SAMRAI User's Guide

The *SAMRAI* Team

Contents

1 Introduction

1.1 Purpose of This Document

The main purpose of this user's guide is to describe the *SAMRAI* library and provide an overview of the main capabilities of *SAMRAI*. This is not a reference manual fully documenting all of the classes. The documentation for each class is included in the class header file and in the Doxygen documentation provided with the library and should be referenced when more detailed API information in needed.

This document is currently very sparse and is barely a placeholder at the moment. The goal is to document additions to the library in this document and fill in older parts as resources permit.

1.2 *SAMRAI* Motivation and Design Goals

The primary mission of the *SAMRAI* project is to provide a framework to enable the development of structured adaptive mesh refinement (SAMR) applications. *SAMRAI* is an object oriented C++ library specifically targeted at parallel machines and uses MPI for it's parallel programming model. *SAMRAI* is used within LLNL for the exploration of SAMR concepts.

The primary design goals of the *SAMRAI* framework are:

- Reduce development time for new SAMR applications by leveraging existing an existing framework.
- Providing high-level abstractions that manage the complexity of data dependencies on an SAMR grid.
- Support the development of large scale parallel SAMR applications.
- Enable computational scientists to explore new applications and algorithms through the composition and extension of existing SAMR software components.

SAMRAI is a large package to learn. As with any sizable object oriented framework, there are abstractions being used which can make it seem like many layers of complexity are being added with little benefit. In some cases this is a result of an intentional focus on flexibility and extensibility over ease of use. The additional complexity is not required for the simple problems however it does enable the framework to be used when simple isn't sufficient.

Some knowledge of design patterns is beneficial when trying to understand *SAMRAI*. No attempt is made to explain the patterns used as better explanations already exist in the plethora of design patterns books and web pages. In particular if Singleton, Factory, and Strategy patterns are unfamiliar, a quick web search will help understand the *SAMRAI* documentation.

2 Restart

2.1 Overview

This chapter covers the restart API in *SAMRAI*. The restart API is used to checkpoint the state of a *SAMRAI* application and restart from that checkpoint. The obvious use is the case of machine failure during a long run.

SAMRAI supports several methods for creation of restart files to enable control over where the *SAMRAI* restart information is stored. All of the options make this section seem way more complicated than it should. However for most applications there are really just two basic questions an application developer needs to answer to figure out which of the API's to use:

- Is *SAMRAI* in control of opening and closing the restart file?
- Is the restart information stored in a HDF, Silo, or other file format?

For applications starting from scratch or applications that do not have an existing restart capability, *SAMRAI* can handle creating the restart files and writing the data to them. For applications that have an existing HDF or Silo restart file, *SAMRAI* can write it's data to that existing file.

In the extreme case, an application with a different file format or mechanism for doing restart can parse the *SAMRAI* restart information from a simple in-memory database and write the information to whatever format is desired.

The sections below provide an overview of the principle classes involved and provide examples of how they would be used in several scenarios.

2.2 Overview of Classes used in Restart

The principle classes used in restart are:

RestartManager Manages the restart process. Objects that have restart data are registered with the restart manager and the restart manager informs the registered objects when they need to dump data.

Serializable The interface classes need to implement in order to be restartable.

Database The interface for interacting with a restart file. Used to put/get data to/from a restart file. Implementations are provided for HDF and Silo files.

Other classes that play a role in restart are:

DatabaseFactory Factory used by the `RestartManager` to create databases when the `RestartManager` is responsible for creating the restart files.

2.3 Checkpoint and Restart Process

The general process for creating a restart file in *SAMRAI* begins with objects registering themselves with the `RestartManager` in the constructor for the class. Classes being registered must implement the `Serializable` interface. The following code snippet shows how the registration is done:

```
tbox::RestartManager::getManager() -> registerRestartItem(object_name, this);
```

`object_name` should be a unique name that identifies the object. This name will be used to identify the information in the restart file so that it can be retrieved. Note this name should generally be unique to each object not class so that the data for multiple objects can be retrieved. Singletons are an obvious exception to this rule.

At some time-step (often specified as some time interval) the application informs the `RestartManager` that a restart file should be saved. This is done via the `writeRestartFile` method:

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

`restart_write_dirname` is a string containing the path to a directory that will be used to store the restart files. `iteration_num` is an integer used to specify the dump number; this is often the iteration count for the time loop in the application. *SAMRAI* will create a set of HDF files containing the restart information (one file per processor).

Internally `RestartManager` iterates over all the registered objects and calls the `PutToDatabase` method (from the `Serializable` interface) and the objects make "put" calls (e.g. putDoubleArray()) on the `Database` object. `PutToDatabase` for a class will look similar to this example from the `LinAdv` code:

```
void LinAdv::putToDatabase( tbox::Pointer<tbox::Database> db)
{
   db->putInteger("LINADV_VERSION",LINADV_VERSION);

   db->putDoubleArray("d_advection_velocity",d_advection_velocity,NDIM);

   db->putInteger("d_godunov_order", d_godunov_order);
   db->putString("d_corner_transport", d_corner_transport);
   db->putIntegerArray("d_nghosts", (int*)d_nghosts, NDIM);
   db->putIntegerArray("d_fluxghosts", (int*)d_fluxghosts, NDIM);

....
```

Note `LINADV_VERSION` is used track version information to prevent reading an old restart file into a newer implementation.

The *SAMRAI* restart process is fairly easy as well but differs in some details from other restart mechanisms which may cause some confusion. For restart the existing restart file is opened as a Database object and this Database is read in each of the constructors. The constructors use this data along with the input file to initialize state. *SAMRAI* tries to preserve the notion that object creation is object initialization so restart is tied with object creation. This is the reason why there is no symmetric `getFromDatabase` method in the `Serializable` interface as one might expect. The following code snippet is from the `LinAdv` example code and shows how the `RestartManager` is checked to see if restart is being done and if so the state is restored using the `getFromRestart` method. `getFromRestart` restores state from the database using "get" calls on the database (just invert the "put" operations done in the `putToDatabase` method).

```
/*
 * Initialize object with data read from given input/restart databases.
 */
bool is_from_restart = tbox::RestartManager::getManager()->isFromRestart();
if (is_from_restart){
   getFromRestart();
}
getFromInput(input_db, is_from_restart);
```

SAMRAI allows changing of some values from the input file on restart. A restart need not be exactly the same state as was originally setup if the user has edited the input file. It is the responsibility of each *SAMRAI* class to determine if information should come from the restart file or the input file. Each class constructor needs ensure input changes are consistent with the restart state. This capability should obviously be used with some caution.

Note that *SAMRAI* a restart file is NOT sufficient to restore state, the input file is needed. *SAMRAI* does not replicate the information in the input file to the restart file. This is different from other restart mechanism that store all of the required state in the restart file.

2.4 Restart when *SAMRAI* is in total control

If you do not have any existing restart capabilities the easiest way to do restart is to let *SAMRAI* control the file for restart and the file format.

Dumping data to a restart file is very simple; simply call to the `writeRestartFile` method of the `RestartManager` singleton as shown in this sample code:

```
if ( (iteration_num % restart_interval) == 0 ) {
       tbox::RestartManager::getManager() -> writeRestartFile(
           restart_write_dirname,
           iteration_num);
}
```

Restoring is done with a call to the `openRestartFile` method before constructing *SAMRAI* objects. As noted previously the *SAMRAI* constructors will load state from the restart file if one has been opened by the `RestartManager`.

```
tbox::RestartManager* restart_manager = tbox::RestartManager::getManager();

restart_manager -> openRestartFile(restart_read_dirname,
                                   restore_num,
                                   tbox::SAMRAI_MPI::getNodes() );

/*
 * SAMRAI constructors will pick up state from the restart file if
 * one was used.
 */
```

Note that the `openRestartFile` must be done before *SAMRAI* objects are created.

2.5 Restart when the application manages file creation

If you have existing restart capabilities based on HDF or Silo and want *SAMRAI* to save/restore state from that existing file, the database API's have the capability to "attach" to an existing file. This avoids having restart state spread over multiple files.

The basic process is for the application to create or open the HDF/Silo file, instantiate one of Database objects (`HDFDatabase` or `SiloDatabase`) and then use the `attachToHDF` or `attachToSilo` method to tell *SAMRAI* where (group for HDF or path for Silo) in the file *SAMRAI* should write/read it's information. The `RestartManager` method setRootDatabase is used to force the `RestartManager` to use that Database instead of opening one of it's own as in the previous section.

This code sample shows the calls for saving state to an application managed Silo file. HDF would be similar; replacing the `SiloDatabase` with an `HDFDatabase`.

```
// Create a SiloDatabase
tbox::Pointer<tbox::SiloDatabase> database =
                               new tbox::SiloDatabase("SAMRAI Restart");

// The application opens a Silo file
std::string name = "./restart." +
        tbox::Utilities::processorToString(tbox::SAMRAI_MPI::getRank()) +
        ".silo";

DBfile *silo_file = DBCreate(name.c_str(), DB_CLOBBER, DB_LOCAL,
```

```
                                NULL, DB_PDB);

// Attach the database to application file at the
// specified Silo path inside the file
database -> attachToFile(silo_file, "/SAMRAI");

// Tell restart manager to use the Silo database
restart_manager -> setRootDatabase(database);

// Write the current SAMRAI state.
// This will store the information to the Silo file via
// the Silo database.
restart_manager->writeRestartToDatabase();

// Other writing to the Silo file by the appliation.

// Close the database and the Silo file
database -> close();
DBClose(silo_file);
```

This code sample shows the calls for restoring state from an application managed Silo file. HDF would be similar; replacing the `SiloDatabase` with an `HDFDatabase`.

```
// Create the Silo database instance
database = new tbox::SiloDatabase("SAMRAI Restart");

// Open the Silo file that has the restart information
silo_file = DBOpen(name.c_str(), DB_UNKNOWN, DB_READ);

// Attach the Silo file to the SiloDatabase
database -> attachToFile(silo_file, "/SAMRAI");

// Tell restart manager to use the Silo database
restart_manager -> setRootDatabase(database);

// Restore state

// Close the database and Silo file
database -> close();
DBClose(silo_file);
```

2.6 Restart when the application manages file format

If you have existing restart files which are not HDF or Silo and/or you want total control over the scheme used to store the *SAMRAI* data there are two options.

The first method is to create your own class that implements the `Database` API. The existing Silo or HDF implementations should serve as a decent starting point for doing this. For example, implementing a `Database` for reading/writing XML should be relatively simple. The new user supplied `Database` would then supplied to the `RestartManager setRootDatbase` method in a similar manner to the previous section.

The second option is to use the `MemoryDatabase` and is not recommended if it can be avoided. In this case a `Database` that resides in memory is used as a temporary location for the restart data. To create

a checkpoint file you would first dump the state to the the `MemoryDatabase`. The in memory database would then be traversed and the data can be stored in whatever format you wish. Traversal can be done using the `getAllKeys` and `getArrayType` methods to walk the database hierarchy. Restart is simply the inverse, create a `MemoryDatabase` and fill it in with data from your file restoring the original structure. The `setRootDatabase` is then used to provide *SAMRAI* with restart information. This method is more complicated than the other mechanisms but allows you total control over the structure of data in the restart file. Note that this method requires additional memory since data is being replicated temporarily in an instance of the `MemoryDatabase`.

2.7 DatabaseFactory and the RestartManager

When the `RestartManager` is in control of file creation it uses a `DatabaseFactory` to create new database objects. By default this is set to the `HDFDatabaseFactory`. The `setDatabaseFactory` method can be used to override this default behavior if a different file format is desired. The `SiloDatabaseFactor` will create Silo files or if you have created your own `Database` implementation you can also create a factory for it to allow the `RestartManager` control over file creation. Controlling file creation using a `DatabaseFactor` and the `RestartManager` is entirely optional; it is simply a convenience so you don't have to manage the creation process in the application code main time loop.