

Software Design of SHARP

A. Siegel, T. Tautges, A. Caceres, D. Kaushik, and P. Fischer

Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL
Mathematics and Computer Science Division
siegela@mcs.anl.gov tautges@mcs.anl.gov acaceres@mcs.anl.gov kaushik@mcs.anl.gov
fischer@mcs.anl.gov

G. Palmiotti, M. Smith

Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL
Nuclear Engineering Division
gpalmiotti@anl.gov msmith@anl.gov

J. Ragusa

Texas A&M University
College Station, TX
Department of Nuclear Engineering
ragusa@ne.tamu.edu

ABSTRACT

SHARP (Simulation-based High-efficiency Advanced Reactor Prototyping) is a modern suite of codes to simulate the key components of a fast reactor core. The SHARP toolkit is organized as a collection of modules, each representing the key components of the physics to be modeled – neutron transport, thermal hydraulics, fuel/structure behavior – together with pre and post-processing for geometry definition, mesh generation, visualization, user interface, etc. The physics models are designed to make minimal possible use of lumped parameter models, homogenization, and empirical correlations in favor of more direct solution of the fundamental governing equations, when sufficient computing resources are available. Thus, one of the key design goals is to effectively leverage leadership class computing resources – viz. BG/P and Cray Supercomputers that are on the current trajectory to delivering sustained petaflops performance. Further, the nature of the physical problem to be investigated will require either strong or weak coupling between some or all of the existing modules (e.g. operator split vs. fully coupled), while multiple implementations of each physics module, representing different algorithms, will also be required (e.g. deterministic versus Monte Carlo) for verification and to explore different physical regimes. Accomplishing these goals in the context of ultra-scalable architectures and multidisciplinary and possibly distributed development teams is a daunting task. In this paper we explain our initial lightweight and loosely coupled framework, its initial design, and a number of current open research questions in this area.

Key Words: Nuclear Reactor simulation, coupling, scientific software design

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 paper 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.

Such a sophisticated project is unlikely to succeed without considerable forethought into the software architecture of the overall system. We are well aware of the pitfalls of one-size-fits-all frameworks that are overly general and focus too much on software abstraction; that said, the other extreme is equally unproductive. We instead advocate a more agile and balanced physics-based approach. This first step is to hypostatize a (somewhat artificial) boundary among different components of the system that are then developed independently and brought together by a defining rules, standards, and a set of shared utilities based on these formalisms (the *framework*). This is critical to manage complexity, minimize redundancy of effort, facilitate future change, and make the code more readily comprehensible to third party and/or in-house developers.

In this paper 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. 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, mesh mapping, 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:

$$\dot{\mathbf{f}} = \mathbf{L}\mathbf{f} + \mathbf{N}\mathbf{f} \quad (1)$$

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

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

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:

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

In (3), v 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 is 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:

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

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):

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

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):

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

$$\nabla \cdot \rho_0 \vec{u} = 0 \quad (7)$$

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 independently 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 and 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

The general approach we take is to 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 paper only the ITAPS mesh interface is described. The implementation of this interface we are using is MOAB ([4]).

The data model defined in IMesh 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 Entitys, 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 IMesh 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 effort required to attach 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.

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 a previous section. This domain can be presented 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 (3.3.3). 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 (3.3.3) 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 (3.3.3). 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 explore.

This use case is depicted in (3.3.3) 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.

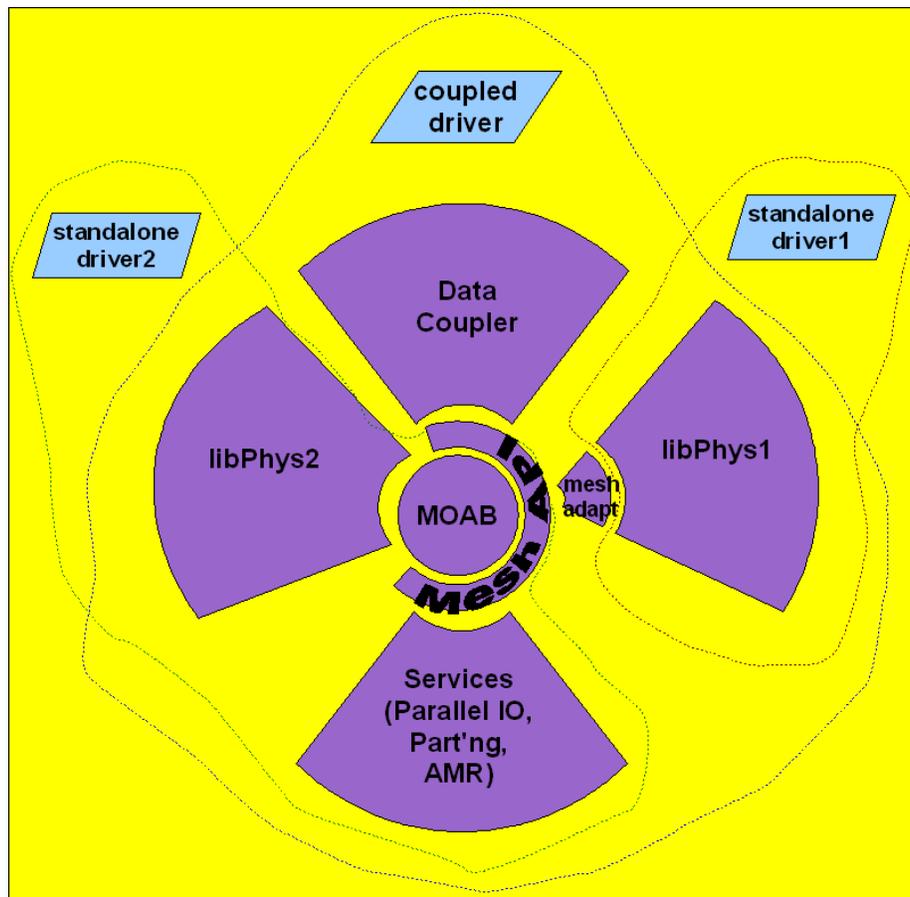


Figure 1. Interrelationship among the components of MOAB

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. One thing this does require is that the mesh for both physics modules is available through that API. In effect, this forces the representation of those meshes to be in the same implementation (in our case, MOAB). However, this is desirable from the standpoint of computational efficiency too. For example, 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 (3.3.3) depicts a coupled driver incorporating all of these design elements. In this code, libPhys1 retains its native data representation, using a 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 Details

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 $\Sigma, \rho_e, \rho_f, T_c, T_f$

Step 2: Update ϕ on Ω_n using (3) and compute q''' from ϕ using (4)

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

Step 4: Update values of ν and α on Ω_{th}

Step 5: Update $T_f, T_c,$ and ρ_c on Ω_{th} using q''' as source term

Step 6: Update Σ on coarse mesh subregion of Ω_n using new values of T_f, T_c and ρ_c

Figure 4.1 shows how this algorithm is realized in terms of the different modular components of the code. Note that, as described in the next section, 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). This is covered in more detail in the next section. 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 tin

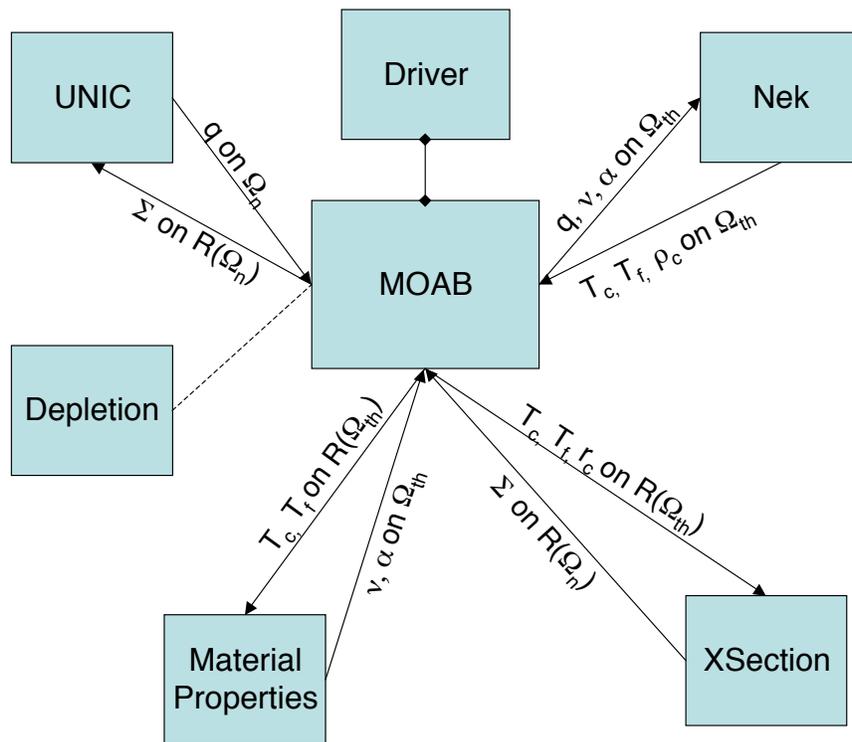


Figure 2. Depiction of the individual modules of SHARP and their interrelationships. Ω_n denotes the UNIC mesh, Ω_{th} the Nek mesh and R is a linear restriction (coarsening) operator that represents entities on homogenized regions of the mesh.

4.2. Data Volume

As shown in Figure 4.1, we need to exchange data proportional to the size of coarse mesh (number of vertices) between neutronics module (UNIC) and thermal hydraulics module (Nek). 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, the data exchanged is relatively quite minimal compared 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 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. Framework Configuration and Testing

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.

5.1. Module API

A typical API for a physics module *phys* includes functions such as:

`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 MOAB/TSTT. If the module uses MOAB for its internal representation, this is a no-op.

`phys_import()` Read from MOAB/TSTT the parts of the solution that are calculated by other modules. For example: if *phys* is a thermal-hydraulics module, read in the heat generation computed by the neutronics module.

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

Greater granularity is of course possible, but the above functions represent the minimum functionality expected from most modules.

When data needs to be transferred between two modules, the SHARP driver calls their respective import/export functions; if the two modules use different meshes, the driver calls a MOAB function to map the data in between the export and import steps.

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 makes 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.

Although we have not yet encountered issues with name clashes – two modules defining functions or global variables with the same name – we expect to be able to write simple tools to automatically detect and correct these.

Other issues are anticipated to appear, 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. Testing

Our framework is designed to allow for a comprehensive testing system which we are in the process of implementing, using both unit and integrated testing.

Unit testing occurs at the level of the module APIs: a test will set up a context, call an individual function, and verify the output. The more fine-grained the API, the more stringent testing can be. Here, the use of IMesh simplifies testing since we can access the output of different modules' functions through a single interface.

The other half of our testing strategy is integrated testing, which treats a SHARP executable as a black box. Typical types of tests that fall under this category are scaling, performance and regression.

6. 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 and places a high degree of emphasis on their autonomy via a weak coupling strategy that 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. We have implemented the framework with initial implementations for physics and utility modules and carried out a simple coupling benchmark. In a subsequent paper, we aim to demonstrate the

flexibility of the module to incorporate alternate implementations of the same physics as well as examine in detail the scalable performance of these techniques.

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)