ANL-AFCI-215

Argonne NATIONAL LABORATORY

Status Report on SHARP Coupling Framework

Nuclear Engineering Division
Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne, see www.anl.gov.

Availability of This Report

This report is available, at no cost, at http://www.osti.gov/bridge. It is also available on paper to the U.S. Department of Energy and its contractors, for a processing fee, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone (865) 576-8401
fax (865) 576-5728
reports@adonis.osti.gov

ANL-AFCI-215

Status Report on SHARP Coupling Framework

by
C. Rabiti, M. A. Smith, W. Yang, G. Palmiotti
Nuclear Engineering Division

A. Caceres, A. Siegel, T. J. Tautges, J. Lottes, P. Fischer, D. Kaushik
Mathematics and Computer Science Division

September 30, 2007

work sponsored by

U. S. Department of Energy,
Office of Nuclear Energy, Science and Technology

UChicago ▶ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

Status Report on SHARP Coupling Framework

A. Caceres, A. Siegel, T. J. Tautges, J. Lottes, P. Fischer, D. Kaushik
Mathematics and Computer Science Division

C. Rabiti, M. A. Smith, W. Yang, G. Palmiotti
Nuclear Engineering Division

September 2007

Abstract

This report presents the software engineering effort under way at ANL towards a comprehensive integrated computational framework (SHARP) for high fidelity simulations of sodium cooled fast reactors. The primary objective of this framework is to provide accurate and flexible analysis tools to nuclear reactor designers by simulating multiphysics phenomena happening in complex reactor geometries. Ideally, the coupling among different physics modules (such as neutronics, thermal-hydraulics, and structural mechanics) needs to be tight to preserve the accuracy achieved in each module. However, fast reactor cores in steady state mode represent a special case where weak coupling between neutronics and thermal-hydraulics is usually adequate. Our framework design allows for both options. Another requirement for SHARP framework has been to implement various coupling algorithms that are parallel and scalable to large scale since nuclear reactor core simulations are among the most memory and computationally intensive, requiring the use of leadership-class petascale platforms.

This report details our progress toward achieving these goals. Specifically, we demonstrate coupling independently developed parallel codes in a manner that does not compromise performance or portability, while minimizing the impact on individual developers. This year, our focus has been on developing a lightweight and loosely coupled framework targeted at UNIC (our neutronics code) and Nek (our thermal hydraulics code). However, the framework design is not limited to just using these two codes.

Contents

1 Introduction

The SHARP project (Simulation-based High-efficiency Advanced Reactor Prototyping) at Argonne National Laboratory is a multi-divisional collaborative effort to develop a modern set of design and analysis tools for liquid metal cooled fast reactors. Ultimately, this suite of codes is envisioned to replace, piecemeal, existing legacy tools that were first designed over twenty years ago and have since served as the standard for fast reactor design and analysis. SHARP includes a strategy to allow newly built codes to coexist with and couple to legacy codes as part of an incremental "phasing in" that allows uninterrupted productivity by the reactor design team end users.

In this report we report on our early efforts to develop an improved modeling capability specifically for the reactor core. Conceptually, the physical phenomena, though coupled, can be decomposed roughly along traditional mono-disciplinary lines: heat transfer, neutron transport, and structural/fuel behavior. In each area (to varying degrees) the legacy codes are considerably simplified compared to what one finds in companion fields – this is true e.g. in terms of supported dimensionality, spatial/temporal resolution, numerical methods, physical models, and sophistication of software design. In terms of the physics, traditionally some degree of tuning and calibration has been used to "validate" the codes, which then has enabled predictions for states in some sense "close to" this validation baseline. Drastically improved models offer the hope of both more accurate predictions (to reduce uncertainty margins, e.g. "hot channel factors") as well as something closer to *virtual prototyping* – viz. predictive simulations for regimes much further from an experimental reference.

Given the complexity and unique requirements of the SHARP software, considerable up-front work must be done to ensure that the individual modeling efforts are unified to meet the physics/engineering goals of the project from the perspective of the end user. This includes allowing the end user to select combinations of physics models based on the specific design problem – e.g. low resolution, fast turnaround single-physics studies for early scoping using sub-channel models; localized Direct Numerical Simulations (DNS) or Large Eddy Simulations (LES) to study isolated physical effects; full core Reynolds Averaged Navier-Stokes (RANS) for late-stage design studies; tightly coupled modules for fast transients; Monte Carlo calculations to establish benchmark baselines; deterministic second order calculations for homogenized whole core power profiles, etc.

Assembling a coupled physics capability from existing physics modules, while preserving the ability of those modules to operate (and be developed)

in standalone fashion, requires a careful design of the over all coupling framework. This framework must balance simplicity of adding new physics modules and support for commonly needed functionality (e.g. parallel IO), on one side, with efficiency and minimization of dependencies between modules on the other. Striking the right balance in this spectrum has motivated our decision *not* to use integrated frameworks developed in other DOE programs, e. g. the Common Component Architecture (CCA) or the Model Coupling Toolkit (MCT). However, we have chosen to use technology developed in the SciDAC program, particularly from the ITAPS and TOPS projects, precisely because they enable and simplify the "loose" coupling approach described above. The reliance on this design approach will become more evident in later years of the project, but has also guided the preliminary work done this year and described in this report.

In this report we present the initial design of the SHARP framework in the context of the simpler sub-problem of a coupled reactor core simulation. The intent is to illustrate the framework design, its interaction with the development of the individual physics modules, and how ancillary services such as unit testing, pre- and post-processing, and parallel coupling are provided by the framework in a loosely coupled and flexible manner.

2 Problem Statement

The design of the SHARP framework includes specifications for the "hooks" (placeholders) for both the individual physics modules and computational tools that comprise the entire code system. Additionally, SHARP contains particular implementations of these physics modules that are critical to meeting our physics/engineering goals in the early phase of the project. It is understood that alternative codes and/or enhancements to these existing codes will be continuously required throughout the evolution of the project. Defining and refining the rules that will accommodate these alternate implementations is a key focus of the early part of the project.

To study specific steady state and slowly transient phenomena, we follow a loosely coupled design philosophy that emphasizes the independence of the individual modules while retaining the ability to couple them in a variety of configurations (or to run modules in standalone mode). Abstractly, the framework includes the following *physics components*: heat transfer, neutron transport, depletion, and fuel/structural materials. Additionally, there is a complementary collection of *utility modules* for cross section processing, material properties, mesh generation, visualization, solution transfer

between meshes, load balancing, parallel i/o, and unit/integral testing.

2.1 Governing equations

Assuming suitable boundary conditions are applied, the coupled reactor core neutronics-thermal/neutronics equations can be written as a single coupled system:

$$\dot{\mathbf{f}} = \mathbf{L}\mathbf{f} + \mathbf{N}\mathbf{f} \tag{1}$$

where $\dot{}$ denotes time differentiation, $L(\mathbf{f})$ denotes the linear part of the operator, $N(\mathbf{f})$ the nonlinear part, and

$$\mathbf{f} = \begin{bmatrix} \psi \\ T \\ \rho \\ \vec{u} \end{bmatrix} \tag{2}$$

In (2), ψ is the angular neutron flux, T the medium temperature, ρ the medium density and $\vec{u}$ is the coolant velocity. The temperature and density can further be divided into coolant and fuel regions, denoted by $T = T_c \cup T_f$ and $\rho = \rho_c \cup \rho_f$. The angular flux ψ is then obtained from the linear transport equation:

$$\begin{aligned} \left[\frac{1}{v}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\nabla + \sigma_{\rho,T}(\vec{r},E)\right]\psi(\vec{r},\hat{\Omega},E,t) = \\ q_{ex}(\vec{r},\hat{\Omega},E,t) \\ + \int dE' \int d\Omega' \sigma_s(\vec{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\psi(\vec{r},\hat{\Omega}',E',t) \\ + \frac{\chi(E)}{4\pi}\int dE'\nu_f\sigma(\vec{r},E')\int d\Omega'\psi(\vec{r},\hat{\Omega}',E',t). \end{aligned} \tag{3}$$

In (3), v is the scalar neutron speed, E the neutron energy, $\vec{r} = (x,y,z)$ the spatial coordinate, t the time, $\hat{\Omega} = (\theta,\phi)$ the direction of neutron travel, q_{ex} an external neutron source, $\sigma_{\rho,T}$ the total interaction cross section, σ_s the scattering cross section, χ the fission spectrum, and ν_f the number of neutrons emitted per fission.

The subscripts on $\sigma_{\rho,T}$ are explicitly included to denote the dependence of cross sections on both the fuel and coolant temperature and density, which

is the principle source of (non-linear) coupling between different physics modules.

With a solution to (3) the volumetric heat generation rate q''' can be estimated by:

$$q'''(\vec{r},t) = \int dE' \sigma(E') W(E') \int d\Omega' \psi(\vec{r},\hat{\Omega}',E',t) \tag{4}$$

The heat equation for both fluid and fuel can then be solved with q''' as a source (of course $\vec{u} = 0$ in fuel region):

$$\rho C_p \left(\frac{\partial}{\partial t} + \vec{u} \cdot \nabla \right) T(\vec{r},t) = \nabla \cdot \kappa_T \nabla T(\vec{r},t) + q'''(\vec{r},t) + f \tag{5}$$

The fluid velocity $\vec{u}$ in (4) is obtained from the Navier-Stokes equation. For simplicity, we write the Boussinesq approximation of the equation where density changes are considered negligible except in the buoyancy term (in general an equation of state is needed to close the system):

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\frac{1}{\rho_0} \nabla p + \nabla \cdot \nu_T \nabla \vec{u} + \frac{\rho'}{\rho_0} g \hat{k} \tag{6}$$

$$\nabla \cdot \rho_0 \vec{u} = 0 \tag{7}$$

Equation (6) is non-linearly coupled to the heat equation through the temperature dependence of the dynamic viscosity ν, denoted explicitly by ν_T.

The details of the discrete formulations of above equations are reported in companion papers [1][2] and are not essential to understanding the inter-module coupling problem. When the physics is split into separate components, these couplings take the form of specific data values that are interpolated from source to target meshes and sent between the modules through well defined interfaces.

SHARP currently includes the following specific implementations of physics modules to solve the above equations:

Nek: a spectral element code that solves the 3-D incompressible (Boussinesq) time-dependent Navier-Stokes equation with conjugate heat transfer on unstructured higher-order quadrilateral meshes [5]. Nek is written primarily in Fortran77 using MPI for distributed memory systems and has recently showed good scalability to 30,000 processors on BG/L.

UNIC: an unstructured deterministic neutron transport code that incorporates parallel even parity, sweeping, and ray tracing algorithms flexibly

within the same code. UNIC has recently been parallelized using PETSc (which itself uses MPI) and is currently running benchmarks on moderate-sized clusters as a precursor to larger scalable runs on Cray XT/4 and IBM Blue Gene systems.

Separating physics modules into distinct components and implementing coupling as interactions between these components imposes a number of requirements on the overall design of the SHARP framework. This design will have a strong influence on the degree and frequency with which these interactions can occur during a given simulation. The design of the SHARP framework is discussed in general terms next, followed by descriptions of specific use cases.

3 Design of Coupling Framework

We start with a collection of physics codes which are currently developed and run in standalone mode. To assemble pieces of these codes into a coupled physics code, we first separate each physics module into a driver and library, with the bulk of modeling located in the library and the driver containing code specific to a particular use case. Coupling between physics modules is implemented by passing data through a common mesh representation, accessed through a common Application Programming Interface (API). This approach preserves flexibility in a number of key aspects, while also providing a common focal point for the coupling activities.

The mesh API is an important part of this design, since it acts as the communication mechanism between physics modules. As described below, it also provides access to various mesh services needed for high-performance reactor simulation. The ITAPS mesh interface, which is used for this API, is described next.

3.1 The ITAPS Mesh Interface

The Interoperable Tools for Advanced Petascale Simulations (ITAPS) center is a collaboration between several universities and DOE laboratories funded by the DOE Scientific Discovery for Advanced Computing (SciDAC) program. The primary objective of the ITAPS center is to simplify the use of multiple mesh and discretization strategies within a single simulation on petascale computers. This is accomplished through the development of common functional interfaces to geometry, mesh, and other simulation data. Although eventually all the ITAPS interfaces will be used, in this report only

the ITAPS mesh interface is described. The implementation of this interface we are using is MOAB [4].

The data model defined in ITAPS mesh interface consists of four basic data types:

Entity: The basic topological entities in a mesh, i.e. the vertices, triangles, tetrahedra, etc.

Entity Set: Arbitrary groupings of entities and other sets. Sets can have parent/child relations with other sets.

Interface: The object through which mesh functionality is accessed and which "owns" the mesh and its associated data.

Tag: A piece of application-defined data assigned to Entities, Entity Sets, or to the Interface itself.

This data model, although quite simple, is able to represent most data in a typical PDE-based simulation. For example, combinations of sets and tags can be used to represent processor partitions in a mesh, groupings of mesh representing the geometric topology on which the mesh was generated, and boundary condition groupings of mesh. It has also been shown to allow efficient storage of and access to mesh data.

The MOAB library [6] is an implementation of the ITAPS mesh interface interface. MOAB provides memory- and CPU time-efficient storage and access to mesh data using array-based storage of much of these data. MOAB provides element types encountered in most finite element codes (including quadratic elements) as well as polygons and polyhedra. Tools included with MOAB provide parallel partitioning, parallel IO, and commonly used functions like finding the outer skin of a contiguous mesh.

3.2 Requirements

There are several process-based requirements which guide the design of SHARP for various reasons:

- *Minimally intrusive:* A variety of codes and code modules have already been developed for use in reactor simulation, each with tens or even hundreds of man-years invested in their development and qualification. It is infeasible to expect that these applications be entirely re-written to fit into a new code coupling framework. Therefore, the process of attaching a new code or physics module to the framework must be minimally intrusive to the original code.

- *Compatibility with standalone development:* There are many simulation methods which, while not developed originally for that purpose, are well suited to reactor simulation [5]. The coupling framework must be compatible with standalone development of physics modules, to be able to get updates to these modules as they are developed.

- *Utility services:* New physics modules are being developed which could take advantage of various advanced techniques like adaptive mesh refinement if components implementing these techniques could be incorporated easily. The coupling framework must also include utility services like these, along with a mechanism to add other services as they become available.

- *Integrated multi-physics:* Finally, there is a need to integrate physics modules of various types to perform coupled simulation for some problems. The coupling framework should be designed to minimize the effort required to incorporate additional physics modules and couple them to other modules in the framework. The framework must also be designed to enable such coupling to be computationally efficient; that is, the framework itself must not impose a large overhead on the basic computations required for coupling.

3.3 Spatial Domain Coupling

There is a wide spectrum of possibilities in how to couple various types of physics modules together. In a loosely coupled system, each physics might be solved on an entirely different spatial discretization, with exchange of data only at the beginning of each timestep or iteration. Closely coupled systems can use related or identical grids, and can even be formulated implicitly with the other physics and solved in the same solution step. However, a common element in all these formulations is the spatial domain or discretization(s) on which the physics are solved. The SHARP framework design couples physics together and with other services through the spatial discretization or mesh.

The mesh can be accessed using the Interface object described in 3.1. This domain can be presented using the ITAPS data model as an entity set; if multiple meshes are used for loosely coupled systems, these are presented simply as multiple entity sets within the same Interface object. The dependent variables computed by each physics module can be written to the grid as tags, either at the end of each call to the module or throughout the course of that module's execution. Hence the mesh acts as a communication

mechanism between physics modules through those tags. If multiple grids are used, other utility modules are used to map data between those grids.

A graphical depiction of this framework is shown in Figure 1. The mesh Interface, implemented using MOAB, acts as the focal point for interactions between physics modules and with various other services. Various types of codes can be constructed on this framework, as described below.

3.3.1 Basic requirement: driver & library

A basic requirement we use to accomplish this coupling approach is to separate each module into a driver code and a library, where the driver sets up the calculation assuming standalone operation, then calls functions in the library to compute the actual physics. The standalone driver implements its own IO, communication, and other functionality otherwise found in the framework of the coupled code. The libraries of physics capabilities are shown in Figure 1 as libPhys1 and libPhys2.

3.3.2 Standalone physics code

Assuming the separation into driver and library, a physics code can be built and developed as a standalone code, as shown for driver1 in Figure 1. Improvements to most physics capabilities are likely to occur in the library rather than in the driver, and thus are available to other use cases which rely on the library. This approach requires the additional effort to define functional interfaces to new capabilities as they are added to the library. However this effort is low, given that these interfaces do not change frequently.

Given this structure, new developments in the physics module do not prevent coupled solutions based on that module from running as before (because the interfaces to previous capabilities do not change). Coupled solutions can take advantage of new capabilities simply by using the interfaces defined for those functions.

3.3.3 Standalone physics code + services

Using a common mesh API gives a code access to other mesh services also using that API. For example, mesh services based on the ITAPS mesh interface include mesh partitioning [8], adaptive mesh refinement [9], and mesh smoothing [7]. Using the MOAB implementation of this mesh API provides a mesh representation which is highly memory- and cpu time-efficient as well [4]. For codes which do not already have a significant investment and

which are early in the development cycle, using this mesh representation speeds development of the physics capability. This approach also enables the physics module to use advanced computational techniques like adaptive mesh refinement that they would otherwise not have time to develop themselves.

This use case is depicted in Figure 1 as the combination of driver2, libPhys2, and the core mesh API and services. In this case, the physics module can couple either directly to MOAB, for efficiency, or through the Mesh API, to preserve the option of using a different implementation of the Mesh API.

3.3.4 Coupled physics code

The degree of coupling used for a given coupled code depends not only on what is appropriate numerically, but also what kinds of physics modules, solution strategies, and grids are available for that code. For example, requiring two physics modules to solve on the same grids restricts the choice of available implementations of each module to those which use the chosen grid type or those which can be changed to use that grid type in the available time. In some cases this approach will over-constrain the problem, due to resource constraints or simply because the required physics modules are not available. Therefore, we choose to preserve as much flexibility as possible, by designing the system to allow both loose and tight coupling, on the same grid or different ones.

If different grids are used, there will be a need to pass solution data from one grid to another. This process can be accomplished using a "data coupler" tool, which is simply another tool implemented on top of the mesh API. In our approach, we require both meshes be accessed through the same Interface object, and likely in the same implementation (in our case, MOAB). This is desirable from the standpoint of computational efficiency. This preserves the option of implementing the data coupler inside MOAB, allowing it to operate on data in its native representation (rather than indirectly through a Mesh API) for greater efficiency. The only additional requirement for this approach is that MOAB be able to represent grids used by the various physics modules. This is our motivation for choosing MOAB, which is general enough to do this.

Figure 1 depicts a coupled driver incorporating all of these design elements. In this code, libPhys1 retains its native data representation, using a

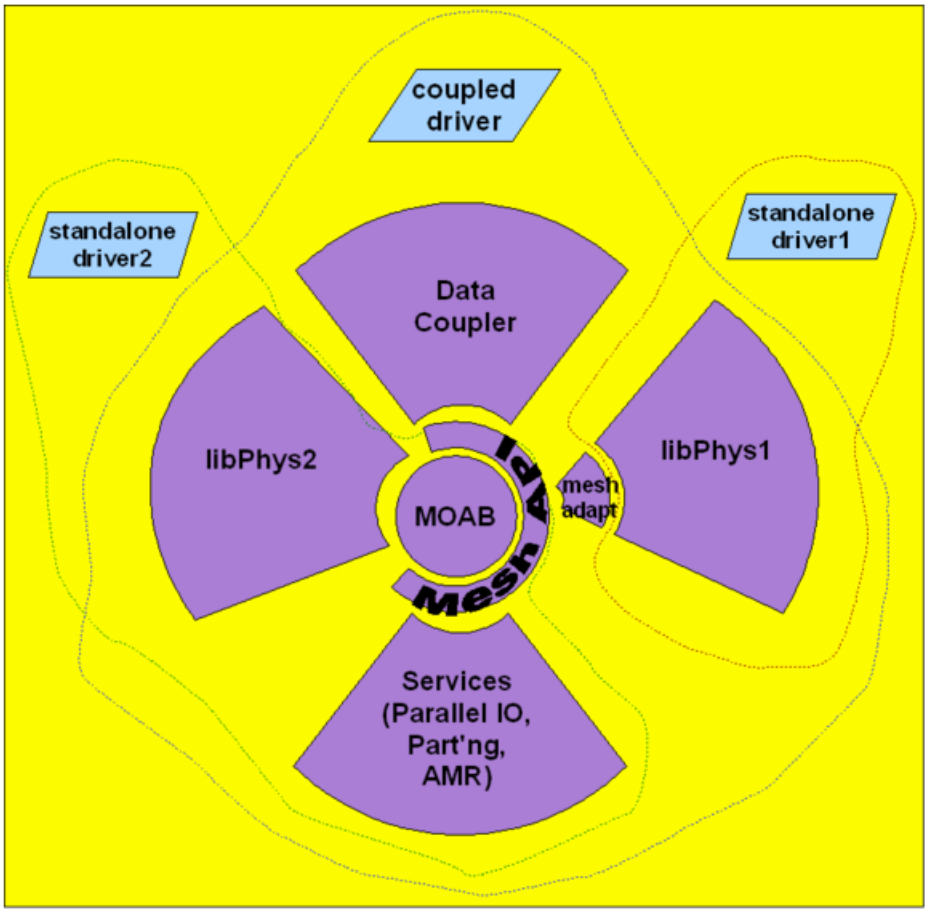

Figure 1: **Interrelationship among the components of MOAB**

mesh adapter to pass a subset of its data through the Mesh API for coupling purposes. libPhys2 uses MOAB as its native representation, for efficiency, and therefore does not need an adapter. The code system uses common services available through the Mesh API, including a data coupler to couple data from libPhys1 and libPhys2. The overall solution process and passing data between physics modules is coordinated by the coupled driver. In the future, it may be desirable to separate some of this coordination logic into an additional library, for use in multiple, separate coupled codes.

4 Coupling Theory

4.1 Data Flow

Given appropriate discretizations of (3) on mesh Ω_n and (4) on mesh Ω_{th}, regardless of the numerical procedure used within each solver and the details of the coupler, the following is an abstract description of the algorithmic structure for the coupled steady state problem:

Step 1: Start with initial guess for macroscopic cross section $\Sigma, \rho_c, \rho_f, T_c, T_f$

Step 2: Solve (3) and (4) to get ϕ and q''' on Ω_n, respectively

Step 3: Map q''' on Ω_n to q''' on Ω_{th}

Step 4: Solve (6) to get ν and α on Ω_{th}

Step 5: Solve (4) and (7) for T_f, T_c, and ρ_c on Ω_{th}, using q''' on Ω_{th} as source term

Step 6: Map T_f, T_c and ρ_c from Ω_{th} to Ω_n, and use them to compute Σ on Ω_n

Figure 2 shows how this algorithm is realized in terms of the different modular components of the code. Note that each module individually, including the mesh component, is partitioned across a distributed memory parallel machine in a way that in general is completely arbitrary and configurable by the user (or automated by load balancing software with user suggestions). Also notice that this description is valid for steady state problems as well as slowly evolving physics (e.g. burnup). When studying faster transients, the architecture will in general need to allow a tighter degree of coupling to achieve both good performance and adequate time accuracy.

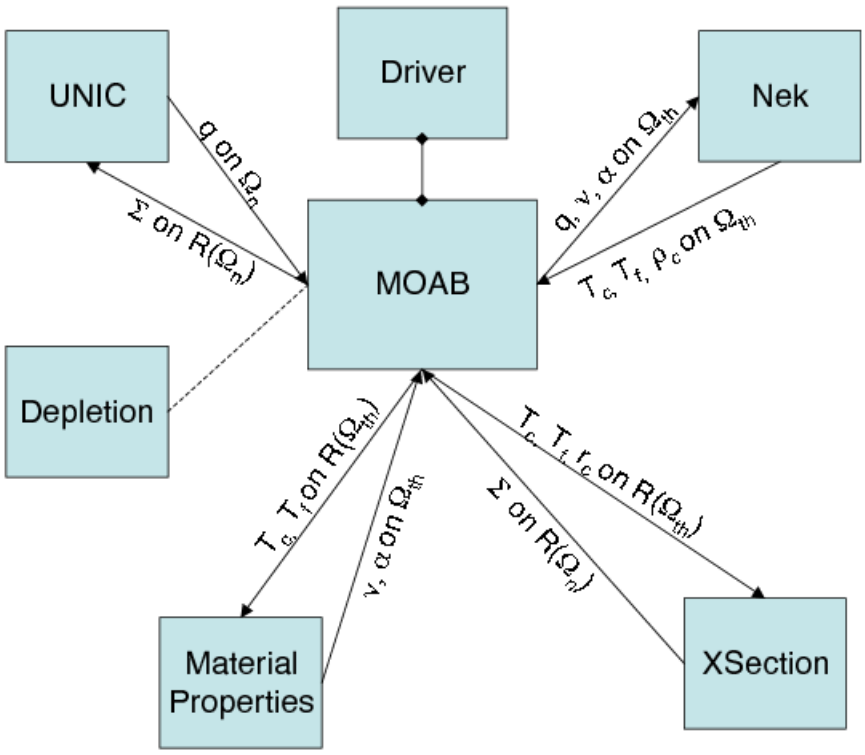

Figure 2: **Depiction of the individual modules of SHARP and their interrelationships. Ω_n denotes the UNIC mesh, Ω_{th} the Nek mesh and R is a linear restriction (coarsening) operator that represents entities on homogenized regions of the mesh.**

4.2 Data Volume

As shown in Figure 2, we need to exchange data proportional to the size of the coarser of the meshes used by the neutronics module (UNIC) and the thermal hydraulics module (Nek). In practice, the neutronics mesh is usually much coarser than thermal hydraulics mesh. We expect to use between one million (subassembly level) to fifty million (full core level) mesh vertices in UNIC while Nek will use roughly ten times that amount. The number of degrees of freedom (DOF) in UNIC is proportional to the product of number of energy groups (G_N), mesh vertices (S_N), and angular moments ($A_N = \frac{n(n+1)}{2}$ where n is the P_n order). So the ratio of DOF to data exchanged is $G_N \times A_N$. Since we ultimately expect to use a large number (O[10,000]) of energy groups and high angular order ($\tilde{P}_{15}$), the data exchanged is quite minimal relative to the DOF employed in the neutronics problem. Though the cross sections used in UNIC depend on temperature and density, these variations in the steady state case are also expected to be small. Therefore, the coupling between UNIC and Nek is expected in general to be weak with a relatively small amount of data exchange (as compared to the DOF each computation). We are currently beginning detailed coupling performance studies to test this hypothesis.

5 Implementation

In our implementation, all modules are compiled into a single executable, SHARP, which then runs as a single MPI process. The top-level execution flow is controlled by the SHARP driver, which successively calls the individual modules' API functions, as described in section 3.

At the moment we have successfully coupled two physics modules, Nek and UNIC, and are running small (32 processor) test problems on the *Jazz* cluster at ANL. The current implementation departs from the specification outlined in section 3 in that the physics modules are not currently copying their data into the ITAPS mesh interface; instead, the data is copied (via a module API function) into placeholder arrays created by the driver. This is a temporary measure taken in order to explore physics coupling while portions of the framework (mainly coupling computations on MOAB) are being put in place.

5.1 Module API

The current API for bot Nek and UNIC consists mainly of the following high level c-callable functions (*phys* below stands for either *nek* or *unic*):

`phys_init` Module initialization: allocate storage, set up MPI communicator, read in input files, etc.

`phys_solve` Compute our part of the solution to the problem.

`phys_export` Write the solution from the module's internal data structures to memory allocated by the driver (stand-in for MOAB/ITAPS)

`phys_import` Read in the part of the solution that is calculated by the other modules. For example: if *phys* is Nek, the thermal-hydraulics module, read in the heat generation computed by the neutronics module.

`phys_write_data` Write our data to disk, in our native format.

`phys_finalize` Clean-up: close files, compute diagnostics, etc.

When data needs to be transferred between two modules, the SHARP driver calls their import/export functions with pointers to memory allocated by the driver; as the two modules use different meshes (and define their data on these meshes in different ways), the driver must call a (parallel) mapping function between export and import steps. This is discussed in section 5.4.

5.2 Common conversion changes

Converting a standalone application to expose an API as described in section 5.1 is for the most part a simple question of refactoring code or creating a wrapper layer. New code is in principle only needed to implement the `import` and `export` functions. This approach has a negligible impact on the performance of the standalone application.

One key aspect of the refactoring is that data declarations in the entry point of the standalone application *phys* (its `main` function) must be moved to the `phys_init` function.

Another required change is the switch from using MPI's global communicator (which is sufficient for most standalone codes) to a new communicator specific to the module. This is necessary in the coupled case if, for example, we wish to run a module on only a subset of the processors that SHARP has available.

A consequence of our decision to combine modules into a single executable is the possibility of name clashes. These are resolved by renaming symbols at the object-file level using the (extremely portable) `GNU objcopy` utility.

Other issues are anticipated to appear as new modules are added, however we believe most codes can be relatively easily adapted to work with our framework, without losing performance or the ability to exist as standalone programs.

5.3 Configuration and build system

We have in place a custom configuration/build system which tries to accommodate different modules' existing build systems, should they exist. The main design requirement behind it is to have the possibility that, after a module has been added to the SHARP source tree, it is always possible to automatically create a source distribution of the standalone application. To the SHARP user, our system looks much like the "`./configure; make`" standard.

5.4 Data exchange details

We now describe in detail the data that the driver must map between UNIC and Nek. Physically, UNIC takes temperature T and provides the volumetric heat generation rate q''', and vice-versa for Nek. Computationally, the data is represented as follows:

UNIC: Uses a linear hex mesh (other element types are possible in UNIC but not currently supported by the coupler); this mesh is referred to as Ω_n in 4.1. q''' is given per-element. UNIC partitions Ω_n into sets of elements called *blocks*. As input, UNIC expects the average T inside each block, which it then uses to compute cross-sections. All communications between the driver and UNIC occurs on the root processor (this is a feature of UNIC which is expected to eventually change).

Nek: Uses hex spectral elements of order N, where N is generally between 4 and 10; this mesh is referred to as Ω_{th} in 4.1. Each element has $N \times N \times N$ grid points, on which are defined all fields (including q''', T and x, y, z coordinate fields). Thus, to specify a physical field in Nek its value must be given at each grid point. To evaluate a field at an arbitrary point in space, one needs to determine what

element that point lies in and then perform a weighted sum of all the gridpoints belonging to that element. Finding the weights for a given point requires inverting the coordinate fields numerically to obtain the point's parametric coordinates. Elements are partitioned among processors and communication with the driver occurs on every processor.

Given this, one simple prescription for interpolating data between meshes is: for each Nek gridpoint, set its q''' value to that of the UNIC element it lies in; to interpolate from Nek to UNIC, integrate T over each UNIC block by evaluating T on the Nek mesh at every UNIC node and do weighted sum using the known UNIC element volumes. Thus, our interpolation method requires computing the following *interpolation maps*:

- For each UNIC vertex we need to know inside which Nek element it lies and its parametric coordinates within that element.
- For each Nek grid point we need to know inside which UNIC element it lies.

Part of Nek is a library, *findpt*, that, given a distributed list of points and distributed Nek-like mesh, will compute the location in the mesh of each point (meaning, what element it's in, its parametric coordinates within that element, and which processor owns the element). At the beginning of a coupled simulation, we compute and store in parallel the interpolation maps using *findpt*. During the course of a simulation, the map is used to efficiently interpolate data between Nek and Unic. For steady state calculations, the cost in both memory and time of interpolation (including computation of the maps) is negligible. *Findpt* is known to scale, like Nek, to tens of thousands of processors.

We expect to refine these interpolation methods in the future, as we need to properly address the issue of energy conservation, in particular for the case of q''' interpolation.

Figures 3 and 4 illustrate our interpolation method in two views of one coupled steady-state simulation of a single pin with sodium channel.

6 Mesh Generation

An important part of supporting reactor simulation is generating input for the various analysis codes. For the codes being developed in SHARP, the

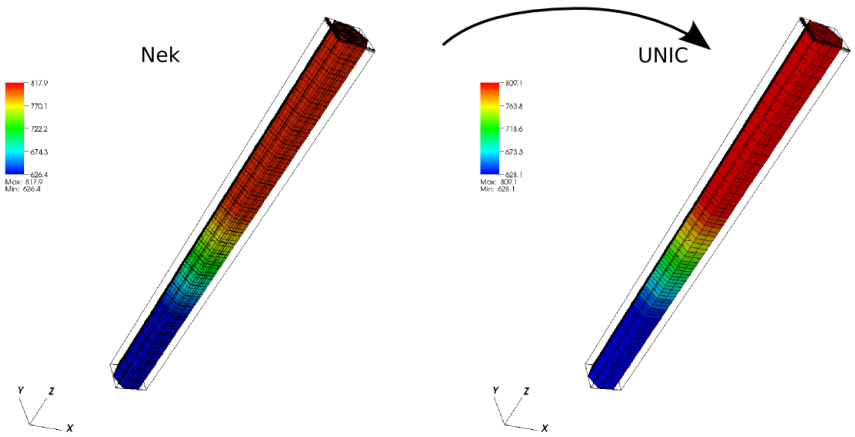

Figure 3: T **mapped from Nek to UNIC blocks, single pin. The range of values differs due to block-averaging. Vertical axis has been scaled by a factor of** 0.05**.**

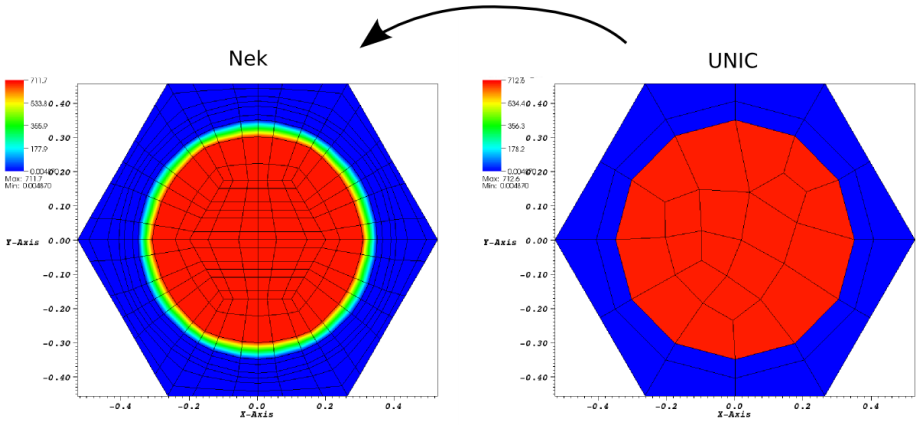

Figure 4: q''' **mapped from UNIC to Nek, horizontal cross-section through the middle of a single pin with sodium. In the case of Nek, inner nodes are rendered (order of spectral elements is** 4**).**

input consists of finite element meshes of various types. During this initial phase of our GNEP work we explored the use of the CUBIT mesh generation toolkit from Sandia National Labs [10] for this purpose. Two models were developed, supporting neutronics and fluid flow computations; these efforts are described below.

6.1 ABTR Mesh

The Advanced Breeder Test Reactor (ABTR) is an important focus of the overall reactor simulation part of GNEP. This model consists of hexagon-shaped assemblies of various types (inner and outer fuel, control, reflector, and shield), each assembly consisting of a hexagonal lattice of pins inside a hexagonal duct wall. For a more detailed description of this reactor design, see [11].

Because of the relatively straightforward layout of simple hex-shaped primitives, a geometric model for this core was developed inside CUBIT (more complex geometric models would normally be developed in a CAD tool like Pro/Engineer). The geometric model is shown in Figure 5 for a 60-degree section of the core. Generating this model with CUBIT was a very memory-intensive process, requiring almost all of the 4GB of RAM on a 64-bit workstation even without the use of graphics. Generating a mesh for this model required over 5GB of memory, even for a moderately-sized mesh of several million elements. Other problems encountered in this effort include:

- Problems generating relatively coarse meshes for hexagonal-shaped annular surfaces, resulting in a great deal of interactive effort to tune the geometric model to avoid these problems.
- The inability to mesh a collection of stacked hexagonal prism-shaped volumes, when the meshing schemes have been specified as e.g. sweeping surface "A" to "B" but surface "B" is meshed first. This forced the generation of the mesh for each reference assembly type (fuel, control, etc.), before the assembly was copied and moved in the hexagonal lattice of assemblies. This generation method was sufficient for this core only because of the separation of assemblies by a thin sodium region.

After struggling with these problems, it was decided to use CUBIT only for surface mesh generation. A standalone tool was used to sweep this surface mesh into the third dimension. The surface mesh generated by CUBIT in this process is shown in Figure **??**. While this process was sufficient

for this problem, it depended on the mesh sweep being a pure translation. Clearly, this approach will not be sufficient for more complicated reactor models.

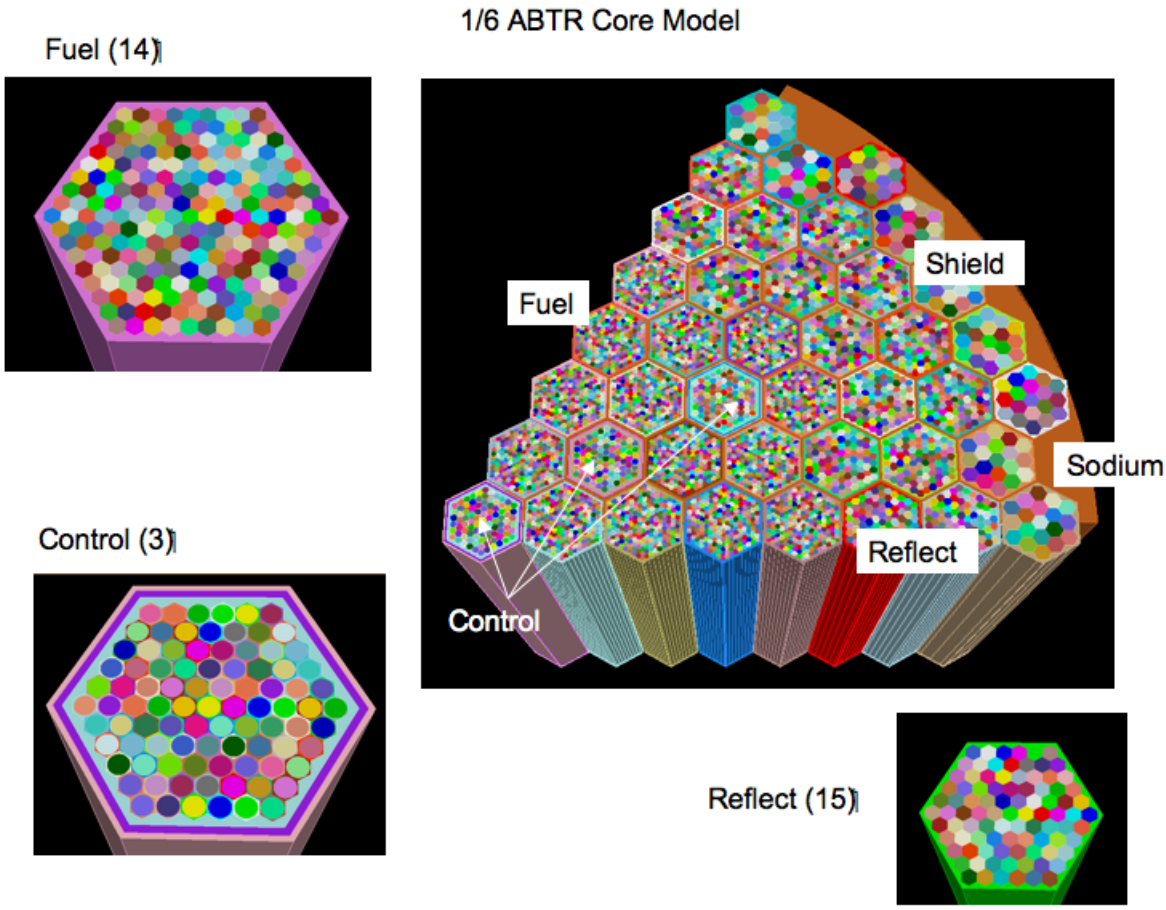

Figure 5: **The ABTR core model developed in CUBIT. This model contains 5744 distinct 3D volumes and almost 50,000 surfaces.**

6.2 Wire-Wrap Fuel Pin Bundle

An important part of the SHARP reactor simulation effort is modeling sodium flow through a lattice of wire-wrapped fuel pins. Preliminary simulations of this problem have already yielded a better understanding of fluid

and heat transfer in these assemblies [1]. The geometric model used in [1] modeled the wire-fuel pin interface using a fillet, to facilitate mesh generation. A key question is whether that fillet strongly affected the results. Furthermore, we now need to extend these simulations to the full 217-pin assemblies, possibly including geometric detail in the inlet and outlet regions which may affect the flow. Therefore, we also attempted to develop models for this problem in CUBIT.

The geometry and surface mesh developed for a 7-pin hexagonal lattice of wire-wrapped fuel pins is shown in Figure 6. Neighboring pins have mesh which is inclined in opposite directions. This makes it impossible to rotate the mesh going down one pin while having it conform to the mesh in a neighboring pin. As an alternative, we explored the generation of a mesh where the interfaces between neighboring pins was non-conformal, that is, where the mesh did not match between the regions. This mismatch can be accounted for in the fluid flow solution, at some cost in terms of accuracy and solution time. The 3-dimensional mesh for this problem is shown in Figure 7.

We generated this geometry entirely inside CUBIT, but with some difficulty associated with the helical sweep. Again, despite the geometry being a relatively simple sweep and rotate, CUBIT's meshing algorithms had difficulty generating good-quality mesh for these models, and forced the decomposition of the model axially. This indicates to us the need for more model-specific mesh generation methods and tools. We believe that the component-based framework developed for SHARP, shown in Figure 1, will facilitate the development of these tools.

6.3 Summary

Although not insurmountable, these problems and the general difficulty of using CUBIT for these models indicate the need for a different approach to this problem, at least for fast reactor core models. This will be part of future efforts on the SHARP project.

7 Conclusions

We have demonstrated a simple lightweight software architecture for coupled steady state and slow transients calculations in a fast reactor core. The architecture implements parallel coupling of physics modules (UNIC and Nek) and places a high degree of emphasis on their autonomy via a weak coupling

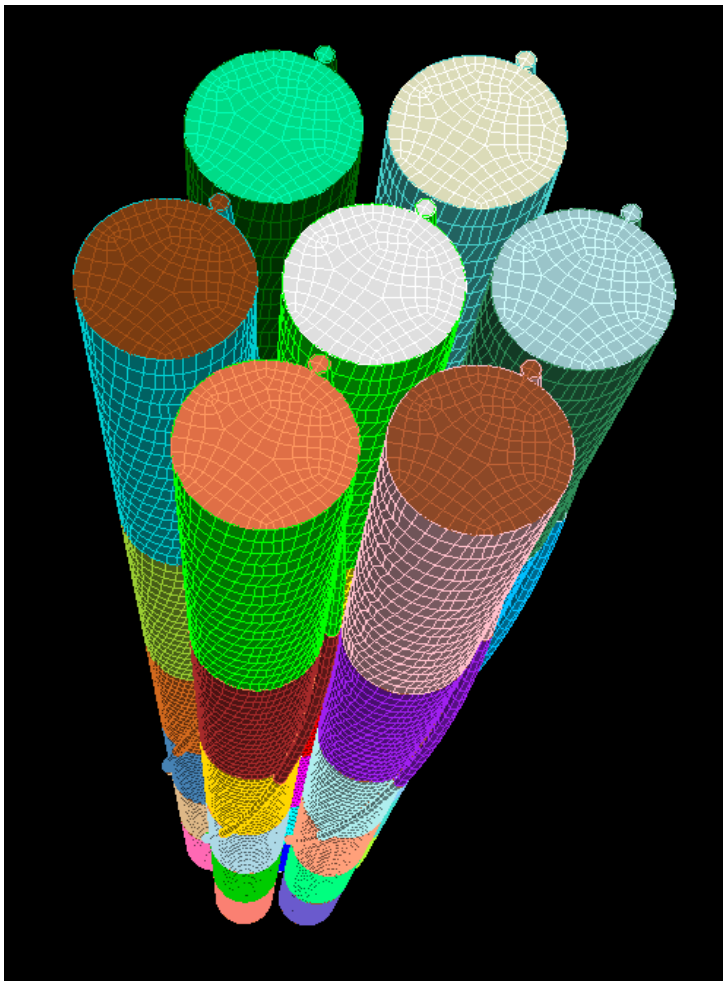

Figure 6: **7 pin wire-wrapped fuel pin lattice, showing surface mesh and without sodium.**

strategy. This approach allows for different mesh representations and prescribes relatively non-restrictive rules for incorporating legacy components. While the basic concepts of the framework are general, we outlined some specific aspects of the reactor core modeling problem that mitigate against potential performance problems associated with this technique.

This year we have implemented the framework using the initial implementations for physics and utility modules, and carried out a simple coupling benchmark. In a subsequent years, we expect to demonstrate the flexibility

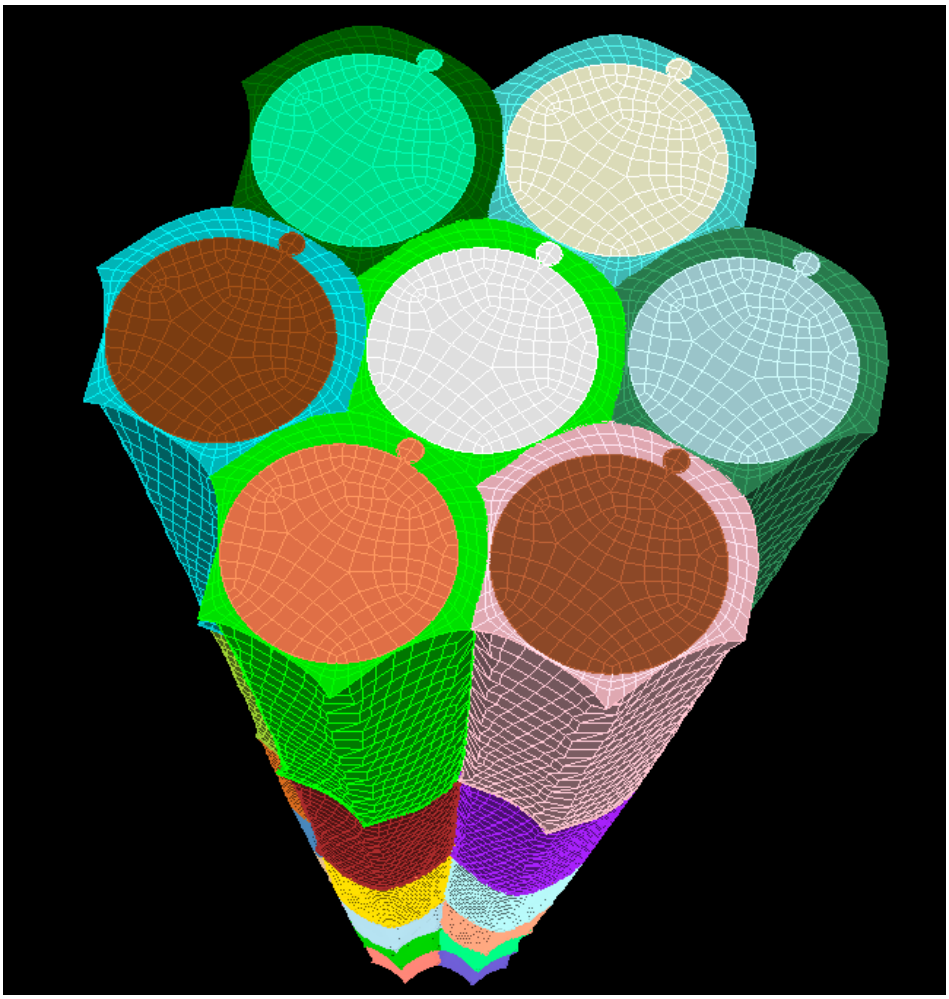

Figure 7: **7 pin wire-wrapped fuel pin lattice, with sodium region shown.**

of the framework by incorporating alternative implementations of the same physics. Additionally, the performance and scalability of the various coupling algorithms on the petascale platforms (at ANL and ORNL) will be studied in detail.

References

[1] P. Fischer, J. Lottes, A. Siegel, G. Palmiotti, "Large Eddy Simulation of Wire Wrapped Fuel Pins" *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications*, Monterey, CA, April, (2007).

[2] G. Palmiotti, M. Smith, C. Rabiti, M. Leclere, D. Kaushik, A. Siegel, B. Smith, E. E. Lewis, "UNIC: Ultimate Neutronic Investigation Code" *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications*, Monterey, CA, April, (2007).

[3] "Spallation Neutron Source: The next-generation neutron-scattering facility in the United States," http://www.sns.gov/documentation/sns_brochure.pdf (2002).

[4] R. Meyers et. al, "SNL Implementation of the TSTT Mesh Interface," *Proceedings of 8th International conference on numerical grid generation in computational field simulations*, Honolulu, HA, June 2-6, 2002.

[5] P. Fischer, G.W. Kruse, and F. Loth, "Spectral Element Methods for Transitional Flows in Complex Geometries," *J. Sci. Comput.*, **17**, pp. 81-98 (2002).

[6] "MOAB, a Mesh-Oriented datABase" http://cubit.sandia.gov/MOAB/

[7] M. Brewer et. al, "The Mesquite Mesh Quality Improvement Toolkit," *Proceedings of 12th International Meshing Roundtable*, Santa Fe, NM, September 14-17 2003, pp. 239-250

[8] K. Devine et. al, "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, **4**, pp. 90-97 (2002)

[9] K. Devine et. al, "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, **4**, pp. 90-97 (2002)

[10] T. D. Blacker et al., "CUBIT mesh generation environment, Vol. 1: User's manual", SAND94-1100, Sandia National Laboratories, Albuquerque, New Mexico, May 1994, http://cubit.sandia.gov/release/doc-public/Cubit_UG-4.0.pdf

[11] Y. I. Chang and P. J. Finck and C. Grandy, "Advanced Burner Test Reactor Preconceptual Design Report", ANL-ABR-1 (ANL-AFCI-173), Argonne National Laboratory, Argonne, IL (2006)

Nuclear Engineering Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 208
Argonne, IL 60439-4842

www.anl.gov

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC