch number can lead to errors for parallel runs.

Index Spaces on SAMR Grids: The `Index` and `IntVector` Classes

To make it easy to determine the location of grid cells within the SAMR grid, SAMRAI takes a global approach to indexing. Within a `Patch` on a `PatchLevel`, the index of each grid cell (identified by the center of the grid cell) is specified relative to a common origin for the entire `PatchLevel`[2]. The relationship between indices on different `PatchLevel`s in a `PatchHierarchy` is also simple. This relationship is best illustrated by example.

Consider a two-dimensional `PatchHierarchy` where the ratio between the grid cells two levels that completely overlap each other is two in the x-direction and three in the y-direction (see Figure ??). Defining the *refinement ratio* between two different levels in a SAMR grid to be the vector of integers specifying the number of fine grid cells that a coarse grid cell is cut into in each coordinate direction, we say that refinement ratio between the two levels in this example is $(2, 3)$. To define the indices for the grid cells on the fine level, we can think of the fine level as having been generated by taking each grid cell on the coarse level and dividing it into a 2-by-3 block of grid cells on the fine level. To "create" enough indices on the fine grid for the new cells, we multiply the coarse grid index by the refinement ratio (component-wise) to obtain a base index in on the fine level. We then add an offset to this base index in order to uniquely identify the position of each fine grid cell within the subdivided coarse grid cell. For example, consider a grid cell with index (i_c, j_c) on the coarse level, the indices of the corresponding 2-by-3 block grid cells on the fine level would be $(2i_c, 3j_c)$, $(2i_c + 1, 3j_c)$, $(2i_c, 3j_c + 1)$, $(2i_c + 1, 3j_c + 1)$, $(2i_c, 3j_c + 2)$, and $(2i_c + 1, 3j_c + 2)$ ordered from left to right and bottom to top. Computing the index of the coarse grid cell corresponding to a fine grid cell with index (i_f, j_f) merely requires component-wise division of the fine grid index by the refinement ratio and rounding down to the nearest integer: $(i_c, j_c) = (\lfloor i_f/2 \rfloor, \lfloor j_f/3 \rfloor)$.

Generalization of the above example to arbitrary refinement ratios and spatial dimensions is straightforward. For an n-dimensional problem with refinement ratio $(r_1, r_2, ..., r_n)$ between two levels, the r_1-by-r_2-by-...-by-r_n block of fine grid cells associated with a coarse grid cell having index $(i_{1c}, i_{2c}, \ldots, \imath_{nc})$ is the set of indices $\{(r_1 i_{1c} + o_1, r_2 i_{2c} + o_2, \ldots, r_n i_{nc} + o_n)\}$, where $o_i = 1, 2, \ldots, r_i - 1$, and the index of the coarse grid cell associated with the fine grid cell at index $(i_{1f}, i_{2f}, \ldots, i_{nf})$ is $(i_{1c}, i_{2c}, \ldots, i_{nc}) = (\lfloor i_{1f}/r_1 \rfloor, \lfloor i_{2f}/r_2 \rfloor, \ldots, \lfloor i_{nf}/r_n \rfloor)$.

In SAMRAI, grid cell indices are represented by the `Index` class which is essentially an n-dimensional vector of integers where n is the number of spatial dimensions in the problem. The `IntVector` class, which is also a vector of integers, is a closely associated class that is involved in many grid index manipulations and operations. Intuitively, the distinction `Index` and `IntVector` objects is analogous to the difference between points and vectors in coordinate geometry.

[2] An alternative choice would be to index grid cells relative to a local origin for each `Patch` (for instance relative to the lower corner of the `Patch`)

Rectangular Blocks in Index Spaces: The `Box` Class

For patch-based SAMR simulations, the grid cells on a refinement level are organized into logically rectangular blocks, which can be uniquely defined by specifying the indices of its lower and upper corners within the global index space on the refinement level. In SAMRAI, these blocks of grid cells are represented by the `Box` class. `Box`es are typically used to specify the blocks of grid cells covered by `Patches` and expansion of these blocks to include ghost cells.

3.2 Geometry

The geometry of the computational grid is an important component of many numerical simulations. For instance, geometric information plays a key role when calculating spatial derivatives or imposing certain types of boundary conditions, such as periodic boundary conditions. For SAMR simulations, the geometry is also used when refining and coarsening data between refinement levels.

While SAMRAI is designed with a highly extensible geometry infrastructure, SAMRAI currently only provides direct support for Cartesian grids through the `CartesianGridGeometry` and `CartesianPatchGeometry` classes. The `CartesianGridGeometry` class stores information about the physical/computational domain and its boundaries, the coordinate directions that are periodic, and spatial refinement/coarsening operators[3]. Each `PatchHierarchy` object is associated with one `CartesianGridGeometry` object that provides the geometry information for the SAMR grid represented by the `PatchHierarchy`. The `CartesianPatchGeometry` class holds information associated with a specific `Patch`. Each `CartesianPatchGeometry` object is associated with a specific `Patch` and holds information about (1) the grid spacing and (2) the spatial coordinates of the lower and upper coordinates of the `Patch`. To access `CartesianGridGeometry` and `CartesianPatchGeometry` objects, we use the `PatchHierarchy::getGridGeometry()` and `Patch::getPatchGeometry()` methods:

```
// get pointer to grid geometry object
Pointer< CartesianGridGeometry<2> > grid_geom =
  patch_hierarchy->getGridGeometry();

// get pointer to patch geometry object
Pointer< CartesianPatchGeometry<2> > patch_geom =
  patch->getPatchGeometry();
```

Input Parameters

The `CartesianGridGeometry` class has four input parameters: `domain_boxes`, `x_lo`, `x_hi`, and `periodic_dimension`. Of these, only `periodic_dimension` is optional. The meanings of these parameters are as follows:

- `domain_boxes` array of boxes representing the index space for the entire domain on the coarsest refinement level. Each box is specified by the grid indices of its lower and upper corners. For example, `[(0,0), (2,3)]` is a two-by-three rectanglular region of index space corresponding to the indices $0 \leq i \leq 2$ and $0 \leq j \leq 3$.

- `x_lo` array of double values representing the spatial coordinates of the lower corner of the computational domain. If the computational domain is not a parallelipiped, then `x_lo` is taken to be the minimum over the lower corners of all of the boxes whose union make up the entire computational domain.

- `x_up` array of double values representing the spatial coordinates of the upper corner of the computational domain. If the computational domain is not a parallelipiped, then `x_up` is taken to be the maximum over the upper corners of all of the boxes whose union make up the entire computational domain.

[3] Technically, these pieces of information are stored by the parent classes of `CartesianGridGeometry`.

- `periodic_dimension` array of integer values representing the coordinate directions in which the computational domain is periodic. A non-zero value indicates that the direction is periodic; a zero value indicates that the coordinate direction is not periodic. By default, all coordinate directions are assumed to be non-periodic.

The input database for a `CartesianGridGeometry` object might look like the following:

```
CartesianGeometry {
  domain_boxes = [(0,0), (99,99)], [(100,0), (149,49)]
  x_lo = 0.0, 0.0
  x_up = 1.5, 1.0
  periodic_dimension = 0, 0
}
```

3.3 Simulation Data

Managing and accessing simulation data are central concerns for all numerical simulations. In simple programs, these tasks require no more than understanding how to create and use data arrays using standard programming language constructs (*e.g.,* `malloc()/free()` or `new/delete`). However, in SAMRAI these tasks are slightly more complicated as a result of the data structures required for SAMR grids and to allow for more flexible memory management during SAMR simulations.

Referencing Simulation Data: Data Handles

In SAMRAI applications, each simulation variable is assigned a unique integer *data handle*. We defer a detailed discussion of how to define simulation variables and generate data handles to Section 4.2. For now, it suffices to know that data handles are used to access and manage memory for the simulation data associated with each simulation variable.

Accessing Simulation Data: The `PatchData` Class

In SAMRAI, data associated with a simulation variable is managed by concrete subclasses of the `PatchData` class. In this primer, we will focus solely on `PatchData` types that are used to represent grid functions: `CellData` (cell-centered), `FaceData` (face-centered), `SideData` (face-centered), `EdgeData` (edge-centered), and `NodeData` (node-centered). Specialty data types, such as `IndexData`, are also available but will not be discussed here.

Each `PatchData` object is associated with a single `Patch` object. To obtain the `PatchData` object for a given simulation variable, we use the `getPatchData()` method from the `Patch` class passing in the simulation variable's data handle as the argument:

```
// get PatchData for cell-centered a simulation variable
// identified by data_handle
Pointer< CellData<2,double> > patch_data = patch.getPatchData( data_handle );
```

`PatchData` objects hold three important pieces of information: the number of components (known as *depth* in SAMRAI) associated with the simulation data (*e.g.,* 1 for scalar fields and 3 for vector fields in three-dimensions), the region of index space covered by the simulation data, and the pointers to actual data arrays that contain the simulation data. It is important to note that the region of index space covered by the simulation data need *not* coincide with that of the `Patch` that it is associated with. In general, the `Box` for a `PatchData` object is larger than the `Box` for the `Patch` because it possesses extra grid cells, known as *ghost cells*, which are used to provide data from neighboring patches, coarser refinement levels, or boundary conditions (see Figure ??).

For a `PatchData` object named `patch_data`, we obtain the depth of the simulation data via the `getDepth()` method:

```
int depth = patch_data->getDepth();
```

The `Box` that represents the region of index space covered by the simulation data may be determined by using the `getGhostBox()` method:

```
Box<2> ghostbox = patch_data->getGhostBox();
```

A related method is the `getGhostCellWidth()` method which returns an `IntVector` representing the ghost cell width in each coordinate direction.

We retrieve `Pointer`s to data arrays containing simulation data by using the `getPointer()` method. The argument to this function is an integer which indicates the desired component (zero-based numbering) of the simulation data. For instance,

```
double* simulation_data = patch_data->getPointer(1);
```

returns the second component of simulation data managed by `patch_data`. Once we have the pointer to the actual simulation data, we can proceed with the numerical calculation exactly as we would on a uniform grid. For a calculation that involves looping over a two-dimensional grid, the following C/C++ code could be used:

```
// get dimensions of box that covers interior of Patch
Box<2> interior_box = patch_data->getBox();
const IntVector<2> interior_dims = interior_box.numberCells();

// get dimensions of box that covers interior of Patch plus ghost cells
Box<2> ghostbox = patch_data->getGhostBox();
const IntVector<2> ghostcell_dims = ghostbox.numberCells();
const IntVector<2> ghostcell_width = patch_data->getGhostCellWidth();

// loop over grid
for (int j = 0; j < interior_dims(1); j++) {
  for (int i = 0; i < interior_dims(0); i++) {

    // compute index into linear data array
    // NOTE: the data in simulation_data is stored in Fortran order
    int idx = (i+ghostcell_width(0)
            + (j+ghostcell_width(1))*ghostbox_dims(0));

    // do some calculations
    simulation_data[idx] = ...

  }
}
```

Fortran Ordering of Simulation Data

An important issue when dealing with multidimensional data that are stored in a linear array is *how* the data are ordered in the linear array. In SAMRAI, data are assumed to be stored in the Fortran order (also known as column-major order). The primary reason for this choice is performance. In order to achieve high programmer performance (*i.e.,* well-designed APIs and high level data structures) while maintaining high computational performance, SAMRAI advocates a multiple-language approach to simulation development. High programmer performance is achieved by using C++ to design the programming interfaces; high computational performance is achieved by using Fortran to implement calculation intensive numerical kernels. For example, the numerical kernels for refinement/coarsening of data and the example programs are implemented in Fortran.

It is important to point out, however, that while Fortran may be the language of choice for some developers, it is perfectly acceptable to write numerical kernels in C/C++. The user need only respect the Fortran ordering for multi-dimensional data to maintain compatibility with the built-in SAMRAI refinement

and coarsening operators. The example code in the previous section provides an example of the correct way to compute the index (`idx`) into the data array given the grid index (i, j).

For those C/C++ programmers that might balk at the thought of programming in Fortran, it is worth considering the ease of expressing many mathematical formulae (especially those involving data on multidimensional grids) in the Fortran language. For example, numerical schemes for computing derivatives are very easy to implement thanks to support for multidimensional indexing into data arrays built into the Fortran language.

Managing Memory for Simulation Data

In SAMRAI, the memory for simulation data is allocated/deallocated either (1) simultaneously for all of the `Patches` on an entire `PatchLevel` or (2) for each `Patch` individually. While the former method is convenient, improved memory usage can often be achieved using the latter method. For example, because computations on different `Patches` are decoupled once boundary data has been exchanged between neighboring `Patches` and `PatchLevels`, there is no need to simultaneously allocate data arrays used for intermediate calculations on all `Patches` on a `PatchLevel`.

To allocate memory for a given simulation variable, we use the `allocatePatchData()` method from the `PatchLevel` or `Patch` classes passing in the simulation variable's data handle as the argument:

```
// allocate memory for simulation variable identified by data_handle
// on all of the Patches on the PatchLevel named plevel
plevel.allocatePatchData( data_handle );

// allocate memory for simulation variable identified by data_handle
// on a single Patch named patch
patch.allocatePatchData( data_handle );
```

Deallocating memory for a simulation variable proceeds in an analogous manner using the `deallocatePatchData()` method from the `PatchLevel` or `Patch` classes.

For computations involving many simulation variables, it is convenient to group variables together using the `ComponentSelector` class. Both the `Patch` and `PatchLevel` classes provide versions of the `allocatePatchData()` and `deallocatePatchData()` methods that take a `ComponentSelector` object as an argument.

3.4 Pulling It All Together: Computations on Indiviudal Patches

An important principle underlying patch-based SAMR is that the calculations performed on patches should closely mimic those which would be performed for a numerical simulation on a uniform grid. This principle motivates the design of the SAMRAI `Patch` class which acts as a central coordinator for the grid, geometry, and simulation data required to carry out a computation on a single patch – `Patch` objects provide access to all of the information needed for a numerical calculation on the block of uniform grid cells contained within the `Box` covered by the `Patch`.

A typical computation on a `Patch` begins by determining the index space covered by `Patch` and the geometry configuration for the `Patch`. For a simulation on a Cartesian grid, this information can be obtained from a `Patch` object named `patch` using a variation of the following code:

```
// get dx and coordinates of lower corner of patch
Pointer< CartesianPatchGeometry<2> > patch_geom = patch.getPatchGeometry();
const double* dx = patch_geom->getDx();
const double* x_lower = patch_geom->getXLower();

// get grid cell indices for the upper and lower corner of the region
// of index space covered by patch
```

```
Box<2> box = patch.getBox();
const Index<2> box_lower = box.lower();
const Index<2> box_upper = box.upper();
```

Next, the simulation data and the index space associated with it are determined. For example, if we have cell-centered data with only a single component, we might use the following code:

```
// get CellData associated with data_handle
Pointer< CellData<2,double> > patch_data = patch.getPatchData( data_handle );

// get grid cell indices for the upper and lower corner of the region
// of index space covered by patch expanded to include the ghost cells
// associated with patch_data
Box<2> ghostbox = patch_data->getGhostBox();
const Index<2> ghostbox_lower = ghostbox.lower();
const Index<2> ghostbox_upper = ghostbox.upper();

// get pointer to actual simulation data
double* simulation_data = patch_data->getPointer();
```

Finally, we carry out the desired numerical calculation on the simulation data.

Chapter 4

SAMRAI Setup: The Dirty Work

4.1 Managing the SAMR Grid

Construction and management of the SAMR grid (especially in parallel) are relatively involved processes. In SAMRAI, these important procedures are coordinated by the `GriddingAlgorithm` class. This class (in conjunction with several supporting classes) provides support for creating and initializing data SAMR grids, dynamic reconfiguration (also known as regridding) of SAMR grids, decomposition of the computational domain and `PatchLevel`s into `Box`es, and distribution of `Patches` across processors for parallel simulations.

There are three primary objects that each `GriddingAlgorithm` object relies on to provide its functionality: a `BergerRigoutsos` object, a `LoadBalancer` object, and a `StandardTagAndInitialize` object. The `BergerRigoutsos` class implements the Berger-Rigoutsos algorithm for generating boxes given a collection of grid cells which should be covered by the boxes. The `LoadBalancer` class handles distribution of `Patches` across processors in parallel simulations. Finally, the `StandardTagAndInitialize` class manages initialization of simulation data on newly generated or reconfigured SAMR grids and *tagging* (*i.e.,* identification) of grid cells on a `PatchLevel` that should be refined.

The procedure for creating a two-dimensional `GriddingAlgorithm` object is as follows:

```
Pointer< StandardTagAndInitialize<2> > standard_tag_and_init_obj =
  new StandardTagAndInitialize<2>(
    "StandardTagAndInitialize", my_application_obj,
    input_db->getDatabase("StandardTagAndInitialize"));
Pointer< BergerRigoutsos<2> > box_generator = new BergerRigoutsos<2>();
Pointer< LoadBalancer<2> > load_balancer =
  new LoadBalancer<2>("LoadBalancer",
                      input_db->getDatabase("LoadBalancer"));
Pointer< GriddingAlgorithm<2> > gridding_alg = new GriddingAlgorithm<2>(
  "GriddingAlgorithm",
  input_db->getDatabase("GriddingAlgorithm"),
  standard_tag_and_init_obj,
  box_generator,
  load_balancer);
```

The `my_application_obj` that is passed as an argument to the constructor for the `StandardTagAndInitialize` object is an instance of a user-defined subclass of the `StandardTagAndInitStrategy` class (discussed below) that implements application specific initialization and tagging routines.

Input Parameters

The `GriddingAlgorithm`, `StandardTagAndInitialize`, and `LoadBalancer` can all be configured via parameters specified in an input file. In this section, we briefly describe the input parameters that may be used

for each of these objects.

GriddingAlgorithm

There are three required input parameters for the `GriddingAlgorithm` class: `max_levels`, `largest_patch_size`, and `ratio_to_coarser`. The meanings of these parameters are as follows:

- `max_levels` is the maximum number of levels allowed in the SAMR grid hierarchy
- `largest_patch_size` is an array of integer vectors that specify the dimensions of the largest patch allowed on each level of the SAMR grid hierarchy. The database key for each integer vector has the form `level_`i where i indicates the number of the `PatchLevel` and ranges from 0 to (`max_levels` -1). For example, if the largest patch size is 50 by 50 on level 0 and 25 by 25 on level 1 (for a 2D simulation), the input file entry would look like:

  ```
  largest_patch_size {
    level_0 = 50, 50
    level_1 = 25, 25
  }
  ```

  If there are fewer entries than the maximum number of levels in the SAMR grid hierarchy, then the last entry will be used for all levels without a specified input value. If extra entries are given, they will be ignored.

- `ratio_to_coarser` is a set of (`max_level` -1) integer vectors which indicate the ratio of the index space of each `PatchLevel`to the next coarser level. The database key for each integer vector has the form `level_`i where i indicates the number of the `PatchLevel` and ranges from 1 to (`max_levels` -1). For example, if the ratio of level 1 to level 0 is 2 in each coordinate direction, the input file entry would look like:

  ```
  ratio_to_coarser {
    level_1 = 2, 2
  }
  ```

  It is an error for there to be fewer entries than (`max_levels` -1) in the `ratio_to_coarser` input database.

There are also several optional input parameters. For information on these parameters, we refer the reader to the header file or Doxygen documentation for the `GriddingAlgorithm` class.

A sample input database for a `GriddingAlgorithm` object might look like:

```
GriddingAlgorithm {

  max_levels = 3       // Maximum number of levels in hierarchy.

  ratio_to_coarser {   // vector ratio to next coarser level
    level_1 = 4 , 4
    level_2 = 4 , 4
    level_3 = 4 , 4
  }

  largest_patch_size {
    level_0 = 40 , 40
    // all finer levels will use same values as level_0...
  }

}
```

StandardTagAndInitialize

There are no required input parameters for the `StandardTagAndInitialize` class. There are, however, a few optional input parameters which are often used: `tagging_method` and `RefineBoxes`. The meanings of these parameters are as follows:

- `tagging_method` is a string specifying the method that should be used to tag cells for refinement. Valid taggging methods include any combination of: `GRADIENT_DETECTOR`, `REFINE_BOXES`, and `RICHARDSON_EXTRAPOLATION`. If `tagging_method` includes `GRADIENT_DETECTOR`, the "gradient detection" algorithm implemented in the user-defined `applyGradientDetector()` is used to tag cells. Note that the "gradient detection" algorithm need not have anything to do with gradients of the field variables. If `tagging_method` includes `REFINE_BOXES`, the list of boxes listed in the `RefineBoxes` input parameter is used to tag cells. Finally, if `tagging_method` includes `RICHARDSON_EXTRAPOLATION`, the Richardson extrapolation algorithm implemented in the user-defined `applyRichardsonExtrapolation()` is used to tag cells.

- `RefineBoxes` is a collection of `Boxes` on each level that should be tagged for refinement. The database key for each level has the form `level_`i where i indicates the number of the `PatchLevel` and ranges from 0 to (`max_levels` $-$ 2). For example, if the `Boxes` $[(0,0),(9,15)]$ and $[(10,10),(16,16)]$ on level 0 should be tagged for refinement, the input file entry would look like:

  ```
  RefineBoxes {
    level_0 = [(0,0),(9,15)], [(10,10), (16,16)]
  }
  ```

  Note that the `RefineBoxes` input parameters are only used if `tagging_method` includes `REFINE_BOXES`.

A sample input database for a `StandardTagAndInitialize` object might look like:

```
StandardTagAndInitialize {
  tagging_method = "GRADIENT_DETECTOR", "REFINE_BOXES"

  RefineBoxes {
    level_0 = [(15,30),(74,69)],[(75,50),(94,89)]
    level_1 = [(73,150),(199,249)],[(225,225),(349,249)]
    level_2 = [(325,650),(399,799)]
  }
}
```

LoadBalancer

There are no required input parameters for the `LoadBalancer` class. The default parameter values are usually sufficient. There is, however, one optional input parameter which can be useful: `processor_layout`. The `processor_layout` is an integer array with size equal to the number of spatial dimensions indicating the way that the domain should be chopped up along each coordinate direction when a level can be described as a single parallelipiped region. If no input value is provided, or the product of the `processor_layout` values does not equal the number of procesors, then the processor layout will be computed automatically by SAMRAI. A sample input database for the `LoadBalancer` object might look like:

```
LoadBalancer {
  processor_layout = 2, 8
}
```

Application Specific Functions/Routines: The `StandardTagAndInitStrategy` Class

Initialization of simulation data and tagging of grid cells for refinement are always application specific. To make use of the grid management classes provided by SAMRAI, application developers are required to

provide implementations for these procedures in a concrete subclass of the `StandardTagAndInitStrategy` class. There are three methods that application developers typically need to be concerned with: `initializeLevelData()`, `resetHierarchyConfiguration()`, and `applyGradientDetector()`. In addition, for simulations that use Richardson extrapolation to tag grid cells for refinement, application developers must provide implementations for the `applyRichardsonExtrapolation()` and `coarsenDataForRichardsonExtrapolation()` methods.

As its name suggests, the `initializeLevelData()` method is used to initialize[1] the simulation data on a `PatchLevel` that has been newly added to a `PatchHierarchy`. The signature for the `initializeLevelData()` method is:

```
void initializeLevelData(const Pointer< BasePatchHierarchy<2> > hierarchy,
                         const int level_number,
                         const double init_data_time,
                         const bool can_be_refined,
                         const bool initial_time,
                         const Pointer< BasePatchLevel<2> > old_level =
                               Pointer< BasePatchLevel<2> >(NULL),
                         const bool allocate_data = true);
```

`hierarchy` is a pointer to a `BasePatchHierarchy` object. For our purposes, it can just be thought of as a pointer to a `PatchHierarchy` object[2]. `level_number` is the number of the newly added `PatchLevel`. `init_data_time` is the simulation time at which the data on the `PatchLevel` is being initialized. `allocate_data` is a boolean flag indicating whether memory for the simulation data needs to be allocated. During regridding operations, the new `PatchLevel` introduced may be replacing a previously existing `PatchLevel`. In this case, the `old_level` argument, which is a pointer to the old `PatchLevel`, is not null. It is up to the application developer to write the code to transfer data from the old `PatchLevel` to the new `PatchLevel` (*e.g.,* using the data transfer procedure described in Section 4.3). The remaining two arguments are flags indicating whether the new level can be refined (*i.e.,* if the new level has the finest resolution in the SAMR grid) and whether the current call to `initializeLevelData()` is at the initial simulation time. These arguments are provided in case the initialization procedure is different in these situations.

The `resetHierarchyConfiguration()` method is used to update any aspects of the simulation algorithm that depend on the configuration of the SAMR grid. For example, the communication schedules used to transfer data between `Patches` and `PatchLevels` are typically recomputed after any changes to the SAMR grid configuration. The signature of the `resetHierarchyConfiguration()` method is:

```
void resetHierarchyConfiguration(
  const Pointer< BasePatchHierarchy<2> > hierarchy,
  const int coarsest_level,
  const int finest_level);
```

`hierarchy` is a pointer to a `BasePatchHierarchy` object. As before, we can just think of it as a pointer to a `PatchHierarchy` object. `coarsest_level` and `fineest_level` are the coarsest and finest levels in the SAMR grid.

The `applyGradientDetector()` method is used to tag cells for refinement. Its name reflects the original refinement criterion used by SAMR simulations – grid cells with large gradients in the field variables were tagged for refinement. However, the application developer is free to choose the criteria used to tag cells for refinement. The signature for the `applyGradientDetector()` method is:

```
void applyGradientDetector(const Pointer< BasePatchHierarchy<2> > hierarchy,
                           const int level_number,
                           const double error_data_time,
                           const int tag_handle,
                           const bool initial_time,
                           const bool uses_richardson_extrapolation_too);
```

[1] If `allocate_data` is true, then it is also necessary to allocate memory for the simulation data before it is initialized. Memory management for simulation data is discussed in Section 3.3.

[2] The `BasePatchHierarchy` is used to support advanced SAMR simulations.

hierarchy is a pointer to a BasePatchHierarchy object. As before, we can think of it as a pointer to a PatchHierarchy object. level_number is the number of the PatchLevel on which to tag grid cells for refinement. error_data_time is the current simulation time. tag_handle is the data handle for the cell-centered, integer data that is used to mark grid cells for refinement. The data associated with tag_handle should be set to 1 in grid cells that are marked for refinement. The remaining two arguments are flags indicating whether the current call to applyGradientDetector() is at the initial simulation time and whether the Richardson extrapolation is also being used to identify grid cells that need to be refined. These arguments are provided in case the tagging procedure is different in these situations.

Initializing the SAMR Grid and Simulation Data

Once the GriddingAlgorithm object has been set up, it is straightforward to construct the SAMR grid and initialize simulation data on it. The GriddingAlgorithm class provides two methods that are used to create the PatchLevels in the PatchHierarchy: makeCoarsestLevel() and makeFinerLevel(). As its name suggests, makeCoarsestLevel() is used to create the coarsest level (typically level zero) in the PatchHierarchy. makeFinerLevel() is used to create all finer levels in the PatchHierarchy. When initializing the SAMR grid and simulation data at the beginning of a simulation, the following procedure can be used:

```
// create and initialize coarsest level of PatchHierarchy
gridding_alg->makeCoarsestLevel(patch_hierarchy, time);

// create and initialize all finer levels
bool initial_time = true;
int tag_buffer = 0;
for (int level_num = 0;
     gridding_alg->levelCanBeRefined(level_num);
     level_num++) {

  gridding_alg->makeFinerLevel(patch_hierarchy, time,
                               initial_time, tag_buffer);
}
```

The initial_time argument to makeFinerLevel() indicates that the fine level is being added at the initial simulation time. The tag_buffer argument indicates the minimum number of grid cells between tagged grid cells and the edge of the newly created PatchLevel. The buffer is important in order to ensure that features that require high grid resolution remain on the fine grid until the PatchHierarchy is regridded. The number of PatchLevels in the PatchHierarchy depends on the input parameters used to construct the GriddingAlgorithm object. Note that when makeFinerLevel() is invoked, no finer level will be created if no regions on the current finest level in the PatchHierarchy require refinement.

Reconfiguring the SAMR Grid

As a simulation progresses, the regions of space that require higher grid resolution may move through the computational domain. As a result, the SAMR grid may need to be periodically reconfigured to reflect changes in the resolution requirements of the solution. In SAMRAI, the regridAllFinerLevels() method in the GriddingAlgorithm class manages the regridding process. The signature for the regridAllFinerLevels() method is:

```
void regridAllFinerLevels (Pointer< BasePatchHierarchy<2> > hierarchy,
                           const int level_number,
                           const double regrid_time,
                           const Array<int> &tag_buffer,
                           Array<double> regrid_start_time=Array<double>(),
                           const bool level_is_coarsest_to_sync=true);
```

`hierarchy` is a pointer to a `BasePatchHierarchy` object, which we continue to think of as a pointer to a `PatchHierarchy` object. `level_number` is the level number of the coarsest `PatchLevel` in the `PatchHierarchy` that will be regridded. `regrid_data_time` is the simulation time at which regridding is taking place. `tag_buffer` is an array of integers indicating the minimum number of grid cells between tagged grid cells and the edge of each newly regridded `PatchLevel`. The i-th element `tag_buffer` specifies the width of the buffer the i-th `PatchLevel`. The remaining two arguments are for advanced SAMR applications, so it is usually safe to just accept the default values.

4.2 Creating Simulation Variables

In simple programs, simulation variables are defined by the variable names given to the data arrays within the program. In SAMRAI, this direct connection between simulation variables and variable names in programs is broken in order to support general SAMR calculations. Instead, each simulation variable is assigned a unique integer *data handle*[3] which is used to provide a consistent way of accessing information associated with a simulation variable throughout the computation. SAMRAI's approach to variables also makes it easier to manage large, complex simulations because it allows the application developer to think in terms of physical/mathematical variables rather than program variables.

Management of simulation variables is primarily handled by the `VariableDatabase` class. An important property of the `VariableDatabase` class is that it is a *singleton* class, which means that only one `VariableDatabase` object can exist at a time. A pointer to the singleton `VariableDatabase` object can be obtained by using the static `getDatabase()` method in the `VariableDatabase` class:

```
// get pointer to VariableDatabase object for 2D simulation
VariableDatabase<2>* variable_db = VariableDatabase<2>::getDatabase();
```

The concept of a simulation variable in SAMRAI is a combination of two notions: (1) a physical/mathematical variable and (2) the context in which the variable is used in the simulation. The context for a physical/mathematical variable can be thought of as a label attached to the variable. For instance, the different stages of a time-integration scheme (*e.g.,* "CURRENT" and "NEXT") can be thought of as different contexts. Different contexts might also be used if the physical/mathematical variable appears in different parts of a multi-physics computation (*e.g.,* "FLUID FLOW" and "TRANSPORT"). In SAMRAI, physical/mathematical variables and contexts are represented by the `Variable` and `VariableContext` classes, respectively. Each `Variable-VariableContext` pair uniquely defines a SAMRAI simulation variable.

As with `PatchData`, there are several subclasses of the `Variable` class that reflect the type of data that the `Variable` represents. We will focus solely on those data types that are used to represent grid functions: `CellVariable` (cell-centered), `FaceVariable` (face-centered), `SideVariable` (face-centered), `EdgeVariable` (edge-centered), and `NodeVariable` (node-centered). In addition to specifying the centering of the data for a simulation variable, each `Variable` object has a depth and name associated with it. Because `VariableContext` objects are essentially labels for `Variable` objects, each `VariableContext` object only has a name associated with it.

To define a SAMRAI simulation variable (including any temporary variables for intermediate calculations), we use the following procedure:

1. Create a `Variable` object with the desired "centering", data type, depth and name. For example, for a scalar, cell-centered, double-precision variable, we would create a `CellVariable<double>` object named "temperature" with a depth of 1:

   ```
   // create 2D CellVariable named "temperature" with type "double" and depth 1
   Pointer< CellVariable<2,double> > temperature_variable =
     new CellVariable<2,double>("temperature",1);
   ```

[3] Within the SAMRAI library, data handles go by the name of `PatchData` indices. In this primer, we have adopted the data handle terminology because (1) the origin of the name PatchData index is not readily transparent to users not familiar with the internal implementation details of the SAMRAI library and (2) the term index is already overloaded for grid indices and indices into data arrays.

Note that creation of a simulation variable does *not* require any information about the memory requirements for the computational grid. In SAMRAI, the notion of a simulation variable is completely independent of the memory required to store the associated simulation data.

Alternatively, if a variable has already been registered with the `VariableDatabase`, a pointer to it can be retrieved by using the `getVariable()` method:

```
// get pointer to 2D CellVariable named "temperature"
Pointer< CellVariable<2,double> > temperature_variable =
  variable_db->getVariable("temperature");
```

2. Create a `VariableContext` object with the desired name using the `getContext()` method from the `VariableDatabase` class:

```
// get "CURRENT" VariableContext from the VariableDatabase
Pointer< VariableContext > current_context = variable_db->getContext("CURRENT");
```

3. Register `Variable-VariableContext` pair with `VariableDatabase` using the `registerVariableAndContext()` method. This method also takes an `IntVector` that specifies the ghost cell width required by the `Variable-VariableContext` pair in each coordinate direction. For example, if we want to register a `Variable-VariableContext` pair with a ghost cell width of one in all coordinate directions, we could use the following code:

```
// set the number of ghostcells for data to 1 in all directions
IntVector<2> one_ghostcell(1);

// register variable-context pair (with specified ghostcell width)
// with the VariableDatabase to get data handle
data_handle = variable_db->registerVariableAndContext(
  temperature_variable, current_context, one_ghostcell);
```

Note that is it *how* the `Variable` is used that determines the memory/storage requirements, which is why this information is only specified when a `Variable-VariableContext` pair is registered.

When registering `Variable-VariableContext` pairs with the `VariableDatabase`, there are a few important programming details to keep in mind. The `VariableDatabase` object does *not* create a deep copy of `Variable` and `VariableContext` objects that are registered with it. As a result, all `Variable` and `VariableContext` objects *must* be dynamically allocated. Otherwise, errors may occur when the `VariableDatabase` tries to access those objects later in in the simulation. Freeing `Variable` and `VariableContext` objects registered with the `VariableDatabase` will lead to the same problem. To avoid these problems, it is best, in general, to use smart-pointers (see Section **??**) when dealing with `Variable` and `VariableContext` objects.

Occasionally, it may be necessary to recover the `Variable` and/or `VariableContext` associated with a data handle. The `mapIndexToVariable()` and `mapIndexToVariableAndContext()` methods in the `VariableDatabase` class are available for this purpose.

4.3 Moving Data Between `Patches` and `PatchLevels`

There are many stages in parallel and SAMR simulations where data needs to be transferred between `Patches` and `PatchLevels`. In this section, we discuss the most common situations which require data transfer: (1) filling ghost cells (*i.e.,* boundary data) from neighboring patches and coarser levels, (2) copying data from one simulation variable to another on the overlap between patches (*e.g.,* when the SAMR grid is updated), and (3) computing a consistent representation of the solution across levels of the SAMR grid hierarchy.

The general procedure for all data transfer operations in SAMRAI can be broken down into three steps. First, we specify the simulation variables involved in the data transfer (*i.e.,* source, destination, and scratch

space data handles). At this point in time, we also select the desired coarsen/refinement operator to use when interlevel data transfer is required. Next, we construct the communication schedules that manage the transfer of data between patches (possibly between processors). Finally, we execute the communication schedules to actually move the data between patches (again, possibly between processors).

Filling Ghost Cell Data

Ghost cell data for individual patches is filled in one of three ways:

1. by copying data from neighboring `Patches` on the same `PatchLevel`,
2. by refining data from next coarser `PatchLevel` when there is no neighboring `Patch` on the same `PatchLevel`, or
3. by calling user-defined routines that impose boundary conditions at the boundaries of the computational domain.

SAMRAI provides direct support for the first two of these via the `RefineAlgorithm` and `RefineSchedule` objects. Setting ghost cells based on boundary conditions is also managed by these objects but only to the extent that they call user-defined functions for ghost cells that are outside of the computational domain. Ultimately, it is the user's responsibility to correctly set the values for these ghost cells. Setting boundary conditions is discussed in further detail in Section 4.4.

To fill ghost cell data for one or more simulation variables, we use the following procedure:

1. Create a `RefineAlgorithm` object using the default constructor. For example, to create a `RefineAlgorithm` object named `refine_alg`, we use:

   ```
   Pointer< RefineAlgorithm<2> > refine_alg = new RefineAlgorithm<2>;
   ```

2. For each simulation variable whose ghost cells need to be filled, use the `RefineAlgorithm::registerRefine()` method to specify the data handles for the source, destination and scratch simulation variables. When filling ghost cell data, it is not uncommon for these three data handles to be the same. In addition, the user must specify the refinement operator to use when data from coarser levels must be used to fill ghost cell data. The refinement operators provided by SAMRAI are discussed in more detail in Section 4.3.

   For example, to set up the `RefineAlgorithm` `refine_alg` to fill the ghost cells of the simulation variable associated with `data_handle` using the specified `refine_op`, we could use the following code:

   ```
   refine_alg->registerRefine(
     data_handle,  // destination data handle
     data_handle,  // source data handle
     data_handle,  // scratch space data handle
     refine_op);
   ```

   If `refine_op` is `NULL`, no interpolation will be used to fill destination data that requires refinement of data from coarser levels (*i.e.,* "constant" refinement will be used). Notes that in order to use `data_handle` as the scratch space, the PatchData associated with `data_handle` *must* have enough ghost cells to carry out the requested refinement operation.

3. For each level in the SAMR grid hierarchy, use one of the versions of the `createSchedule()` method in the `RefineAlgorithm` class that creates a communication schedule for copying data from interiors of source data to the interior and ghost cells of destination data.

   - When the source data required to fill the interior and ghost cells of the destination data reside on the current and next coarser `PatchLevels`, we use the version of `createSchedule()` that takes as arguments a pointer to the current `PatchLevel`, the level number of the next coarser `PatchLevel`, a pointer to the `PatchHierarchy` containing the current and next coarser `PatchLevels`, and a pointer to a user-defined concrete subclass of the `RefineStrategy` class:

```
Pointer< RefineSchedule<2> > refine_schedule = refine_alg->createSchedule(
  level,                  // pointer to PatchLevel
  next_coarser_level_num, // level number of next coarser PatchLevel
  hierarchy,              // pointer to PatchHierarchy
  refine_strategy);       // pointer to user-defined subclass of
                          // RefineStrategy class
```

When this version of `createSchedule()` is used, interpolation from data on the next coarser `PatchLevel` is used to fill destination data when source data is not available on the current `PatchLevel`.

- When the source and destination data for all reside on the same `PatchLevel` (or there is no need to fill destination data on grid cells that are not available on the current `PatchLevel`), we use the version of `createSchedule()` that takes as arguments a single pointer to a `PatchLevel` and a pointer to a user-defined concrete subclass of the `RefineStrategy` class:

```
Pointer< RefineSchedule<2> > refine_schedule = refine_alg->createSchedule(
  level,             // pointer to PatchLevel
  refine_strategy);  // pointer to user-defined subclass of
                     // RefineStrategy class
```

In both cases, the user-defined subclass of the `RefineStrategy` class provides support for imposing boundary conditions at the boundaries of the computational domain.

4. Fill the ghostcells (and interiors if the source and destination data handles are not the same) using the `RefineSchedule::fillData()` method. For example, to fill ghost cell data at `time`, we could use the following code:

```
refine_schedule->fillData(time, true);
```

The second argument is set to `true` when boundary conditions at the boundaries of the computational domain should be imposed. When it is not necessasry to impose the boundary conditions, the second argument should be set to `false`.

Note that the first and second steps only depend on data that are being moved and are independent of the configuration of the SAMR grid. As a result, they typically are only done once when setting up the simulation. The third step depends on the current configuration of the SAMR grid hierarchy, so it needs to be done every time the hierarchy is reconfigured. Finally, the last step needs to be done any time the actual simulation data has changed.

Copying Data Between Simulation Variables

To copy data between simulation variables, we follow the same general procedure as when we fill ghost cell data. In fact, when source data for grid cells that do not require refinement of data from the next coarser `PatchLevel` resides on the same `PatchLevel` as the destination data, the procedure is exactly same as when we fill ghost cell data (see Section 4.3). Whenever the SAMR grid is reconfigured, however, data needs to be transferred between two *different* `PatchLevel`s (in addition to refining data from coarser levels where necessary). In this situation, the third step in the procedure described in Section 4.3 is modified to use one of the following two versions of the `createSchedule()` method that takes two pointers to `PatchLevel`s as arguments, one for the source data and one for the destination data.

- When source data required to fill the destination data on the destination `PatchLevel` come from the source and the next coarser `PatchLevel`s, we use the version of `createSchedule()` that takes as arguments a pointer to the destination `PatchLevel`, a pointer to the source `PatchLevel`, the level number of the next coarser `PatchLevel`, a pointer to the `PatchHierarchy` containing the current and next coarser `PatchLevel`s, and a pointer to a user-defined concrete subclass of the `RefineStrategy` class:

```
Pointer< RefineSchedule<2> > refine_schedule = refine_alg->createSchedule(
  dst_level,              // pointer to destination PatchLevel
  src_level,              // pointer to source PatchLevel
  next_coarser_level_num, // level number of next coarser PatchLevel
  hierarchy,              // pointer to PatchHierarchy
  refine_strategy);       // pointer to user-defined subclass of
                          // RefineStrategy class
```

When this version of `createSchedule()` is used, interpolation from data on the next coarser `PatchLevel` is used to fill destination data when source data is not available on the source `PatchLevel`.

- When all of the source data required to fill the destination data resides on the source `PatchLevel` (or there is no need to fill destination data on grid cells that are not available on the source `PatchLevel`), we use the version of `createSchedule()` that takes as arguments a pointer to the destination `PatchLevel`, a pointer to the source `PatchLevel`, and a pointer to a user-defined concrete subclass of the `RefineStrategy` class:

```
Pointer< RefineSchedule<2> > refine_schedule = refine_alg->createSchedule(
  dst_level,         // pointer to destination PatchLevel
  src_level,         // pointer to source PatchLevel
  refine_strategy);  // pointer to user-defined subclass of
                     // RefineStrategy class
```

Coarsening Data

The procedure for coarsening data in SAMRAI is similar to the procedure for filling ghost cell data and copying data between simulation variables. The main difference is that `RefineAlgorithm` and `RefineSchedule` objects are replaced by `CoarsenAlgorithm` and `CoarsenSchedule` objects. Coarsening data is relatively straightforward compared to refining/copying data because there is only one type of communication schedule that can be created.

To coarsen data for one or more simulation variables, we use the following procedure:

1. Create a `CoarsenAlgorithm` object using the default constructor. For example, to create a `CoarsenAlgorithm` object named `coarsen_alg`, we use:

   ```
   Pointer< CoarsenAlgorithm<2> > coarsen_alg = new CoarsenAlgorithm<2>;
   ```

2. For each simulation variable whose data needs to be coarsened, use the `CoarsenAlgorithm::registerCoarsen()` method to specify the data handles for the source and destination simulation variables. When coarsening data, it is not uncommon for these two data handles to be the same. In addition, the user must specify the coarsening operator that should be used to coarsen data from the coarse to the fine level. The coarsening operator provided by SAMRAI is discussed in more detail in Section 4.3.

   For example, to set up the `CoarsenAlgorithm coarsen_alg` to coarsen the data for the simulation variable associated with `data_handle` using the specified `coarsen_op`, we could use the following code:

   ```
   coarsen_alg->registerCoarsen(
     data_handle,  // destination data handle
     data_handle,  // source data handle
     coarsen_op);
   ```

   In special situations, it may be necessary to fill ghost cell regions on the coarse level using data from the ghost cell regions of the fine level. In this case, the optional fourth argument of `registerCoarsen()` should be set to an `IntVector` which indicates the number of ghost cells on the coarse level that need to be filled. Note that the fine level *must* have enough ghost cells to cover the entire ghost cell region of the coarse level.

3. For each level except the coarsest level in the SAMR grid hierarchy, create a communication schedule for coarsening data from the level to the next coarser level:

```
Pointer< CoarsenSchedule<2> > coarsen_schedule = coarsen_alg->createSchedule(
  coarse_level,  // pointer to coarse (destination) PatchLevel
  fine_level);   // pointer to fine (source) PatchLevel
```

4. Coarsen the data using the `CoarsenSchedule::coarsenData()` method:

```
coarsen_schedule->coarsenData();
```

The first and second steps only depend on data that are being moved and are independent of the configuration of the SAMR grid, so they typically are only done once when setting up the simulation. The third step depends on the current configuration of the SAMR grid hierarchy, so it needs to be done every time the hierarchy is reconfigured. Finally, the last step needs to be done any time the actual simulation data has changed.

Coarsen and Refinement Operators

Because many data transfer operations require data to be coarsened from finer levels or interpolated from coarser levels, the user *must* specify which coarsen or refinement operator to use should interlevel data transfer be required. SAMRAI provides the `lookupCoarsenOperator()` and `lookupRefineOperator()` methods in the `CartesianGridGeometry` class to obtain pointers to coarsen and refinement operators. Both of these methods take two arguments: a pointer to a SAMRAI `Variable` object and the name of the coarsen/refinement operator.

For example, to get a pointer to a linear refinement operator and a conservative coarsen operator for `variable`, we could use the following code:

```
// lookup refine operator
string refine_op_name = "LINEAR_REFINE";
Pointer< RefineOperator<DIM> > refine_op =
  grid_geometry->lookupRefineOperator(variable, refine_op_name);

// lookup coarsen operator
string coarsen_op_name = "CONSERVATIVE_COARSEN";
Pointer< CoarsenOperator<DIM> > coarsen_op =
  grid_geometry->lookupCoarsenOperator(variable, coarsen_op_name);
```

For most of the standard data centerings supported by SAMRAI, the following two refinement operators are provided: `LINEAR_REFINE` and `CONSERVATIVE_LINEAR_REFINE`. Only one coarsen operators is provided: `CONSERVATIVE_COARSEN`.

4.4 Imposing Boundary Conditions

While there are many ways to impose boundary conditions within the SAMRAI framework, boundary conditions are typically imposed by appropriately filling ghost cells that lie outside of the computational domain. These ghost cells are filled by invoking the user-overridden `setPhysicalBoundaryConditions()` method in the `RefinePatchStrategy` class as the last stage of the `RefineSchedule::fillData()` method.

Each `Patch` is associated with a `CartesianPatchGeometry` object which stores information about the intersection of the `Patch` boundaries with boundaries of the computational domain. To obtain a pointer to the `CartesianPatchGeometry` object associated with a `Patch patch`, we use the `getPatchGeoemtry()` method:

```
// get pointer to patch geometry
Pointer< CartesianPatchGeometry<2> > patch_geom = patch->getPatchGeometry();
```

The first step to take when imposing boundary conditions is to determine whether or not a `Patch` touches the boundary of the computational domain. For this step, we use the `getTouchesRegularBoundary()` and `getTouchesPeriodicBoundary()` methods from the `CartesianPatchGeometry` class. Once it has been determined that a `Patch` touches the boundary of the computational domain, we fill the ghost cells outside of the computational domain by using the following procedure:

1. Use the `getFaceBoundary()`, `getEdgeBoundary()`, and `getNodeBoundary()` methods to get the `BoundaryBox`es associated with the `Patch`. Note that face, edge, and node correspond to two-, one-, and zero-dimensional boundaries, respectively. For example, face boundaries are only present in 3D simulations.

2. For each `BoundaryBox`, determine the location of the boundary by using the `getLocationIndex()` method. The convention for the meaning of the location index is as follows:

   - **1D**
     - **node** (codimension 1):
       x_lo : 0
       x_hi : 1
   - **2D**
     - **edge** (codimension 1):
       x_lo: 0
       x_hi: 1
       y_lo: 2
       y_hi: 3
     - **node** (codimension 2):
       x_lo, y_lo: 0
       x_hi, y_lo: 1
       x_lo, y_hi: 2
       x_hi, y_hi: 3
   - **3D**
     - **face** (codimension 1):
       x_lo: 0
       x_hi: 1
       y_lo: 2
       y_hi: 3
       z_lo: 4
       z_hi: 5
     - **edge** (codimension 2):
       y_lo, z_lo: 0
       y_hi, z_lo: 1
       y_lo, z_hi: 2
       y_hi, z_hi: 3
       x_lo, z_lo: 4
       x_lo, z_hi: 5
       x_hi, z_lo: 6
       x_hi, z_hi: 7
       x_lo, y_lo: 8
       x_hi, y_lo: 9
       x_lo, y_hi: 10
       x_hi, y_hi: 11
     - **node** (codimension 3):
       x_lo, y_lo, z_lo: 0

x_hi, y_lo, z_lo: 1
x_lo, y_hi, z_lo: 2
x_hi, y_hi, z_lo: 3
x_lo, y_lo, z_hi: 4
x_hi, y_lo, z_hi: 5
x_lo, y_hi, z_hi: 6
x_hi, y_hi, z_hi: 7

3. Use the `getBoundaryFillBox()` method to compute the `Box` of ghost cells that needs to be filled to impose the boundary conditions.

4. Loop over the "boundary" ghost cells and set their value to impose the desired boundary conditions.

It is worth mentioning that the above procedure need not be executed within the `setPhysicalBoundaryConditions()` method and can be modified to impose boundary conditions without setting the values in ghost cells.

Chapter 5

SAMRAI Extras: I/O, Restart, Utilities and Other Fun Stuff

5.1 Using Input Files

The SAMRAI `InputManager` and `InputDatabase` classes provide a convenient way to pass parameters into a simulation via an input file. The `InputManager` class takes care of parsing the input file and loading the parameters into the `InputDatabase`:

```
// create input database (named "input_db") and parse data
// from input file with name specified by input_filename
Pointer<InputDatabase> input_db = new InputDatabase("input_db");
InputManager::getManager()->parseInputFile(input_filename, input_db);
```

Note that the `InputManager` can only be accessed via the static `getManager()` method[1].

Accessing the input parameters from the `InputDatabase` is simply a matter of using the appropriate "get" method. For example, to get a integer parameter of named "num_obstacles", we would use:

```
int num_obstacles = input_db->getInteger("num_obstacles");
```

For a full list of the accessor methods provided by `InputDatabase` (or `Database`), see the header file or the Doxygen documentation.

In addition to storing simple data types, the `InputDatabase` can hold sub-databases. In the input file, sub-databases are delimited by a set of open and close braces. For example, a sub-database named "Main" would appear in the input file as:

```
Main {
  // some input parameters
  input_parameter_1 = 1
  input_parameter_2 = 2.345

  NestedSubDatabase {
    // some more input parameters
    nested_input_parameter_1 = 6
    nested_input_parameter_2 = 7.890
  }
}
```

Notice that sub-databases can be nested. This feature can be useful for organizing input parameters. To get a pointer to a sub-database, we use the `getDatabase()` method. For instance, to access the "NestedSubDatabase", we use the following code:

[1] `InputManager` objects cannot be directly created because they are *singleton* classes.

```
// get pointer to nested sub-database
Pointer<Database> = input_db->getDatabase("NestedSubDatabase");
```

Note that `getDatabase()` returns a pointer to a `Database` object, not an `InputDatabase` object. The interface for accessing data within a `Database` object is identical to the interface for an `InputDatabase` object[2].

5.2 Check-point and Restart Capabilities

Restart capabilities in SAMRAI are supported by the `RestartManager` and `Serializable` classes. The `RestartManager` class manages access to restart files and registration of classes for check-pointing. The `Serializable` class defines the interface that any class that requires check-pointing must implement. Note that like the `InputManager` class, `RestartManager` objects can only be accessed via the static `getManager()` method[3].

Instrumenting a user-defined class with restart capabilities requires only the following simple steps:

- Override the `putToDatabase()` method in the `Serializable` class. To place data into SAMRAI `Database` objects, we use "put" methods. For example, to insert a double into the database, we use the `putDouble()` method:

  ```
  // put double into Database
  string key = "some_data_member";
  double value = 5.0;
  db->putDouble(key, value);
  ```

  For a full list of the accessor methods provided by the `Database`, see the header file or the Doxygen documentation.

- Register the class for check-pointing using the `registerRestartItem()` method in the `RestartManager` class (usually done in the constructor for the user-defined class):

  ```
  // register user-defined object for restart
  string object_name = "User-Defined Object"
  RestartManager::getManager()->registerRestartItem(object_name, this);
  ```

- Unregister the class for check-pointing in the destructor of the user-defined class:

  ```
  // unregister object as a restart item
  RestartManager::getManager()->unregisterRestartItem(object_name);
  ```

- It is convenient to write a `getFromRestart()` method that restores the state of the object using the data in the restart file. Typically, this method will begin by opening the root restart database and extracting the sub-database corresponding to the `object_name` used to register the object for check-pointing:

  ```
  // open restart file
  Pointer<Database> root_db =
    RestartManager::getManager()->getRootDatabase();

  // extract sub-databse for user-defined class
  Pointer<Database> object_db = root_db->getDatabase(object_name);
  ```

[2] In fact, the `Database` class is the general parent class for the `InputDatabase` class.
[3] `RestartManager` objects cannot be directly created because they are *singleton* classes.

If the data for a simulation variable needs to be check-pointed, it must be registered for restart by using the `registerPatchDataForRestart()` method in the `VariableDatabase` class:

```
// get pointer to VariableDatabase object
VariableDatabase<2>* variable_db = VariableDatabase<2>::getDatabase();

// register simulation variable associated with data_handle for restart
variable_db->registerPatchDataForRestart(data_handle);
```

Within the main program, the `RestartManager` manages opening, closing, and writing restart files. If a simulation is to be started from a check-point/restart file, we use the following code:

```
// Get the restart manager and root restart database.  If run is
// from restart, open the restart file.
RestartManager* restart_manager = RestartManager::getManager();
if (is_from_restart) {
  restart_manager->
     openRestartFile(restart_read_dirname, restore_num,
                     tbox::MPI::getNodes() );
}
```

Here, `restart_read_dirname` is the name of a directory containing restart files, `restore_num` is the integer label for the restart file that should be opened, and `tbox::MPI::getNodes()` gets the number of processors used for the current run. Note that SAMRAI currently only provides built-in support for restarting simulations with the same number of processors as were used when the check-point files were written. To prevent problems writing new restart files, it is a good idea to close check-point files before the main calculation:

```
restart_manager->closeRestartFile();
```

Writing restart files simply requires a call to the `writeRestartFile()` method with the name of the directory where check-point files should be written and an integer label for the restart file (using the value of a loop variable is a good way to avoid accidentally over-writing restart files):

```
restart_manager->writeRestartFile(restart_write_dirname, count);
```

5.3 Visualization

While the user is free to output simulation data in any visualization format, SAMRAI provides built-in support for writing simulation data to HDF5 files in a format that can be read by VisIt [7], a freely available visualization software developed at Lawrence Livermore National Laboratory. To output VisIt data files, we follow this simple procedure:

1. Create a `VisItDataWriter` object. The constructor takes three arguments: a string name for the data writer (this name can be arbitrary), the name of the directory where visualization files are to be written, and the number of processors whose data should be written to each VisIt file (it is safest to use 1 here).

2. Register the simulation variables whose data should be output to the visualization file using the `registerPlotQuantity()` method in the `VisItDataWriter` class. See the documentation for the `VisItDataWriter` class for the arguments of this method.

3. Output the visualization data at desired points during the simulation using the `writePlotData()` method. See the documentation for the `VisItDataWriter` class for the arguments of this method.

5.4 Design Patterns

There are a few design patterns [6] that are used throughout the SAMRAI library. It is useful to be aware of these when using SAMRAI.

Singleton Classes

Singleton classes only allow a single instantiation of the class to exist during program execution. This property is important for objects which must manage global state information for the program (*e.g.,* `RestartManager`). To enforce the singleton property, singleton classes cannot be directly created using the constructor. Instead, pointers to the singleton object may be obtained via static accessor methods, such as `RestartManager::getManager()`. Because the memory for singleton classes is not controlled by the user, they should never be explicitly deleted.

Smart Pointers

Smart pointers are designed to help manage the memory allocated for objects. Essentially, they keep a count of the number of times an object is referenced and automatically delete the object when the object is no longer referenced. For this system to work for object `obj`, it is important that only smart pointers are used to point to `obj`; otherwise, the reference count will be incorrect and the object may be accidentally deleted while it still has active references to it.

Occasionally, it may be necessary to create a smart pointer in an "unmanaged" state to avoid `unmalloc`-types of errors. This error can arise when the software design forces us to use two different smart pointers for the same object. To create an unmanaged smart pointer, we set the optional second argument in the constructor of the smart pointer to `false`:

```
Object* obj_ptr = new Object;  // bare pointer to Object
Pointer< Object > obj_smart_ptr =
  Pointer< Object >(obj_ptr, false);
```

It is worth mentioning that smart pointers should not be used for singleton classes because singleton classes manage their own memory. Using smart pointers with singleton classes can lead to errors related to premature memory deallocation.

Strategy Pattern

The strategy design pattern is used throughout the SAMRAI library, so it is good to have an understanding of its structure and purpose. The strategy design pattern is used to define a common interface for a family of algorithms so that they can be used interchangeably. For instance, the strategy pattern is useful in situations where a computation that needs to be carried out is problem specific. By using the strategy pattern, it is possible to implement a numerical algorithm in a general and flexible manner. To carry out problem specific calculations, the numerical algorithm need only invoke methods from the strategy class. The problem specific calculations themselves are implemented by the user in a concrete subclass of the strategy class. In essence, the strategy pattern allows us to decouple implementation of general numerical algorithms and problem specific numerical kernels.

Appendix A

Sample SAMRAI `main` Program

```
/*************************************************************************
 * sample_main.cc
 *************************************************************************/

// header file for SAMRAI Configuration
#include "SAMRAI_config.h"

// header files for basic SAMRAI classes
#include "tbox/Database.h"
#include "tbox/InputDatabase.h"
#include "tbox/InputManager.h"
#include "tbox/MPI.h"
#include "tbox/PIO.h"
#include "tbox/Pointer.h"
#include "tbox/RestartManager.h"
#include "tbox/SAMRAIManager.h"
#include "tbox/Utilities.h"

// header files for geometry and patch hierarchy
#include "CartesianGridGeometry.h"
#include "PatchHierarchy.h"

// header files for grid generation and load balancing
#include "BergerRigoutsos.h"
#include "GriddingAlgorithm.h"
#include "LoadBalancer.h"
#include "StandardTagAndInitialize.h"

// header files for variables and variable management
#include "CellVariable.h"
#include "VariableDatabase.h"

// header file for VisIt data writer
#include "VisItDataWriter.h"

// user-implemented classes
#include "MyApplicationClass.h"

// standard namespace
```

```
using namespace std;

// SAMRAI namespaces
using namespace SAMRAI;
using namespace appu;
using namespace geom;
using namespace hier;
using namespace mesh;
using namespace tbox;

// helper functions
void initializePatchHierarchy(
  Pointer< PatchHierarchy<2> > patch_hierarchy,
  Pointer< GriddingAlgorithm<2> > gridding_alg,
  double time);

int main(int argc, char *argv[])
{
  /*********************************************************************
   * Initialize MPI and SAMRAI, enable logging, and process command line.
   *********************************************************************/
  tbox::MPI::init(&argc, &argv);
  tbox::MPI::initialize();
  SAMRAIManager::startup();

  string input_filename;
  string restart_read_dirname;
  int restore_num = 0;

  bool is_from_restart = false;

  if ( (argc != 2) && (argc != 4) ) {
    pout << "USAGE:  " << argv[0] << " <input filename> "
         << "\n"
         << "<restart dir> <restore number> [options]\n"
         << "  options:\n"
         << "  none at this time"
         << endl;
    tbox::MPI::abort();
    return (-1);
  } else {
    input_filename = argv[1];
    if (argc == 4) {
      restart_read_dirname = argv[2];
      restore_num = atoi(argv[3]);
      is_from_restart = true;
    }
  }

  /*********************************************************************
   * Create input database and parse all data in input file.
   *********************************************************************/
  Pointer<Database> input_db = new InputDatabase("input_db");
  InputManager::getManager()->parseInputFile(input_filename, input_db);
```

```
// Read in the input from the "Main" section of the input database.
Pointer<Database> main_db = input_db->getDatabase("Main");

/*********************************************************************
 * Set base for all name strings in program
 *********************************************************************/
string base_name = "my_SAMRAI_program";

/*********************************************************************
 * Set up restart
 *********************************************************************/
// from restart, open the restart file.
RestartManager* restart_manager = RestartManager::getManager();
if (is_from_restart) {
  restart_manager->
     openRestartFile(restart_read_dirname, restore_num,
                     tbox::MPI::getNodes() );
}

int restart_interval = main_db->getInteger("restart_interval");
string restart_write_dirname = base_name + ".restart";
const bool write_restart = (restart_interval > 0);

// Get the restart manager and root restart database.  If run is
/*********************************************************************
 * Set up logging
 *********************************************************************/
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) {
   PIO::logAllNodes(log_file_name);
} else {
   PIO::logOnlyNodeZero(log_file_name);
}

// log the command-line args
plog << "input_filename = " << input_filename << endl;
plog << "restart_read_dirname = " << restart_read_dirname << endl;
plog << "restore_num = " << restore_num << endl;

/*********************************************************************
 * Create objects required for computation
 *********************************************************************/
// create geometry object
Pointer< CartesianGridGeometry<2> > grid_geometry =
  new CartesianGridGeometry<2>("CartesianGeometry",
                               input_db->getDatabase("CartesianGeometry"));

// create patch hierarchy object
Pointer< PatchHierarchy<2> > patch_hierarchy =
  new PatchHierarchy<2>("PatchHierarchy", grid_geometry);
```

```
// create application specific data objects
MyApplicationClass* my_application_obj= new MyApplicationClass;

// create grid management objects
Pointer< StandardTagAndInitialize<2> > standard_tag_and_init_obj =
  new StandardTagAndInitialize<2>(
    "StandardTagAndInitialize", my_application_obj,
    input_db->getDatabase("StandardTagAndInitialize"));
Pointer< BergerRigoutsos<2> > box_generator = new BergerRigoutsos<2>();
Pointer< LoadBalancer<2> > load_balancer =
  new LoadBalancer<2>("LoadBalancer",
                      input_db->getDatabase("LoadBalancer"));
Pointer< GriddingAlgorithm<2> > gridding_alg = new GriddingAlgorithm<2>(
  "GriddingAlgorithm",
  input_db->getDatabase("GriddingAlgorithm"),
  standard_tag_and_init_obj,
  box_generator,
  load_balancer);

/*********************************************************************
 * Set up visualization
 *********************************************************************/
// set up visualization write interval
int viz_write_interval = main_db->getInteger("viz_write_interval");
const bool write_viz = (viz_write_interval > 0);

// set up VisIt parameters
int visit_number_procs_per_file =
  main_db->getInteger("visit_number_procs_per_file");

// get PatchData handles from application specific classes
// for visualization purposes
int data_handle = my_application_obj->getPatchDataHandle();

// creqate VisIt data writer object
Pointer<VisItDataWriter<2> > visit_data_writer = 0;
string visit_data_dirname = base_name + ".visit";
visit_data_writer = new VisItDataWriter<2>("VisIt Writer",
                                           visit_data_dirname,
                                           visit_number_procs_per_file);

// register level set functions and velocity fields for plotting
visit_data_writer->registerPlotQuantity(
  "data", "SCALAR", data_handle, 0, 1.0, "CELL");

/*********************************************************************
 * initialize data on PatchHierarchy
 *********************************************************************/
double time = 0.0;
initializePatchHierarchy(patch_hierarchy, gridding_alg, time);

/*********************************************************************
 * Close restart file before starting main time-stepping loop.
 *********************************************************************/
```

```
  restart_manager->closeRestartFile();

  /*****************************************************************
   * Main simulation loop
   *****************************************************************/
  int count = 0;
  int max_num_time_steps = main_db->getInteger("max_num_time_steps");
  while ( count < max_num_time_steps ) {

    // do simulation work . . .

    // output restart data
    if ( write_restart && (0 == count%restart_interval) ) {
       restart_manager->writeRestartFile(restart_write_dirname, count);
     }

    // output visualization data
    if ( write_viz && (0 == count%viz_write_interval) ) {
      visit_data_writer->writePlotData(patch_hierarchy, count, count);
    }

    // update count
    count++;
  }

  /*****************************************************************
   * Explicitly free memory used for objects that are NOT managed
   * by smart-pointers
   *****************************************************************/
  delete my_application_obj;

  /*****************************************************************
   * Shutdown SAMRAI and MPI
   *****************************************************************/
  SAMRAIManager::shutdown();
  tbox::MPI::finalize();

  return(0);
}

void initializePatchHierarchy(
  Pointer< PatchHierarchy<2> > patch_hierarchy,
  Pointer< GriddingAlgorithm<2> > gridding_alg,
  double time)
{
  // create and initialize coarsest level of PatchHierarchy
  gridding_alg->makeCoarsestLevel(patch_hierarchy, time);

  // create and initialize all finer levels
  bool initial_time = true;
  int tag_buffer = 0;
  for (int level_num = 0;
       gridding_alg->levelCanBeRefined(level_num);
       level_num++) {
```

```
    gridding_alg->makeFinerLevel(patch_hierarchy, time,
                                 initial_time, tag_buffer);
  }
}
```

Appendix B

Sample Subclass of SAMRAI StandardTagAndInitStrategy Class

```
/*************************************************************************
 * MyApplicationClass.h
 *************************************************************************/

#ifndef included_MyApplicationClass
#define included_MyApplicationClass

#include "SAMRAI_config.h"
#include "BoxArray.h"
#include "IntVector.h"
#include "BasePatchHierarchy.h"
#include "BasePatchLevel.h"
#include "StandardTagAndInitStrategy.h"
#include "tbox/Array.h"
#include "tbox/Pointer.h"

using namespace SAMRAI;
using namespace hier;
using namespace mesh;
using namespace tbox;

/*!
 * Class MyApplicationClass implements a concrete subclass of
 * TagAndInitializeStrategy (for DIM equals 2) that defines very
 * simple versions the pure virtual methods in that class for
 * illustration purposes.
 */

class MyApplicationClass:
  public StandardTagAndInitStrategy<2>
{
public:
  /*!
   * Sets up data variables.
   */
  MyApplicationClass();
```

```
/*!
 * Empty destructor for MyApplicationClass.
 */
virtual ~MyApplicationClass();

/*!
 * Returns the PatchData handle for the cell-centered demonstration data.
 */
virtual int getPatchDataHandle();

/*!
 * Initialize data on a new level added to the PatchHierarchy by the
 * GriddingAlgorithm.  The value of the data on the new level is set
 * to be a constant value equal to the level number.
 *
 * Arguments:
 * - hierarchy:       pointer to PatchHierarchy containing new level
 * - level_number:    level number of new level
 * - init_data_time:  current simulation time
 * - can_be_refined:  true if the new level is the finest refinement level
 *                    allowed in the hierarchy; false otherwise.
 * - initial_time:    flag indicating whether init_data_time is the
 *                    initial time in the simulation.  If true, the
 *                    level should be initialized with initial simulation
 *                    values.  Otherwise, it is assumed that
 *                    init_data_time is at some point after the start
 *                    of the simulation.  In this case, the
 *                    initialization of the new level is handled
 *                    differently.
 * - old_level:       pointer to old level that resided in the
 *                    hierarchy before the new level was introduced
 *                    (default = NULL)
 * - allocate_data:   flag indicating whether the memory for data on
 *                    the new level needs to be allocated before
 *                    the data is initialized.  true indicates that
 *                    memory needs to be allocated; falst indicates
 *                    that the memory is already allocated
 *                    (default = true)
 *
 * Return value:      none
 *
 */
virtual void
initializeLevelData(const Pointer< BasePatchHierarchy<2> > hierarchy,
                    const int level_number,
                    const double init_data_time,
                    const bool can_be_refined,
                    const bool initial_time,
                    const Pointer< BasePatchLevel<2> > old_level =
                          Pointer< BasePatchLevel<2> >(NULL),
                    const bool allocate_data = true);

/*!
```

```
   * After hierarchy levels have changed and data has been initialized on
   * the new levels, this routine resets any information needed by the
   * solution method that is particular to the hierarchy configuration.
   *
   * For this class, the resetHierarchyConfiguration() method is empty
   * because this class does not actually carry out any calculations
   * that depend on the hierarchy configuration.
   *
   * Arguments:
   * - hierarchy:       pointer to PatchHierarchy to be reset
   * - coarsest_level:  coarsest level in hierarchy that has changed
   * - finest_level:    finest level in hierarchy that has changed
   *
   * Return Value:      none
   *
   */
  virtual void
  resetHierarchyConfiguration(
    const Pointer< BasePatchHierarchy<2> > hierarchy,
    const int coarsest_level,
    const int finest_level);


private:

  // PatchData handle for data
  int d_patch_data_handle;

  // The following are not implemented:
  MyApplicationClass( const MyApplicationClass&);
  void operator=(const MyApplicationClass&);

};

#endif

#ifdef INCLUDE_TEMPLATE_IMPLEMENTATION
#include "MyApplicationClass.cc"
#endif


/*************************************************************************
 * MyApplicationClass.cc
 *************************************************************************/

#ifndef included_MyApplicationClass_cc
#define included_MyApplicationClass_cc

#include "CellData.h"
#include "CellVariable.h"
#include "VariableContext.h"
#include "VariableDatabase.h"
#include "tbox/Utilities.h"

#include "MyApplicationClass.h"
```

```
using namespace SAMRAI;
using namespace pdat;

MyApplicationClass::MyApplicationClass()
{
  // get pointer to VariableDatabase
  VariableDatabase<2>* variable_db = VariableDatabase<2>::getDatabase();

  // create 2D CellVariable with type "double" and depth 1
  Pointer< CellVariable<2,double> > data_variable =
    new CellVariable<2,double>("data_variable",1);

  // get context for variable from VariableDatabase
  Pointer< VariableContext > variable_context =
    variable_db->getContext("CURRENT");

  // set the number of ghostcells for data to zero
  IntVector<2> zero_ghosts(0);

  // register variable-context pair (with specified ghostcell width)
  // with the VariableDatabase to get PatchData handle, which is used
  // to access data
  d_patch_data_handle = variable_db->registerVariableAndContext(
    data_variable, variable_context, zero_ghosts);

}

MyApplicationClass::~MyApplicationClass()
{
}

int MyApplicationClass::getPatchDataHandle()
{
  return d_patch_data_handle;
}

void MyApplicationClass::initializeLevelData(
  const Pointer< BasePatchHierarchy<2> > base_hierarchy,
  const int level_number,
  const double init_data_time,
  const bool can_be_refined,
  const bool initial_time,
  const Pointer< BasePatchLevel<2> > old_level,
  const bool allocate_data)
{
  // get pointer to new level
  Pointer<PatchHierarchy<2> > hierarchy = base_hierarchy;
  Pointer<PatchLevel<2> > level = hierarchy->getPatchLevel(level_number);

  // allocate memory for data (if required)
  if (allocate_data) {
    level->allocatePatchData(d_patch_data_handle);
  }
```

```
  // loop through the Patches on the new level and set all of the
  // data to be equal to the level number
  for (PatchLevel<2>::Iterator p(level); p; p++) {
    Pointer<Patch<2> > patch = level->getPatch(p());

    // get PatchData
    Pointer< CellData<2,double> > patch_data =
      patch->getPatchData(d_patch_data_handle);
    patch_data->fillAll(level_number);
  }

}

void MyApplicationClass::resetHierarchyConfiguration(
  const Pointer< BasePatchHierarchy<2> > hierarchy,
  const int coarsest_level,
  const int finest_level)
{
  // FOR THIS CLASS, THIS METHOD DOES NOTHING.
}

#endif
```

Appendix C

Migrating From A Uniform Grid Code To SAMRAI Code

1. Decompose the numerical algorithm into small numerical kernels that operate on a single patch. It is convenient for the numerical kernels to take pointers to all of the required data and all required parameters as arguments.

2. Implement a concrete subclass of the `StandardTagAndInitStrategy` (Section 4.1). class to manage creation of simulation variables, communication of data between `Patches` (and processors), and initialization of simulation data.

   (a) Create a simulation variable (Section 4.2) for each piece of simulation data in the serial code. It may also be necessary to create simulation data for temporary/scratch data.

   (b) Construct communication schedules for data that needs to be transferred between `Patches` (Section 4.3).

   (c) Override the `initializeLevelData()` method to initialize simulation data using the SAMRAI idiom described in Section 3.4.

3. Write SAMRAI main program that will act as the framework for constructing the SAMR grid (with only one `PatchLevel`) and managing the numerical algorithm at a high-level (see Section 4.1).

Bibliography

[1] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[2] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.

[3] S. McCormick and J. Thomas. The fast adaptive composite grid (fac) method for elliptic equations. *Mathematics of Computation*, 46:439–456, 1986.

[4] R. E. Ewing, R. D. Lazarov, and P. S. Vassilevski. Local refinement techniques for elliptic problems on cell-centered grids i. error analysis. *Mathematics of Computation*, 56:437–461, 1991.

[5] R. E. Ewing, R. D. Lazarov, and P. S. Vassilevski. Local refinement techniques for elliptic problems on cell-centered grids ii. two-grid iterative methods. *J. Numerical Linear Algebra and Applications*, 1:337–368, 1994.

[6] E. Gamma, R. Johnson R. Helm, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[7] VisIt web site. http://www.llnl.gov/visit/.

[8] HDF5 web site. http://hdf.ncsa.uiuc.edu/HDF5/.