Nuclear Engineering Laboratory
M O N T E C U C C O L I N O

memorandum

Nuclear Reactor Physics
Department of Energy and Nuclear Engineering
and of Environmental Control (DIENCA)
University of Bologna

Doc. ID: LIN-M05.2007
Subject: SAMRAI introduction
Date: July 01, 2007

Author: Giacomo Grasso
Phone: +39-051-6441708
e-mail: giacomo.grasso@mail.ing.unibo.it

A gentle introduction to SAMRAI

The Structured Adaptive Mesh Refinement (SAMR) is a numerical technique for the solution of spatial dependent problems via dynamic creation of a multi scale hierarchy of structured grids. This approach leads to a faster and finer solution, and to the damping of numerical noise at different wave lengths.

The SAMRAI framework[1]is an extremly powerful, flexible tool for SAMR extension of scientific codes. Compared with other tools freely distributed in the scientific community, it is further the most general and the richest of functionalities[2,3]. The main difference lies indeed in the category SAMRAI belongs to: despite the classical "library" definition, it has been developed as a "framework", by means of design patterns[4]. This means that it integrates the existing code within a pre-defined structure which inherits automatic methods for mesh generation, data communication, time integration and problem solution; on the contrary, a library (such as ParaMESH[5]) is a collection of static tools which must be included in the existing code to extent its functionalities.

A framework is a "set of cooperating classes that make up a reusable design for a specific class of software"[4]: this means that a pre-defined set of objects builds up an infrastructure which dictates the architecture of the user application; it defines indeed

- the overall structure;
- its partitioning into classes and objects;
- the key responsibilities;
- how classes and objects collaborate;
- the thread of control.

---

[1]R. D. Hornung and S. R. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[2]R. Hornung. An introduction to the SAMRAI framework. Technical Report UCRL-PRES-202207, Center for Applied Scientific Computing - Lawrence Livermore National Laboratory, 2006.

[3]K. T. Chu. A SAMRAI Primer. Personal note, 2007.

[4]E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: elements of reusable Object-Oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[5]P. MacNeice *et al.* PARAMESH: A parallel adaptive mesh refinement community toolkit. *Comp. Phys. Comm.*, 126:330-54, 2000.

Such an infrastructure becomes important since frameworks emphasize design reuse over code reuse, letting the tool be flexible to a wide family of conceptually similar problems rather than to a specific class of applications.

Main features

Besides the conceptual differences, the SAMRAI framework includes a wide variety of pre-defined methods for mesh hierarchy patches dynamic generation and management, data-patch dynamical assignment, communication and interpolation and load balancing.

The key concept is that patches are the fundamental objects for both the problem description and its parallelization:

- hierarchy of patches in levels is automatically managed by the framework;
- a user-defined criterion is automatically used to flag cells for refinement and sub-patches creation;
- the problem geometry itself can be dynamically chosen at compile time without any further effort, since it is implemented as a patch property;
- grid data are integral with the patch;
- communication between intra- and inter-level patches is implemented as a patch method itself;
- patches are automatically spanned between processors according to a user-defined load balance criterion.

Moreover, a series of secondary - but very practical - features have been included in SAMRAI making the latter an almost complete tool:

- array data memorization by column major ordering (fortran-like) to incite the inclusion of fortran external subroutines in order to squeeze the performances by means of the latter more efficient language with respect to the C++ numerics;
- practical input file management for both SAMRAI settings and user defined simulation parameters;
- check point and restart capabilities;
- provided built-in support for simulation data writing to HDF5 file format, for direct use in VisIt;
- inclusion of Vizamrai, a visualization and analysis tool for SAMR data;
- timing functions by reporting exclusive time, calling tree hierarchy, estimated statistics on self induced overhead; output formatting for Vampir traces visualizer (non-free and high overhead) and Tau application profiler (free and small overhead).

Framework use: an example

As an example of use of the SAMRAI framework, a sample Poisson problem will be modeled in 2D, on a dynamically regridding mesh. Coherently with the SAMRAI phylosophy of external fortran numerical kernels, the field source assignment and the tagging for regridment functions are declared as externals.

The Main.C file

The `Main.C` file collects the initializations of the SAMRAI framework modules and environments (i.e. the MPI and the input/output ones),

```
#include SAMRAI_config.h

#include BergerRigoutsos.h
#include "CartesianGridGeometry.h"
#include "tbox/Database.h"
#include "GriddingAlgorithm.h"
#include "tbox/InputDatabase.h"
#include "tbox/InputManager.h"
#include "LoadBalancer.h"
#include "PatchHierarchy.h"
#include "tbox/PIO.h"
#include "tbox/Pointer.h"
#include "tbox/Array.h"
#include "tbox/SAMRAIManager.h"
#include "StandardTagAndInitialize.h"
#include "tbox/MPI.h"
#include "tbox/TimerManager.h"
#include "tbox/Utilities.h"

#include FACPoisson.h

using namespace SAMRAI;

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

    tbox::Pointer<tbox::Database> input_db =
      new tbox::InputDatabase(input_db);
    tbox::InputManager::getManager()->parseInputFile(
        string input_filename,
        input_db
    );
```

as well as the initialization of the main objects of the SAMR program, i.e. the geometry, the hierarchy, the problem object (allocating the associated array data), the regridding and time integration algorithms and the load balancing criterion.

```
tbox::Pointer<geom::CartesianGridGeometry<NDIM> > grid_geometry =
  new geom::CartesianGridGeometry<NDIM>(
      base_name+"CartesianGeometry",
      input_db>getDatabase("CartesianGeometry")
);
tbox::Pointer<hier::PatchHierarchy<NDIM> > patch_hierarchy =
  new hier::PatchHierarchy<NDIM>(
      base_name+"::PatchHierarchy",
      grid_geometry
);
FACPoisson fac_poisson(
    base_name+"::FACPoisson",
    input_db->isDatabase("FACPoisson") ?
      input_db->getDatabase("FACPoisson") :
        tbox::Pointer<tbox::Database>(NULL)
);
tbox::Pointer<mesh::StandardTagAndInitialize<NDIM> >
  tag_and_initializer = new mesh::StandardTagAndInitialize<NDIM>(
      "CellTaggingMethod",
      tbox::Pointer<mesh::StandardTagAndInitStrategy<NDIM> >                    (&fac_poisson,false),
      input_db->getDatabase("StandardTagAndInitialize")
);
tbox::Pointer<mesh::BergerRigoutsos<NDIM> > box_generator =
  new mesh::BergerRigoutsos<NDIM>();
tbox::Pointer<mesh::LoadBalancer<NDIM> > load_balancer =
  new mesh::LoadBalancer<NDIM>(
      "load balancer",
      tbox::Pointer<tbox::Database>()
);
tbox::Pointer< mesh::GriddingAlgorithm<NDIM> > gridding_algorithm;
gridding_algorithm = new mesh::GriddingAlgorithm<NDIM>(
    "Gridding Algorithm",
    input_db->getDatabase("GriddingAlgorithm"),
    tag_and_initializer,
    box_generator,
    load_balancer
);
```

Finally, the Main function builds up the proper mesh, initializes the plotter function and calls the problem solver before destroying the initialized objects and finalizing the environments.

```
gridding_algorithm->makeCoarsestLevel(patch_hierarchy,0.0);
bool done=false;
for (int lnum = 0;
gridding_algorithm->levelCanBeRefined(lnum) && !done; lnum++) {
    gridding_algorithm->makeFinerLevel(
        patch_hierarchy,
        0.0,
        true,
        0
    );
    done = !(patch_hierarchy->finerLevelExists(lnum));
}
```

```
    /*
     * Define and initialize plotter for the created hierarchy
    ...
     */

    fac_poisson.solvePoisson();

    /*
     * Plot data
    ...
     */

    tbox::TimerManager::getManager()->print(tbox::plog);

    tbox::SAMRAIManager::shutdown();
    tbox::MPI::finalize();
    return(0);
}
```

The problem class FACPoisson.C file

The `FACPoisson.C` file (and the related `FACPoisson.h` one, omitted) defines the problem specific class, defining the variables needed for the simulation, the methods required for its solution and the external methods for numerical kernels.

```
#include FACPoisson.h

#include IntVector.h
#include "CartesianGridGeometry.h"
#include "CartesianPatchGeometry.h"
#include "SimpleCellRobinBcCoefs.h"
#include "CellData.h"
#include "HierarchyCellDataOpsReal.h"
#include "SideData.h"
#include "PoissonSpecifications.h"
#include "tbox/Utilities.h"
#include "Variable.h"
#include "VariableDatabase.h"

extern C {
    // Fortran user subroutines list
    void setrhs_( ... );
}

namespace SAMRAI {
    FACPoisson::FACPoisson fac_poisson(
        const string &object_name,
        tbox::Pointer<tbox::Database> database
    ):
      d_object_name(object_name),
      d_hierarchy(NULL),
      d_poisson_fac_solver(
          object_name+::poisson_hypre,
```

```
        ( !database.isNull() && database->isDatabase(fac_solver) )?
          database->getDatabase(fac_solver):
          tbox::Pointer<tbox::Database>(NULL)
    ),
    d_bc_coefs(
        object_name+"::bc_coefs",
        ( !database.isNull() && database->isDatabase("bc_coefs") )?
          database->getDatabase("bc_coefs"):
          tbox::Pointer<tbox::Database>(NULL)
    ),
    d_context()
{
    hier::VariableDatabase<NDIM> *vdb =
      hier::VariableDatabase<NDIM>::getDatabase();

    d_context = vdb->getContext(d_object_name + :Context);

    tbox::Pointer<pdat::CellVariable<NDIM,double> > comp_soln =
      new pdat::CellVariable<NDIM,double>(
          object_name+:computed solution,
          1  // depth
    );
    d_comp_soln_id = vdb->registerVariableAndContext(
        comp_soln,
        d_context,
        hier::IntVector<NDIM>(1)  // ghost cell width
    );

    // ... same for rhs_variable and d_rhs_id (width=0)

    d_poisson_fac_solver.setBcObjet( &d_bc_coefs );

    return;
}
```

It is interesting to notice that the data arrays are implicitly defined by the geometry and the mesh definition: the user is not asked to specify the size of each array but only its depth (the number of data for every grid location) and the kind of data (cell centered, side or edge located, etc.). The variables are then registered in an automatically managed database for a transparent communication and interpolation of data between patches in an efficient way. On every new patch the variables can be recalled by querying the database with an unique identifier proper of each variable.

```
void FACPoisson::initializeLevelData(
    const tbox::Pointer<hier::BasePatchHierarchy<NDIM> > hierarchy_,
    const int level_number,
    const double init_data_time,
    const bool can_be_refined,
    const bool initial_time,
    const tbox::Pointer<hier::BasePatchLevel<NDIM> > old_level,
    const bool allocate_data
)
{
    tbox::Pointer<hier::PatchHierarchy<NDIM> > hierarchy=hierarchy_;
```

```
        tbox::Pointer<geom::CartesianGridGeometry<NDIM> > grid_geom =
          hierarchy->getGridGeometry();

        tbox::Pointer<hier::PatchLevel<NDIM> > level =
          hierarchy->getPatchLevel(level_number);

        if( allocate_data ) {
            level->allocatePatchData( d_comp_soln_id );
            level->allocatePatchData( d_rhs_id );
        }
        hier::PatchLevel<NDIM> ::Iterator pi(*level);

        for( pi.initialize(*level); pi; pi++) {
            const int pn = *pi;
            tbox::Pointer<hier::Patch<NDIM> > patch =
              level->getPatch(pn);
            hier::Box<NDIM> pbox = patch->getBox();
            tbox::Pointer<geom::CartesianPatchGeometry<NDIM> > p_geom =
              patch->getPatchGeometry();

            tbox::Pointer<pdat::CellData<NDIM,double> > rhs_data =
              patch->getPatchData( d_rhs_id );

            setrhs_(
                ...
                rhs_data->getPointer(),
                ...
            );

        }

    }
```

In the last section of the `FACPoisson.C` file is defined the `solvePoisson` method, for the explicit solution of the Poisson problem implicitly defined in the mesh associated to the patches hierarchy. The solver used is the FAC one, implemented in the `tools` namespace of the SAMRAI framework: it is a fast multi-level solver developed at CASC for the solution of Poisson-like problems in the form

$$C(x) + \nabla \cdot D(x) \nabla u(x) = f(x) \tag{1}$$

for the field $u(x)$, where

$C(x)$ is a scalar field;

$D(x)$ is the diffusion coefficient;

$f(x)$ is the source term.

The definition of both the scalar field and the diffusion coefficient can be done by means of the proper methods:

`setDConstant(doublevalue)`

`setDPatchDataId(intid)`

`setCConstant(doublevalue)`

`setCPatchDataId(intid)`

where the integers `id` for spatially varying C and D coefficients definition are the variables database indexes of the two respective quantities stored on the mesh.

Is to be noticed that no explicit reference to the boundary conditions have been made within the code. It will be shown in the next section that the only initialization of the boundary conditions is made via input file, and then inherited for every patch of the hierarchy.

```
int FACPoisson::solvePoisson()
{
    int ln;
    for( ln=0; ln<=d_hierarchy->getFinestLevelNumber(); ++ln ) {
        tbox::Pointer<hier::PatchLevel<NDIM> > level =
          d_hierarchy->getPatchLevel(ln);
        hier::PatchLevel<NDIM>::Iterator ip(*level);
        for( ; ip; ip++ ) {
            tbox::Pointer<hier::Patch<NDIM> > patch =
              level->getPatch(*ip);
            tbox::Pointer<pdat::CellData<NDIM,double> > data =
              patch->getPatchData(d_comp_soln_id);
            data->fill(0.0);
        }
    }

    /*
     * Poisson equation charachterization
     */

    d_poisson_fac_solver.setDConstant(1.0);
    d_poisson_fac_solver.setCConstant(0.0);
    d_poisson_fac_solver.initializeSolverState(
        d_comp_soln_id,
        d_rhs_id,
        d_hierarchy,
        0,
        d_hierarchy->getFinestLevelNumber()
    );
    int solver_ret;
    solver_ret = d_poisson_fac_solver.solveSystem(
        d_comp_soln_id,
        d_rhs_id
    );
    double avg_factor, final_factor;
    d_poisson_fac_solver.getConvergenceFactors(
        avg_factor,
        final_factor
    );
    d_poisson_fac_solver.deallocateSolverState();
    return 0;
}
```

The problem specific Input file

The input file management in SAMRAI is a sofisticated collection of informations and parameters defining the simulation. Every fundamental aspect of the framework is tuned indeed by the user via input file, without any need to modify the source code.

The first section of the input file defines the name of the simulation, the simulation output logging method and the visualization output to be produced.

```
Main {
  // Base name for output files.
  base_name = "escpif"
  // Whether to log all nodes in a parallel run.
  log_all_nodes = FALSE
  // Visualization writers to write files for.
  vis_writer = "Vizamrai", "VisIt"
}
```

The fundamentals parameters to be specified for every simulation include the geometry definition. The user is called to specify the base domain simulation box (by means of the indexes of both the lower-left and the upper-right corners) and the corresponding position of the domain in the real space.

```
CartesianGeometry {
  // x_lo  -- (double array) lower corner of computational domain [REQD]
  // x_up  -- (double array) upper corner of computational domain [REQD]
  // domain_boxes  -- (box array) set of boxes that define interior of
  //                       physical domain. [REQD]
  // periodic_dimension -- (int array) coordinate directions in which
  //                       domain is periodic.  Zero indicates not
  //                       periodic, non-zero value indicates
  //                       periodicity.
  domain_boxes = [(0,0), (500,150)]
  x_lo         = 0, 0
  x_up         = 10, 3
}
```

The tagging method for either static or dynamic regridding of the mesh has to be specified in the `StandardTagAndInitialize` section. The possible choices are:

`REFINE_BOXES`, followed by a list of static boxes to be refined for each hierarchy level index space;

`GRADIENT_DETECTOR` for dynamical regridding based on gradient threshold cell tagging;

`RICHARDSON_EXTRAPOLATION` for the most sophysticated (but computationally the most expensive) tagging method: it is based on the computation of the differences in the solution obtained on the existing mesh and on an hypothetically regridded mesh[6].

The user can also define a load balancing criterion (the default one is proportional to the number

[6]For further informations, the reader is referred to the documentation included in the SAMRAI distribution.

of cells per patch) by means of proper functions and/or variables to compute a different weight for every patch, in order to re-balance an otherwise odd computational load.

```
StandardTagAndInitialize {
  // tagging_method -- Type of tagging criterion used for refinement.
  //                       Possible choices are:
  //                       REFINE_BOXES [RQRD list of static patches]
  //                       GRADIENT_DETECTOR
  //                       RICHARDSON_EXTRAPOLATION
  tagging_method = "REFINE_BOXES"

  RefineBoxes {
    level_0 = [(0,50),(300,75)]
    level_1 = [(50,0),(200,25)]
    //etc.
  }
}

LoadBalancer {
// using default uniform load balance configuration
}
```

If a dynamical regridding method is chosen, the user can tune the latter by defining the maximum number of levels of refined patches to be considered, their eventual maximum dimension (for load balancing reasons) and the ratio between the base and the refined patches grid steps.

```
GriddingAlgorithm {
   // max_levels -- (int) max number of mesh levels in hierarchy. [REQD]
   max_levels = 3

   // ratio_to_coarser {
   //   level_1 -- (int array) ratio between index spaces on level 1 to     //                       level 0. [REQD]
   //   etc... [REQD]
   // }
   // largest_patch_size {
   //   level_0 -- (int array) largest patch allowed on level 0. [REQD]
   //   etc... [Optional]
   // }
   ratio_to_coarser {
      level_1            = 2, 2
      level_2            = 2, 2
   }
   largest_patch_size {
      level_0 = 32, 32
      // all finer levels will use same values as level_0...
   }
}
```

At least the user is asked to specify the FAC solver parameters (a list of which can be found on the documentation included with the SAMRAI distribution, together with the default values) and the boundary conditions of the problem. These conditions are initially defined only on the base mesh, and then inherited for every patch which overlaps the domain boundary.

The surfaces to apply the boundary conditions to are identified by means of an integer index

obtained by natual ordering of the boundaries: every spatial direction is progressively followed until the last surface before advancing along the next direction, like in a column major ordering scheme.

On every surface the boundary conditions can be specified in several ways, the most common of which are:

- by defining the constant value the unknown assumes on that surface;
- by specifying the two coefficients a and g for the most general Robin boundary condition

$$au + (1 - a)\frac{\partial u}{\partial n} = g \tag{2}$$

  where $a \in [0, 1]$; it is interesting to notice that the common Dirichelet and Neumann conditions can be obtained by letting $a = 1$ and $a = 0$ respectively.

```
FACPoisson {
  fac_solver {
    enable_logging = TRUE   // Bool flag to switch logging on/off
    max_cycles = 10         // Max number of FAC cycles to use
    residual_tol = 1e-8     // Residual tolerance to solve for
    num_pre_sweeps = 1      // Number of presmoothing sweeps to use
    num_post_sweeps = 3     // Number of postsmoothing sweeps to use
    prolongation_method = "LINEAR_REFINE" // Type of refinement
                                          // used in prolongation.
    use_smg = TRUE // Whether to use HYPRE's SMG instead of PFMG.
  }
  bc_coefs {
    // boundary_n - (int) n is the location index of the boundary.
    // textstring -- type of values to set: possible cases are
    //                          value to set boundary value
    //                          slope to set boundary slope
    //                          coefficients to set Robin bc
    // valuestring -- converted to number and set to boundary
    boundary_0 = "coefficients", "0", "0"
    boundary_1 = "coefficients", "0", "0"
    boundary_2 = "value", "0"
    boundary_3 = "value", "0"
  }
}
```

Design patterns

For completion, a list of the design patterns implemented in the development of the SAMRAI framework is here presented and briefly commented:

Singleton Classes Only allow a single instantiation, e.g. RestartManager. Cannot be directly created by constructor; pointers instead may be statically accessed. Memory management is not user controlled.

Smart Pointer Manages memory allocated for objects, keeping count of references to the object and automatically deleting it when that number equals 0. For Smart Pointer to work for an object, only Smart Pointers must be used to point that object.

Strategy Pattern Defines a common interface for a family of algorithms so that they can be interchanged.