INL/RPT-23-73151

Usability and Optimization Improvements in MOOSE

June 2023

Nuclear Energy Advanced Modeling and Simulation Milestone, June 2023

Alexander Lindsay[1], Roy Stogner[1], Guillaume Giudicelli[1], and Cody Permann[1]

[1]*Idaho National Laboratory*

Idaho National Laboratory

INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance, LLC

INL/RPT-23-73151

Usability and Optimization Improvements in MOOSE

Nuclear Energy Advanced Modeling and Simulation Milestone, June 2023

Alexander Lindsay[1], **Roy Stogner**[1], **Guillaume Giudicelli**[1], **and Cody Permann**[1]

[1]**Idaho National Laboratory**

June 2023

Idaho National Laboratory
Computational Frameworks
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Page intentionally left blank

SUMMARY

The Multiphysics Object-Oriented Simulation Environment (MOOSE) framework is a foundational capability used by the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program to create over 15 different simulation tools for advanced nuclear reactors. Due to MOOSE's broad use, improvements to the framework in support of modeling and simulation goals are critical to the program. Such improvements can take many forms, including optimization, improved user experience, streamlined application programming interfaces (APIs), parallelism, and new capabilities. The work described in this report was conducted in direct support of the simulation tools and has already been deployed. The capabilities were implemented in the same order as they are covered in this report: multiple nonlinear systems in the same input file, implementation of generic field transfers and other transfer system enhancements, support for stateful material property redistribution for adaptivity and distributed meshes, and dynamic linking and loading of individually compiled applications. These four additions are fundamental capabilities that will be leveraged by many NEAMS applications.

Page intentionally left blank

CONTENTS

FIGURES

TABLES

ACRONYMS

ANL	Argonne National Laboratory
API	Application Programming Interface
INL	Idaho National Laboratory
MOOSE	Multiphysics Object-Oriented Simulation Environment
MPI	Message Passing Interface
NBX	Non-Blocking Consensus
NEAMS	Nuclear Energy Advanced Modeling and Simulation
OS	Operating System
TA	Technical Area
TIMPI	Templated Interface to MPI

Page intentionally left blank

1. INTRODUCTION

The Nuclear Energy Advanced Modeling and Simulation (NEAMS) program aims to develop simulation tools in support of the nuclear industry. These tools are meant to accelerate reactor design, licensing, demonstration, and deployment. The program is split into five Technical Areas (TAs): Fuel Performance, Thermal Fluids, Structural Materials and Chemistry, Reactor Physics, and Multiphysics Applications. The "physics" TAs involve delivering simulation tools to vendors, laboratories, and licensing authorities, whereas the Multiphysics Applications TA creates foundational capabilities for the program and exercises the tools in order to ensure their usability for the intended purposes.

Reactors are inherently multiphysical: heat conduction, neutronics, solid mechanics, fluid flow, chemistry, and material evolution all combine to create a complex system that is difficult to simulate. To tackle that problem, the NEAMS program utilizes the Multiphysics Object-Oriented Simulation Environment (MOOSE) platform [2] from Idaho National Laboratory (INL) to develop interoperable physics applications. Over 15 MOOSE-based physics applications are being developed within the program, with at least one in every TA. Each physics application focuses on a particular aspect of reactor simulation (e.g., Bison [3] for nuclear fuel performance and Griffin [4] for neutronics).

It is therefore critical to the program that MOOSE continues to be enhanced and supported. Capabilities and optimizations made to the framework are instantly available to all NEAMS applications, making for a large return on investment.

1.1 MOOSE

The MOOSE platform enables rapid production of massively parallel, multiscale, multiphysics simulation tools based on finite element, finite volume, and discrete ordinate discretizations. This platform was developed as open-source software on GitHub [5] and utilizes the Lesser General Public License v2.1, thus allowing for a large amount of flexibility. It is an active project, with dozens of code modifications being merged weekly.

The core of the platform is a pluggable C++ framework that enables scientists and engineers to specify all the details of their simulations. Certain interfaces include finite element/volume terms,

boundary conditions, material properties, initial conditions, and point sources. By modularizing numerical simulation tools, MOOSE allows for an enormous amount of reuse and flexibility.

Beyond its core framework, the MOOSE platform also provides myriad supporting technologies for application development. This includes a build system, a testing system for both regression and unit testing, an automatic documentation system, visualization tools, and many physics modules. The physics modules are a set of common physics utilizable by application developers. Some of the more important modules are the solid mechanics, heat conduction, fluid flow, chemistry, and phase-field modules. All of this automation and reuse accelerates application development within the NEAMS program.

1.2 libMesh

Underpinning the MOOSE framework is the libMesh finite element framework [6]. libMesh is an open-source software library originally developed at the University of Texas at Austin's CFDLab. It provides an enormous amount of numerical support, including the mesh data structures, element discretizations, shape-function evaluations, numerical integration, and interfaces to various linear and nonlinear solvers (e.g., PETSc) [7]. MOOSE and libMesh were developed in lockstep with each other, with any change to one being immediately tested against the other. This symbiotic relationship has worked extremely well over the 14 years of MOOSE development, partly due to INL having employed multiple libMesh developers over the years.

2. MULTIPLE NONLINEAR SYSTEMS

Being able to split a monolithic nonlinear system of equations into smaller subsystems within a single input file has been a desire of MOOSE application developers since at least 2017. Solving the smaller subsystems within a single input as opposed to multiple inputs with the `MultiApp` system saves N - 1 mesh copies and input files, where N is the number of subsystems. Additionally, computational `Transfer` costs are avoided. Finally, system splitting allows application developers to create arbitrary solution schemes when Newton is not desirable. Given these features, a multiple nonlinear system capability was implemented in MOOSE pull request #22226.

An example of a solution scheme enabled by the multiple nonlinear system splitting capability is the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm [8]. SIMPLE takes a pressure-based discretization of the Navier-Stokes equations and converts the "pressure" equation (divergence-free constraint) from having no or small diagonal compared to the velocity degrees of freedom to appearing as a diffusion equation with significant diagonal character. This is done simply by splitting the systems that the pressure and velocity degrees of freedom belong to. Subsequently, incredibly efficient multigrid preconditioning techniques can be applied to the SIMPLE form of the "pressure" equation. SIMPLE has recently been implemented in MOOSE using the multiple nonlinear system capability. Application of SIMPLE to a 3D T-junction flow problem with a Reynolds number of 220 is shown in Figure 1.

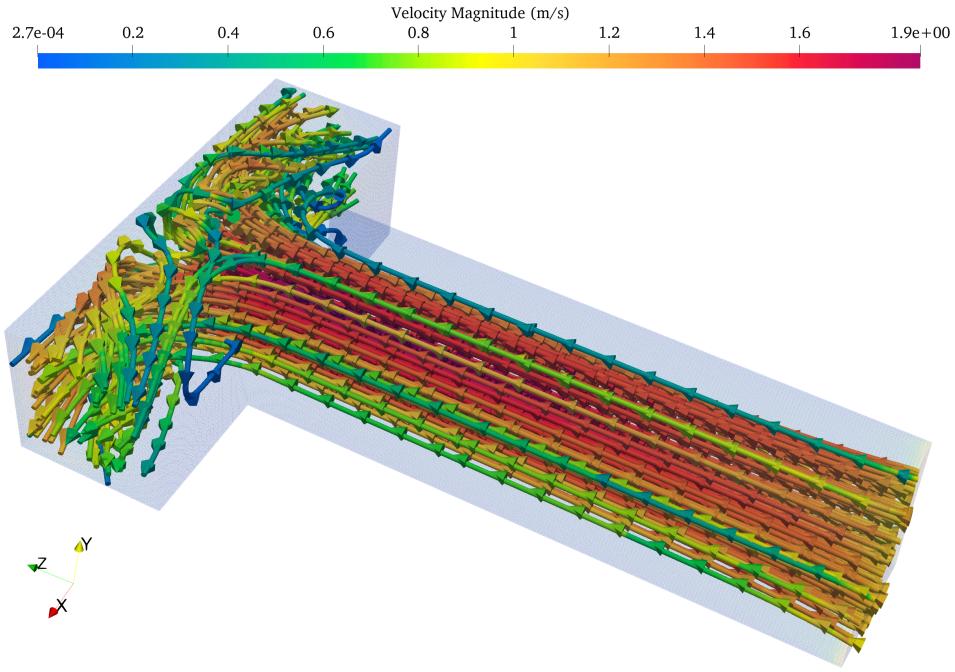


Figure 1: Results for a T-junction flow simulation with a Reynolds number of 220. The Navier-Stokes equations were solved using the SIMPLE algorithm implemented via the new multiple nonlinear system capability in MOOSE.

The multiple nonlinear system capability has been leveraged to solve forward and adjoint problems in an inverse optimization context within a single input. Details about the inverse optimization procedure in MOOSE can be found on the framework documentation website. In

the example we will show here, we are solving a force inversion problem whose algorithmic procedure is described in Figure 2. Given an initial guess for the parameterized heat source (q_v), we run the forward problem (e.g., the physics model) to compute the forward solution T. We then solve the adjoint equation

$$\left(\frac{\partial \mathcal{R}}{\partial T}\right)^T \lambda = -\left(\frac{\partial f}{\partial T}\right)^T \tag{1}$$

where $\mathcal{R}$ denotes the residual vector of the forward problem and f is the objective function which we are minimizing (denoted as J in Figure 2 but renamed here to avoid confusion with the traditional MOOSE notation for the Jacobian). Because we are running the forward and adjoint solves from the same input file we have very easy access to $(\partial \mathcal{R}/\partial T)^T$; we simply take the transpose of the Jacobian computed in the forward solve. After performing the adjoint solve, the gradient of the objective function can be computed via Equation (2).

$$\frac{df}{dp} = \lambda^T \frac{\partial \mathcal{R}}{\partial p} \tag{2}$$

Armed with the objective function and its gradient we can perform an iteration of a TAO [9] gradient-based solve to produce a new guess for the heat source q_v. This iteration proceeds until stopping criterion have been achieved.

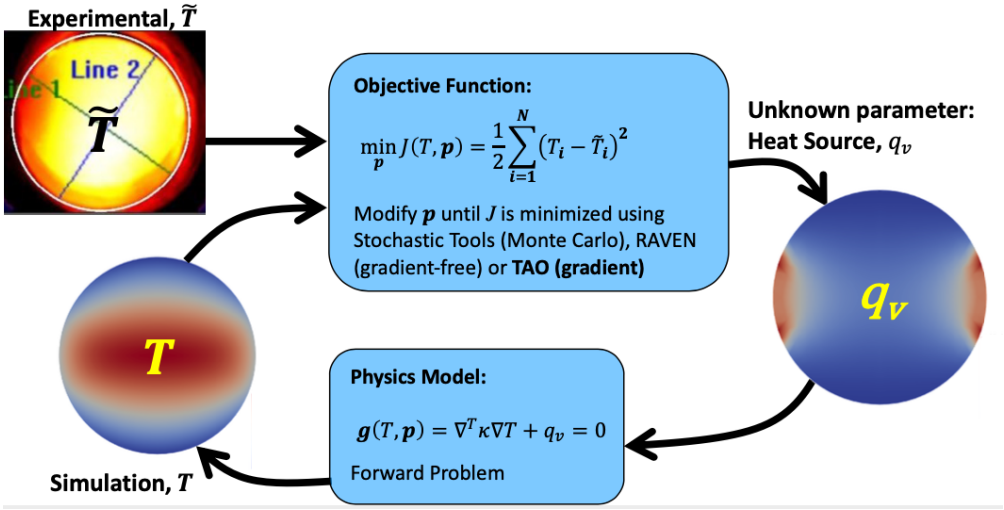


Figure 2: Algorithmic procedure for determining a heat source or body force distribution (here denoted by q_v) that will minimize the difference between the field computed by a physics model (T) and measurements ($\tilde{T}$).

As an example of applying the optimization algorithm, we consider a model problem in which the true forward solution is a Gaussian spot which circles around a square domain. We compute a heat source that will produce the Gaussian heat spot solution using the method of manufactured solutions (MMS). In the optimization algorithm, we parameterize the heat source with four quadrants in space and eleven points in time, for a total number of parameters equal to forty-four. We can denote a given parameter value as q_{ij} where i denotes the spatial index with allowed values between 0 and 3 and j denotes the temporal index with allowed values between 0 and 10. When solving the forward problem, the heat source q in a given spatial quadrant i at a given time t is equal to a linear interpolation between bracketing times t_j and t_{j+1}, for example

$$q = \frac{t_{j+1} - t}{t_{j+1} - t_j} q_{ij} + \frac{t - t_j}{t_{j+1} - t_j} q_{i(j+1)}. \tag{3}$$

Figures 3, 4, 5, and 6 show results at four different time stages. The left hand side of each figure corresponds to "true" values, with the top left of each figure corresponding to the true heat source and the bottom left corresponding to the true temperature distribution. The right hand side shows the optimization algorithm results, with the top right showing the spatial distribution of the computed heat source (the partitioning of space into four quadrants is apparent), and the bottom right showing the computed temperature distribution. A more accurate representation of the true values could be produced with a finer parameterization (e.g., more parameter values in space and time).

While the results shown here were achievable before the advent of the multiple system capability, the optimization module developers have described the transition to the new capability as a paradigm change in usability. One developer described it as being akin to the introduction of automatic differentiation within MOOSE for eliminating the need of application developers to manually write Jacobians.

3. TRANSFER ENHANCEMENTS

The `Transfers` system is at the core of most multiphysics simulations using simulation codes within the MOOSE simulation environment. After each code performs their own physics solve, they exchange information, whether scalars or fields on the mesh, using Transfers.

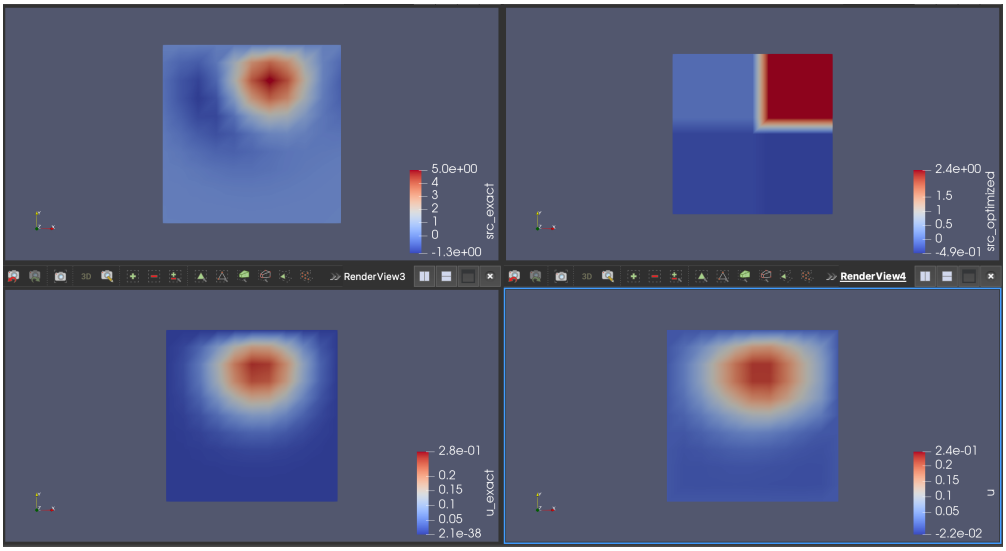


Figure 3: At time = .02, clockwise from top left, the true heat source corresponding to the "exact" temperature solution used for computing measurement error, the heat source computed via the optimization algorithm, the temperature distribution corresponding to the forward solution with the optimized heat source, and the "exact" temperature solution.

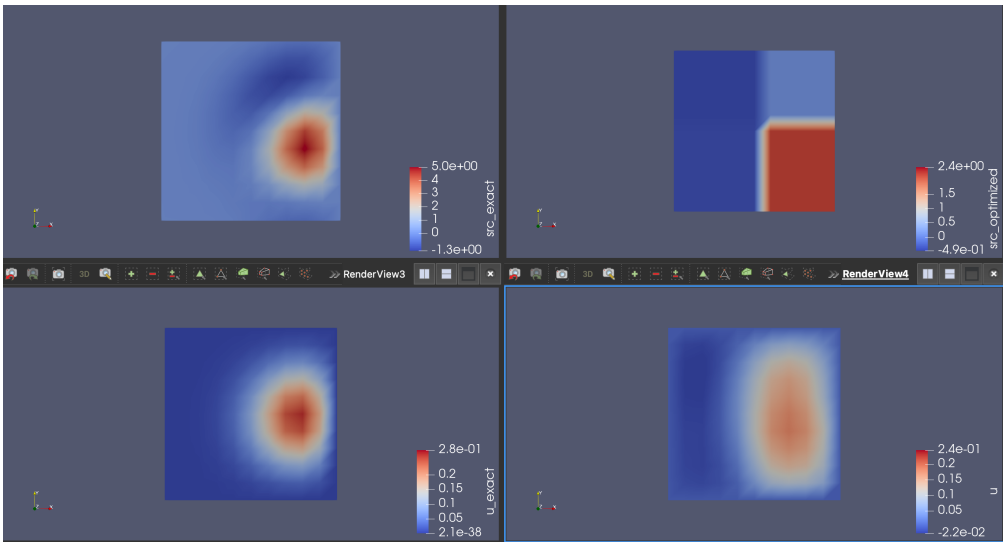


Figure 4: At time = .26, clockwise from top left, the true heat source corresponding to the "exact" temperature solution used for computing measurement error, the heat source computed via the optimization algorithm, the temperature distribution corresponding to the forward solution with the optimized heat source, and the "exact" temperature solution.

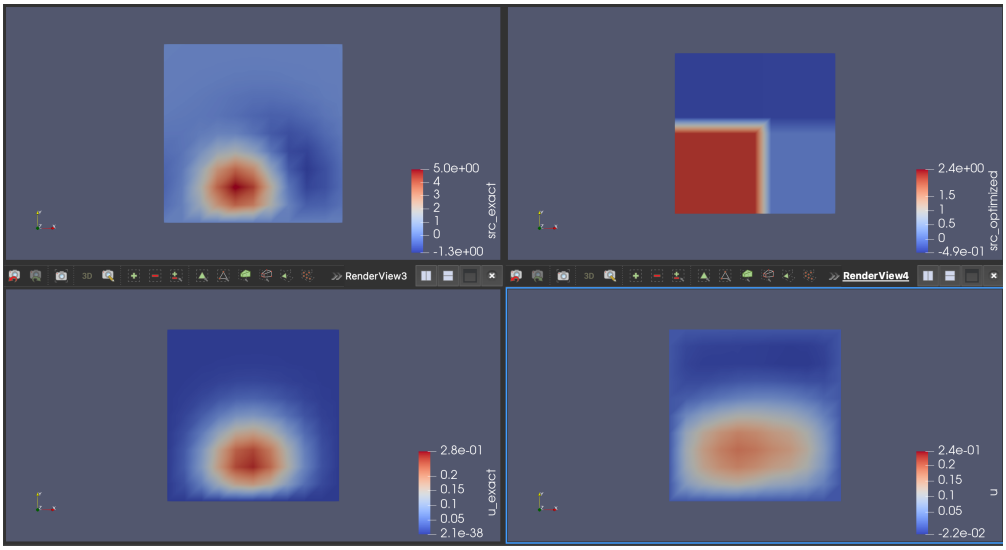


Figure 5: At time = .51, clockwise from top left, the true heat source corresponding to the "exact" temperature solution used for computing measurement error, the heat source computed via the optimization algorithm, the temperature distribution corresponding to the forward solution with the optimized heat source, and the "exact" temperature solution.

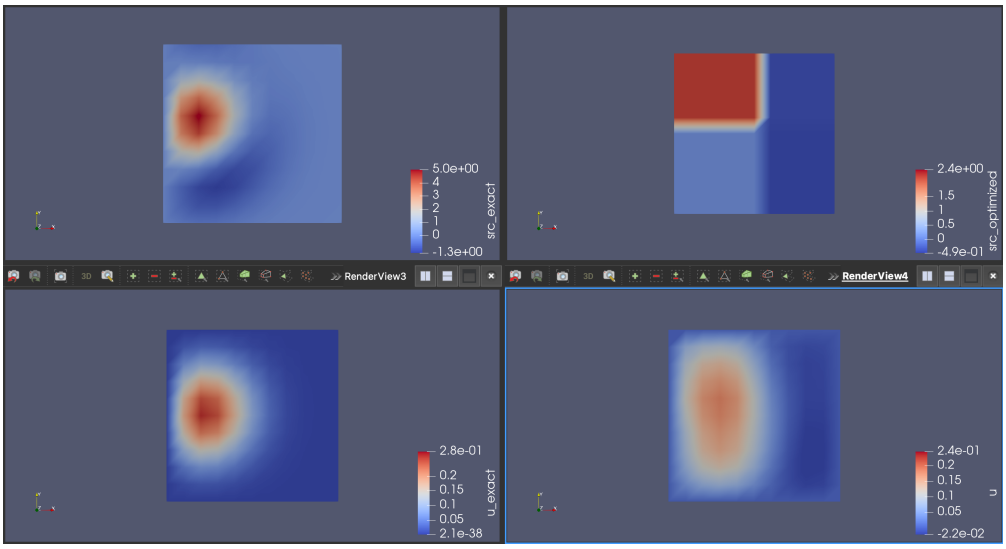


Figure 6: At time = .76, clockwise from top left, the true heat source corresponding to the "exact" temperature solution used for computing measurement error, the heat source computed via the optimization algorithm, the temperature distribution corresponding to the forward solution with the optimized heat source, and the "exact" temperature solution.

Transfers, while conceptually straightforward, are complex in nature due to the sheer number of framework feature combinations. For example, a straightforward field-to-field transfer where the child's domain perfectly matches the parent's domain can be complicated by numerous factors, including: different meshes, different discretization representation (shape functions), different parallel partitions and distribution. Handling the sheer number of different transfers while addressing the different combinations of framework features is a daunting task. Nevertheless, numerous robust algorithms have been developed in MOOSE, MOOSE-based, and MOOSE-wrapped applications to enable these exchanges.

The Transfers in MOOSE can be grouped in five categories:

- Field to field
- Scalar to field
- Field to scalar
- Scalar to scalar
- Arbitrary user object generating spatial value to field.

In fiscal year 2022, siblings transfers were implemented for the scalar-scalar transfers as well as the direct-copy field transferIn fiscal year 2023, we implemented:

- An execution point at the beginning and end of multiphysics fixed-point iterations, allowing for less frequent transfers or more flexible multiphysics schemes. See Section 3.1.
- General field transfers, which include siblings transfer capabilities and many others. See Section 3.2.
- Automated diagnostics of nearest-node transfer issues. See Section 3.2.2.

3.1 Execution Timing of Transfers

Corresponds to MOOSE pull request: #23065

The default execution schedule for transfers in MOOSE is the one that maximizes the "freshness" of data exchanged before it is used. For example, data sent from the main application

to a child application is sent right before the child app executes, and similarly data sent from the child app to the main app is received by the main app immediately after the child app executes. By never lagging information, fixed point iterations converge as fast as possible. It is even possible to use these up-to-date fields to design quasi-Newton fixed point iteration schemes

However, this does not cover the full spectrum of needs for multiphysics iteration schemes. For example, consider a multiphysics problem where three physics are involved, solved by three apps called 'parent', 'child1' and 'child2' respectively. While it may be desirable to couple 'parent' and 'child2' tightly, and iterate between solve and transfers until convergence, it could also be that 'parent' and 'child1' are only loosely coupled and information should be lagged, either because it does not matter or because the numerical scheme dictates it. This is represented in Figure 7, and is possible using the new `MULTIAPP_FIXED_POINT_BEGIN` and `MULTIAPP_FIXED_POINT_END` execution schedules. The boxes represent the execution of an application within a time step, while the arrows indicate a transfer. The transfers to and from 'child1' are executed at the beginning and end of the fixed point iteration. In fact, 'child1' is also executed on that schedule.

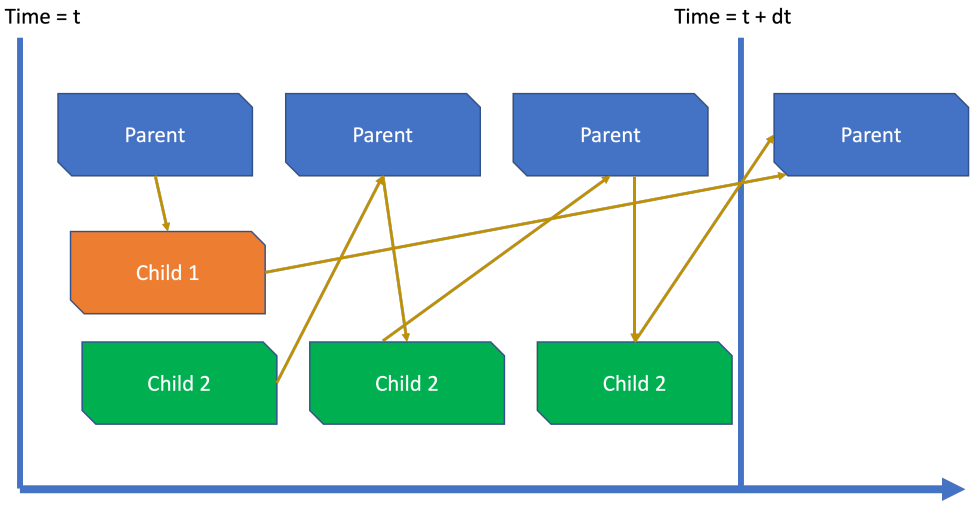


Figure 7: Tight coupling of the 'parent' and 'child2' app, with loose coupling for 'child1'.

Similarly, even with two physics involved, it could be that the numerical scheme the modeler seeks to implement dictates lagging one of the variables exchanged. It then becomes "explicit" in that quantity, while remaining implicit in the other quantities exchanged. This is schematized in Figure 8.

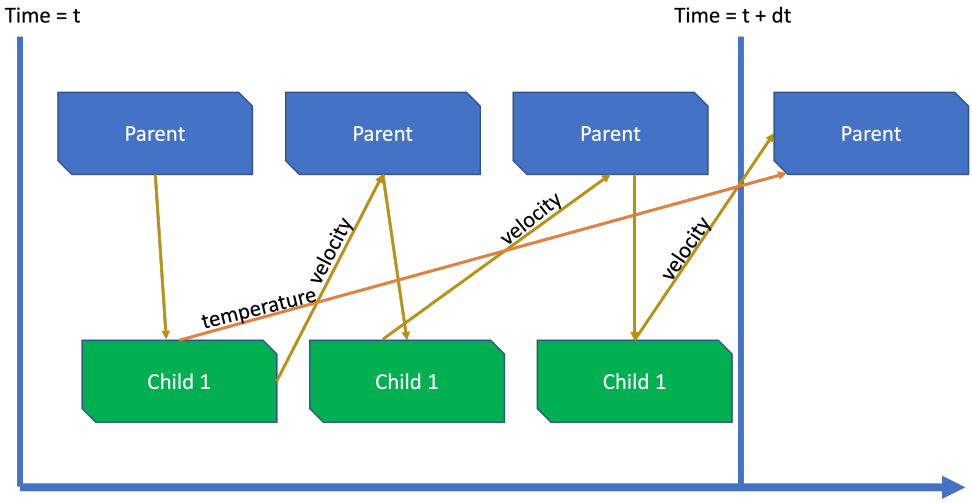


Figure 8: Tight coupling between the 'parent' and 'child1' for the velocity variable, with loose coupling for the temperature variable.

3.2 Deployment of General Field Transfers

General field transfers inherit from a base class called `MultiAppGeneralFieldTransfer`. It holds most setup and communication routines, leaving to the derived classes the charge of computing the transferred values.

A `GeneralField` transfer proceeds as follows:

1 Each process first looks to find which source application it will be talking to. This is based on geometric proximity, leveraging bounding boxes that enclose each application's domain. The bounding boxes may need to be extended using the `bounding box factor` parameter to expand the domain considered for transfers with wider stencils.

2 Then each process shares the list of target points, locations that it needs data for, with each of the processes owning the source applications of interest. These other processes then, using the behavior defined in the derived class, compute the local value of the source variable. They also share geometric information about where this evaluation of the source variable was located, which, for example, could then be used for interpolation.

3 Finally, the data received is placed in the solution vector of the target variable(s). This is performed using a local projection with the shape function of the target variable(s) to support

higher order variables. The libMesh `GenericProjector` is leveraged for that purpose.

3.2.1 Features supported

All transfers derived from this base class should be able to support:

- Block restriction in both the source and target application.
- Boundary restriction in both the source and target application.
- Arbitrary number of parallel processes for both the source and target application.
- Support for replicated and distributed meshes in both applications involved.
- Transfers between parent and child applications.
- Transfers between sibling applications (child to child).
- Transfers between parent and multiple child applications in different locations.
- Transfers to and from a displaced mesh.
- Transfers of nodal and elemental variables.
- Transfers between variables of different finite element/volume family and type.
- Transfers between regular and array variables.
- Transfers from multiple variables to multiple variables.
- Interpolation and extrapolation transfers, as defined by the derived class.
- Detection of indetermination due to source points equidistant to a target point.
- Limitation of transfer source to the nearest position (see the Positions system documentation) of target point.

3.2.2 Automatic diagnostics

The derived classes of `MultiAppGeneralFieldTransfer` may keep track of indetermination in origin values. In this event, a single value is still selected, usually the one from the process with the lowest rank. These indeterminations occur when, for example:

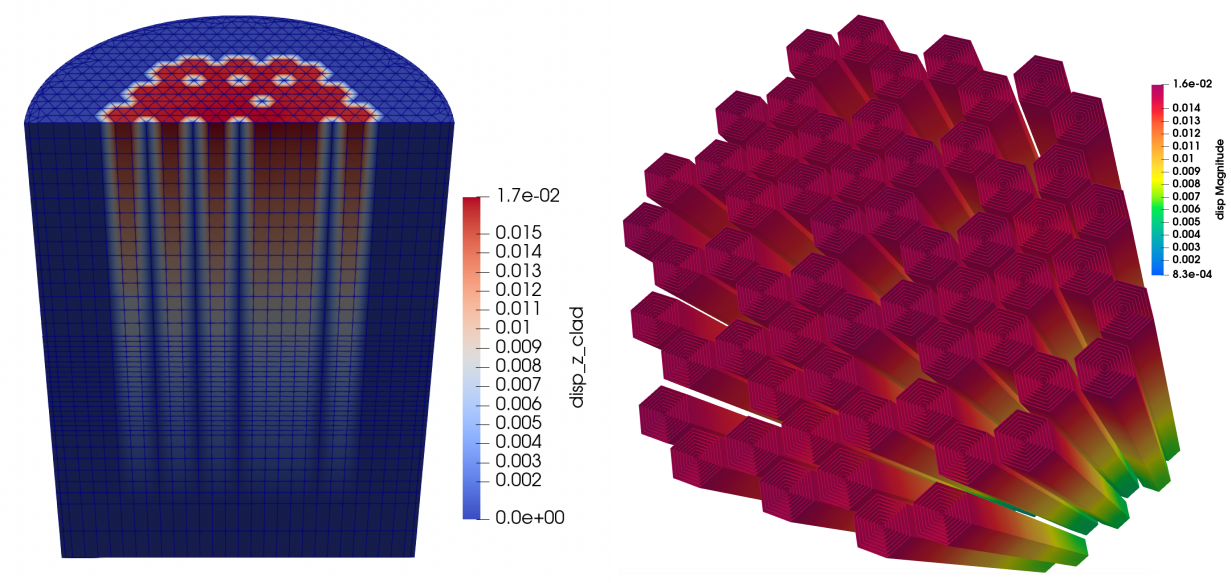
- Multiple points in the origin mesh(es) are equidistant to the target location for nearest-node type transfers.
- Multiple child apps have an equidistant point to the target location for nearest-node type transfers.
- Multiple child apps can compute a valid value for a target location because their meshes overlap.

`MultiAppGeneralFieldTransfer` itself keeps track of indetermination by examining received values for each target point. This occurs when multiple values are received for a single target point. Some transfers examine the distance between origin and target points to select a value, but this can still lead to indetermination if the distances are the same.

Mesh, block, or boundary restrictions can sometimes be used to alleviate these indeterminations in the origin values. Other times, either remeshing one of the applications or using the child application positions parameter to create a very small offset can help remove the indetermination.

3.2.3 Deployment

The `GeneralFieldTransfers` have been deployed on simulations of the Advanced Breeder Test Reactor (ABTR) by Javier Ortensi, Stefano Terlizzi, and Jackson Harter to transfer displacements from the thermo-mechanical simulations to the neutronics simulation. These are shown in Figure 9. The displacement is larger near the top of the core, driven by the axial component. The user object transfer is used to transfer assembly-averaged displacements to the neutronics simulation. The `GeneralFieldTransfer` version of the user object supports distributed meshes, while its predecessor did not. This enabled using this memory-intensive, partially heterogeneous neutronics model.



(a) Clad displacements in source geometry

(b) Target geometry mesh displacement

Figure 9: Transferring solid displacements from homogenized thermo-mechanics (informed by pin-wise heterogeneous calculations) to a partially heterogeneous neutronics ABTR model (with permission from J. Ortensi, S. Terlizzi & J. Harter)

The `GeneralFieldNearestNodeTransfer` has been used for wire-wrapped bundle simulations [1]: one to validate the Cardinal application against a freon-cooled experiment and another for Advanced Burner Reactor (ABR)-type assemblies. The main quantities transferred in those simulations are shown in Figure 10. The nearest-node approach is selected because the meshes are distinct and, most importantly, do not even overlap near the pin surfaces. This prevents the use of simple shape function evaluations (the true value of the variable at a given point), and necessitates a different transfer algorithm. The use of nearest-node values is not inherently conservative, so Cardinal has additional logic to re-normalize the heat flux for conservation. The results of these high fidelity coupled computational fluid dynamics, neutronics and thermo-mechanics simulations are shown in Figure 11.

The `GeneralFieldTransfers` have been deployed on the Virtual Test Bed [10]. For every input using the traditional transfers, the `GeneralField` equivalent of the previously used transfers was able to reproduce the results with no failure in the test suite. The test suite only examines selected integral quantities rather than multi-dimensional field variable distributions; however, it is likely that the conservation of these integral quantities is synonymous with correctness in this case.

General field user object transfers have also been deployed on two advanced reactor

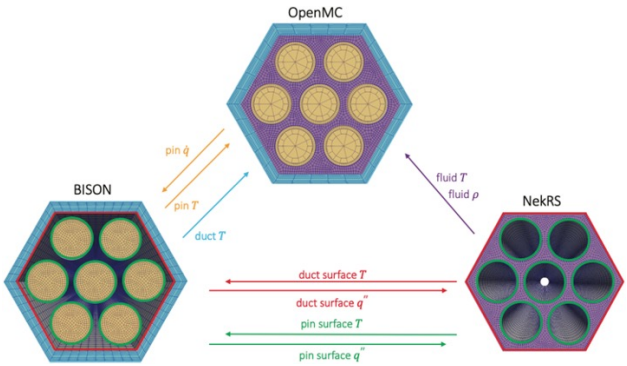
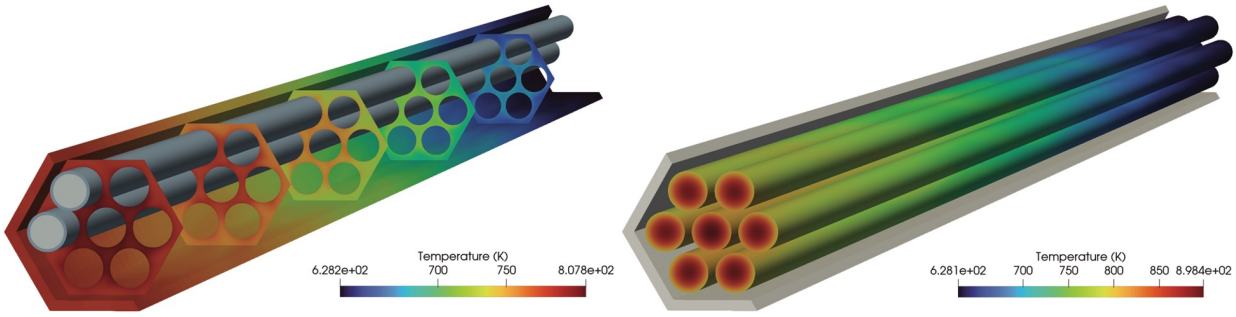


Figure 10: MultiApp setup and transfer scheme for the wire-wrapped bundle simulations using Cardinal, extracted from [1] with permission. The transfers used for temperature and heat fluxes are `MultiAppGeneralFieldNearestNodeTransfer`.



(a) Duct and fluid temperatures

(b) Solid pin temperatures

Figure 11: ABR-pin high-fidelity multiphysics Cardinal simulation results, extracted from [1] with permission.

simulation studies. At the time of this report, permission was not yet granted by the sponsors to provide more information on this use.

Optimization module Siblings transfers were deployed to the optimization module in MOOSE. In some adjoint-based optimization schemes implemented, the forward solution has to be transferred to the adjoint problem and is then used to compute the gradient update. Previously, this operation involved copying the forward problem mesh in the main application (the one that drives the optimization problem) then transferring it down to the child app solving the adjoint problem. Now this can be performed in a single step, directly from the forward to the adjoint problem.

3.2.4 Addressing user feedback

Pull request #23360 A user reported that the behavior of the general field shape evaluation transfer was not as expected on a displaced mesh problem. As a simple rectangular mesh child application was displacing in a much larger main app mesh, it was leaving a track that would remain over all time steps rather than only show values at its location during the current time step. This was fixed by changing the transfer behavior in regions where no values are found to transfer. Before this fix, the previous value of the variable would be left unchanged, which led to this permanence of variable values. Now, a constant extrapolation value is placed in all regions where no value to transfer are found. If a user wants to prevent this, they can still use "target" block and boundary restrictions to limit the parts of the mesh filled by the transfer.

Pull request #23719 After a user support meeting internal to INL, it appeared that a user was both expecting the general field transfers to provide extrapolated values and that the source of the variable values had to be block restricted. This is incompatible, as values are naturally queried outside the block restriction for an extrapolation. Therefore, the code will now error in this situation unless an extrapolation behavior is explicitly provided by the user to indicate that they are fine with this limitation.

Pull request #24196 Following several user support meetings both internal and external to INL, it became obvious to the author of these lines that no one was using the diagnostics created to

ensure the transfers were behaving in a deterministic way, and not relying on some floating point round-offs to decide which of two equidistant nodes was actually the closest. Therefore, it became the default in MOOSE. While the diagnostics are turned on by default, which may come at a performance cost, the user is encouraged to turn them off to save some computational cost if the diagnostics did not find any issue.

3.2.5 Features not supported

The following features currently cannot be supported by general field transfers as a limitation of the `MultiAppGeneralFieldTransfer` base class. The modifications to the transfers would be significant to support these features.

- Bi-directional transfers. A single transfer that sends data to and from the same application. This is typically unnecessary, as having two transfers instead of one is not a large limitation or a large performance penalty.
- Transfers between vector variables. Vector variables are not widely used in MOOSE. When they are used, an easy workaround is to copy the components one by one using auxiliary kernels, then transfer those components with a transfer object per component.

These features are currently unsupported, but could be enabled if necessary with reasonable efforts:

- General coordinate transformations. Only the positions offset of the child applications are currently supported.
- Caching optimizations for when both the target and origin mesh are constant.
- Reduction operations, sum/average/min/max, on data transferred from multiple child applications.

3.3 Miscellaneous Transfers Improvements

Pull request #23669 An internal user reported that misspelling the parameters `from_multi_app` and `to_multi_app` using, for example, `from_multiapp` and `to_multiapp` respectively while also

specifying the deprecated `direction` parameter would lead to undefined behavior of the code. The mistake was very difficult to catch because the crash happened before the unused parameter warnings, and the wrong spelling seems just as valid as the correct one. The code was improved to properly request the missing parameters with a clear error message.

Pull request #22277 Field copy transfers are used when the meshes of the main application and the child application – or two child applications since copy transfers support siblings transfer – are identical, and the variables transferred are identical as well. Variables are identical when they share the same finite element family and order. In this case, the transfer operation is limited to copying a given degree of freedom to the same degree of freedom in the other application, which is extremely fast.

In order to be able to limit this transfer to part of the mesh, we added block restriction to `MultiAppCopyTransfer`. The user may specify one or more subdomains on which the data will be transferred. Only elemental degrees of freedom are supported for this block restriction currently.

4. MATERIAL PROPERTY REDISTRIBUTION

Most MOOSE features, and hence most MOOSE simulations, are compatible with a libMesh feature designed to maximize parallel scalability of large multiphysics problems with intricate domains and solution features: load-balanced, distributed-memory-parallel mesh adaptivity. The distinction between replicated-parallel and distributed-memory-parallel meshing is illustrated in a simple two-dimensional example in Figure 12. In both the serialized and distributed examples there, three Message Passing Interface (MPI) ranks operate on a mesh, and each example uses an identical partitioning; however, in the distributed-memory parallelism the full partitioning is not visible on every rank. As the number of elements in the mesh increases, the fraction of the distributed mesh visible by each of N processors decreases asymptotically toward 1/N, and the entailed memory requirements decrease as well.

In any context, distributed-memory parallelism is necessary to achieve optimal parallel scaling of memory usage. For example, if an increase in numerical approximation fidelity requires twice as many finite elements or finite volumes in a simulation mesh, but we hope to obtain that high fidelity at a similar cost by simply using twice as many supercomputer nodes, then

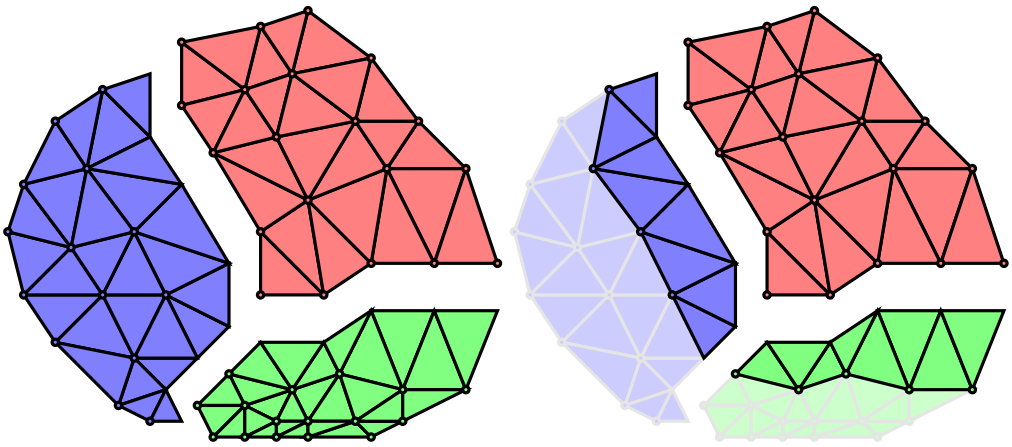


Figure 12: Left: a partitioned but serialized mesh. Each MPI rank has a "ghosted" copy of every element owned by another rank. Right: a partitioned distributed mesh, with elements visible on the "red" rank highlighted. Each MPI rank sees only its "local" elements as well as the few ghosted elements with which it will directly interact.

to maintain a similar memory-per-node limit it is necessary for each node to only keep its own small fraction of the mesh in local memory. For many simulations, this can be relatively simple: a mesh which is initially replicated on every processor is partitioned, each MPI rank is assigned one part, and elements which are not needed by each rank's part are deleted on that rank. One or more layers of "ghost" elements surrounding each part are also retained; the MOOSE `RelationshipManager` system provides a mechanism by which a MOOSE object can specify what elements need to be retained for any given partitioning, as well as what level of further connection (ghosting of algebraic solution data or entries for coupling within a Jacobian sparsity pattern) needs to be synchronized beyond geometric ghosting. By the time a simulation starts to allocate other memory-intensive structures needed for its physics, the data structures needed for its geometry are already distributed and free memory is available. For larger problems, a mesh may be generated on a high-memory system and then pre-split into distributed files, which are then loaded by the final simulation. For the largest problems, many MOOSE mesh generators are capable of creating and transforming a mesh while it is already in a distributed form, so that the full mesh need never exist in memory on a single node. Creating a coarse mesh geometry and then extruding to higher dimension or refining to higher fidelity are two examples of this sort of workflow. Once a static mesh has been distributed in one of these fashions, simulation code can make use of it in the same fashion as it would with a replicated mesh, with each rank

performing algebraic assemblies and postprocessing evaluations only on its part of the mesh and on the ghosted elements that are coupled to that part.

Further distributed parallel capabilities are employed when a mesh is not static, or is expected to change mid-simulation. This can be a requirement of a particular physical model which creates time-dependent changes in the simulation domain itself, as represented by element addition, deletion, or cutting. It can also be part of the process of adaptive mesh refinement and coarsening, where elements are subdivided to increase approximation fidelity in domain areas with particularly tricky or important solution features and previously subdivided elements might be coalesced to increase efficiency in domain areas with easily-approximated or less-important solution features. In all these cases, there might be a "natural" MPI rank whose part of a partition will receive each particular newly created element, but after sufficiently many elements have been added or removed in this fashion, the resulting partition may no longer be an efficient way to distribute simulation work. Ranks whose parts have had disproportionately many elements added will be asked to perform a disproportionately large amount of subsequent computational work, while ranks left unchanged or with elements removed will end up frequently stuck in MPI busy-wait loops, performing no useful computations, while waiting for the overloaded ranks to finish that work. The solution to this problem is repartitioning, again subdividing the mesh in an evenly balanced fashion to give each rank an even share of work in future simulation steps. However, in the case of a distributed mesh, repartitioning entails redistribution. Every element E which previously belonged to processor P and is now owned by processor Q must be communicated from P to processor Q (who might not yet even possess a ghosted copy of E in memory). All ghosted elements required for communications on E must be communicated, and all simulation data associated with these elements must also be communicated.

This last requirement was a limiting factor in MOOSE simulations for several years. Although the libMesh library handled communication of elements and of finite element or finite volume data associated with each element, any data not registered as a libMesh variable was not accessible at that level. MOOSE supports a form of data, stateful material properties, which allows for a high degree of flexibility. At each quadrature point, a material property might be tracking the evolution of a scalar, vector, matrix, or higher rank tensor value. Vector and higher order types might correspond to three-dimensional space, or to higher dimensional data, or might even be a

container of variable dimension. They might be expressed with a libMesh type or an Eigen type for the underlying value, which might be floating-point or integer data. The property might wrap underlying floating-point data in a MetaPhysicL automatic differentiation data type to track a sparse set of the data's derivatives. Some of these data types are less convenient to express as a libMesh finite element solution field, and other types are impossible to express that way, so for data with states that must be retained, MOOSE maintains its own storage system. This storage system was extended to allow for projections and restrictions of property values when refining and coarsening the elements on which they were computed, but it lacked any mechanism for parallel redistribution of those distributed-memory values, even when MOOSE was not operating on a distributed-memory mesh. This year's work corrected that final deficiency.

First, additional invocations were added from libMesh repartitioning code paths to the `redistribute()` virtual function available to `RelationshipManager` objects (which inherit from the `GhostingFunctor` abstract base class in libMesh). Previously, functors' `redistribute()` methods were only invoked when libMesh was redistributing a distributed mesh. MOOSE stateful material properties, however, are distributed-memory data even when the underlying mesh is not, giving MOOSE a chance to `redistribute()` its own data regardless of the underlying mesh settings. This change was necessary to handle mesh changes which redistribute element ownership in cases where element geometric data and indexing itself is replicated.

Second, a new `RelationshipManager` subclass was added to MOOSE, `RedistributeProperties`, which uses this callback as an opportunity to redistribute stored material property data as necessary. In cases where MOOSE expects to do mesh modification with stateful properties, it previously would have disabled load balancing and warned the user of the feature incompatibility. It now creates a `RedistributeProperties` object instead (for the reference mesh and also for any displaced mesh) and registers the result with libMesh. When libMesh triggers the `redistribute()` method, MOOSE can then iterate over all saved material properties and examine each associated element on the mesh. This call occurs during the narrow window in which element repartitioning is complete. Here, MOOSE can know exactly what data will need to be redistributed where, but element redistribution has not yet been performed. The old owners of each element still have them available for examination, even on a distributed mesh where the old owners are about to delete local copies of some elements.

How is the property data redistributed, when the property data types are so flexible? Unfortunately, C++ does not yet have any reflection or serialization as a general language feature, and it supports data members, like pointers, that cannot be directly copied from machine to machine without some form of serialization. Fortunately, MOOSE does already have serialization as a required capability of many objects including material property data, a capability used to support simulation restarts and thus heavily tested. Each data type must support overloads of a `dataStore` method to serialize that data type to an output stream and a `dataLoad` method to deserialize that data type from an input stream. In the context of simulation restart these streams are file streams. A store writes characters to a restart file, and a corresponding load restores data while reading that file. But this same interface can write to any C++ stream, including `stringstream` objects which retain the characters in memory. By reusing this capability in the context of stateful material property redistribution, we can obtain an in-memory serialization of every property. These serializations are then combined with location-identifying information (element IDs, numbers of quadrature points, property indices) in maps of vectors of data to be sent to target processors. The existing Non-Blocking Consensus (NBX) algorithm in the Templated Interface to MPI (TIMPI) library is then used to multiplex, communicate, and demultiplex these data. New stateful material property storage is instantiated for data received on target processors, and data values copied into it. No-longer-used stateful storage is freed on source processors.

Alongside this process, we undertook some work refactoring the stateful material property interfaces, to try to make any necessary future feature additions easier to accomplish while still retaining the system's overall flexibility. Data structures are better encapsulated by accessors. New assertions catch possible corruption. The MOOSE `HashMap` is thread-safe for more operations. Application Programming Interfaces (APIs) have been renamed to better reflect behavior. Minor bug fixes (for the combination of adaptive refinement and neighbor material properties, and for a postprocessor in the porous-flow module) were added.

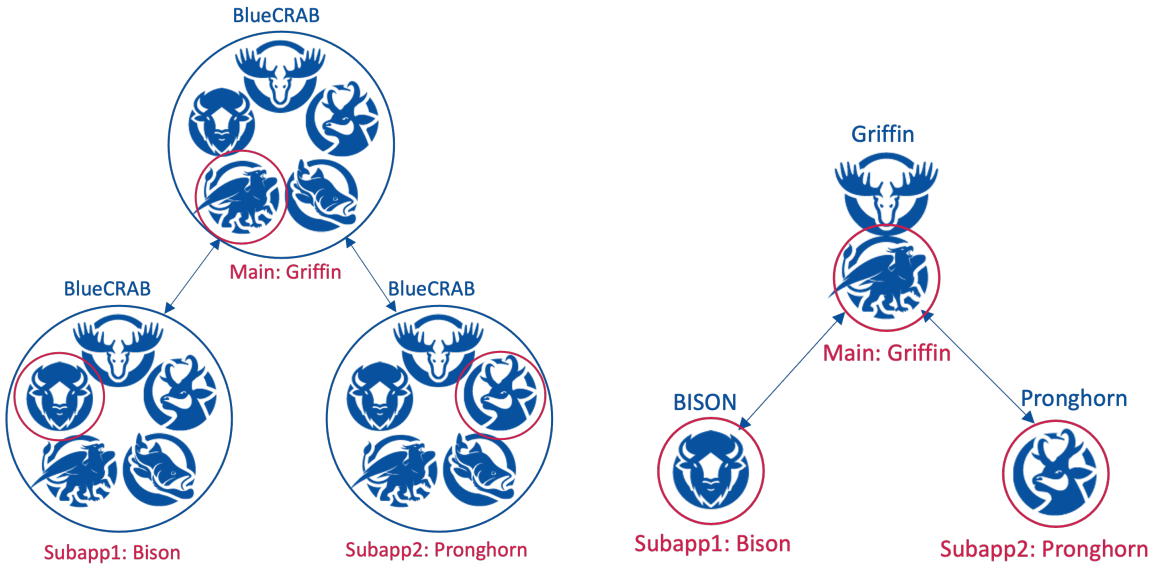
5. DYNAMIC APPLICATION LOADING

One of the features that developers cite for choosing the MOOSE framework over competing packages is the simplicity of the build system, specifically linking to other applications or simply

turning on new physics modules on demand. This can be performed with a single line change for each module in each application's "Makefile." Applications in MOOSE were designed to be just another extendable system like every other one in MOOSE. As a result, the MOOSE community has built a rich hierarchy of applications which exist primarily to seamlessly bring several independent efforts together. While this system has served the community well for many years, a new capability was needed to support the flexibility demanded by the communities of different analyst teams working on slightly different problems and therefore requiring slightly different combinations of applications. Enter the "dynamic application loading" capability (or "dynamic loading" for short).

Dynamic application loading is a feature in the MOOSE framework that allows users with multiple applications to compile them separately, but use them together in a single simulation. The technology supporting this capability has existed on operating systems for decades in the form of "plugins" or other application-specific extensible loading technologies. Developers can use standard operating system functions to dynamically load a compiled library or plugin by name. They can run functions from within the library to extend capabilities without the need to compile or link the library in during the compilation stage. This workflow is extremely popular with web browsers. Users can download plugins or extensions that can be loaded in the web browser to perform various tasks for the user such as changing the look and feel of each page or performing more complex functions like interacting with cloud-based services to report the current lowest price on a retail item displayed on the user's screen. With the MOOSE framework, this technology is used to allow developers and analysts to load entire applications on demand to assemble complex multiphysics simulations without requiring the creation of new source code or the need to complete additional compilation steps.

The dynamic loading feature first appeared in MOOSE more than 6 years ago. It was originally an experimental capability that proved to be less than robust for anything of sufficient complexity. As a result this capability was mostly abandoned, but it garnered a lot of interest from analysts and developers in the NEAMS program. For years NEAMS developers created makeshift patches in their Makefile to couple applications for various multiphysics needs. This behavior sparked renewed interest in this capability among the development team, and led to its re-imagining.



(a) `Multiapp` with monolithic application (b) `Multiapp` with dynamic loading capability

Figure 13: A typical `Multiapp` simulation setup using a monolithic configuration (13a) and a dynamic loading configuration (13b)

5.1 Dynamic Loading Implementation

As part of this milestone, a new robust dynamic loading capability was added to the MOOSE framework allowing for all MOOSE-based and even "MOOSE-wrapped" applications to be dynamically loaded using existing input file syntax. One simply needs to supply a path to the location for which MOOSE can find compiled libraries using one of two syntax options:

1. Utilize the `library_path` parameter in the `Multiapps` block in the input file. e.g.

```
[MultiApps]
  [./sub_app]
    type = TransientMultiApp
    input_files = 'sub.i'

    app_type = BisonApp
    library_path = '../../../bison/lib'
  [../]
[]
```

Listing 1: Multiapp block from a MOOSE input file.

2. Utilize the `MOOSE_LIBRARY_PATH` environment variable to store searchable library paths. This

variable can be exported in the user's shell and contain a colon separated list of searchable paths that MOOSE will inspect when looking for the requested app_type parameter in the [Multiapps] in the input file. e.g.

```
export MOOSE_LIBRARY_PATH=$HOME/projects/bison/lib:$HOME/projects/griffin/lib
```

The capability relies on a few standard conventions coded into every MOOSE-based application. First, each application library must have a standard naming convention. This allows the framework to find the library name based on "application name". For example, the BisonApp application will be found in a library named libbison.so or libbison.dylib for Linux and Mac operating systems, respectively. If the application contains camel case syntax like BlackBearApp, the associated library name must be libblack_bear.so or libblack_bear.dylib. The framework contains utilities to perform these conversions so in most cases users need not be aware of them.

Each library must also contain a standard library entry point naming convention. Using the pattern above, the convention is simply the application name followed by a pair of suffixes (e.g. The BisonApp application would contain BisonApp__registerApps and BisonApp__registerAll entry methods). Each application created with MOOSE's application creation template (i.e. Stork) comes with both of these methods pre-populated and ready to use for this capability.

When MOOSE sets up a simulation and encounters a Multiapp block such as the one shown in listing 1, it will first check it's known application name registry to see if knows about the application referenced in the app_type parameter. If it does not locate an entry, MOOSE attempts to find a dynamic library following the naming conventions mentioned above in any path specified through both the library_path parameter (if supplied) and the MOOSE_LIBRARY_PATH environment variable. If successful, the framework creates an instance of that application with no discernible difference apparent between this approach and one where the application is linked in during compile time. If the application cannot be found, an error message and diagnostic information about searched paths is written to the console to aid in troubleshooting.

5.2 Dynamic Object Registration

In addition to the dynamic loading capability covered in this section, MOOSE also contains a related capability called "Dynamic Object Registration." This capability, as the name implies, is a run-time feature where MOOSE-based libraries are loaded at run-time, and all of the objects within are registered with the currently running application. This capability allows developers to extend the capabilities of applications again without the need to obtain the source of those applications. New objects can be registered and used side by side with the objects from another application. This capability is distinct from the dynamic loading capability in that it allows users to utilize objects in a fully coupled manner instead of through the MOOSE `Multiapp` system.

5.3 Current Adoption

Several NEAMS program participants are actively using this capability on many of the program's software projects. This capability is known to work for the Microreactor analysis team at Argonne National Laboratory (ANL) as well as the Cardinal team. Cardinal [11] is significant in that it is a MOOSE-wrapped application linking in non-MOOSE-based applications such as NekRS [12] and OpenMC [13]. We fully expect this capability to be more widely used as new needs arise where a dedicated coupling application does not exist as well as in other non-NEAMS efforts.

5.4 Dynamic Loading versus Dynamic Linking

Dynamic linking is a commonly used term among application developers. So much so, that this jargon may even be familiar among some users as well. Briefly, dynamic linking refers to an Operating System (OS) and compiler capability where compilers can create separate library files that can be loaded automatically during run-time when the program requires the instructions or resources from those files. While a more in-depth discussion is beyond the scope of this report, there are two key points about dynamic linking worth pointing out here to differentiate it from the dynamic **loading** capability discussed in this report:

1. Requires at least some amount of access to the target application's source files since this capability works through "directory" information stored in the application and library files

at compile time.

2. Separate library files are loaded by the OS automatically and seamlessly without any knowledge or special application code needing to be added by the developer.

To reiterate, the dynamic loading capability implemented here does *not* require that the main application and extensions be compiled together opening up the use of this capability to binary users. Additionally, while the "automatic and seamless" loading results in the same experience for end users, explicit code to perform these function was added to MOOSE to account for the lack of linking information stored in our library files, precisely since libraries are *not* required to be compiled together. Finally, it should also be noted here that MOOSE does and has always utilized dynamic linking.

5.5 Future Enhancements

While the dynamic application loading capability is very powerful and gives developers and analysts a much easier path to building up more complex simulations, there are still potential pitfalls which we will continue to refine over the coming months and years as needed. Versioning is without a doubt one of the biggest drawbacks of dynamic loading in MOOSE and in general. Utilizing libraries compiled with drastically different versions of the framework may fail due to internal API changes. While the interfaces between applications are fairly stable, other framework features such as the growing `Transfer` system is under heavy enhancement. We are already planning a versioning system which will warn users if versions of applications have not specifically been tested together. We may also perform other versioning of internal APIs and check for compatibility in the future to ensure robust operation of this dynamic loading capability.

6. CONCLUSION

This report highlighted four foundational capabilities added in support of NEAMS applications: multiple nonlinear systems in the same input file, which continues to expand the coupling tools available to developers, implementation of generic field transfers and other transfer system enhancements giving more information exchange options to users for complex multiphysics simulation, support for stateful material property redistribution for adaptivity on

distributed meshes allowing for efficient simulation of systems which require stateful material support, and dynamic linking and loading of individually compiled applications, opening the door for adhoc combinations of applications with the need for source access to all applications. These additions are expected to be heavily leveraged by MOOSE-derived NEAMS applications for the ever-growing need for higher fidelity multiphysics modeling and simulation needs.

7. ACKNOWLEDGMENTS

The authors would like to thank Zachary Prince, who created the optimization inputs used to produce the results shown in Section 2.

REFERENCES

[1] A. J. Novak, P. Shriwise, P. K. Romano, R. Rahaman, E. Merzari, and D. Gaston, "Coupled monte carlo transport and conjugate heat transfer for wire-wrapped bundles within the moose framework," *Nuclear Science and Engineering*, vol. 0, no. 0, pp. 1–24, 2023.

[2] A. D. Lindsay, D. R. Gaston, C. J. Permann, J. M. Miller, D. Andrš, A. E. Slaughter, F. Kong, J. Hansel, R. W. Carlsen, C. Icenhour, L. Harbour, G. L. Giudicelli, R. H. Stogner, P. German, J. Badger, S. Biswas, L. Chapuis, C. Green, J. Hales, T. Hu, W. Jiang, Y. S. Jung, C. Matthews, Y. Miao, A. Novak, J. W. Peterson, Z. M. Prince, A. Rovinelli, S. Schunert, D. Schwen, B. W. Spencer, S. Veeraraghavan, A. Recuero, D. Yushu, Y. Wang, A. Wilkins, and C. Wong, "2.0 - MOOSE: Enabling massively parallel multiphysics simulation," *SoftwareX*, vol. 20, p. 101202, 2022.

[3] R. L. Williamson, J. D. Hales, S. R. Novascone, G. Pastore, K. A. Gamble, B. W. Spencer, W. Jiang, S. A. Pitts, A. Casagranda, D. Schwen, A. X. Zabriskie, A. Toptan, R. Gardner, C. Matthews, W. Liu, and H. Chen, "Bison: A flexible code for advanced simulation of the performance of multiple nuclear fuel forms," *Nuclear Technology*, vol. 207, no. 7, pp. 954–980, 2021.

[4] M. DeHart, F. N. Gleicher, V. Laboure, J. Ortensi, Z. Prince, S. Schunert, and Y. Wang, "Griffin user manual," Tech. Rep. INL/EXT-19-54247, Idaho National Laboratory, 2020.

[5] MOOSE Team, "MOOSE GitHub repository." `https://github.com/idaholab/moose`, 2014.

[6] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations," *Engineering with Computers*, vol. 22, no. 3-4, pp. 237–254, 2006.

[7] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini,

H. Zhang, H. Zhang, and J. Zhang, "PETSc/TAO users manual," Tech. Rep. ANL-21/39 - Revision 3.17, Argonne National Laboratory, 2022.

[8] S. Patankar, *Numerical heat transfer and fluid flow*. Taylor & Francis, 2018.

[9] S. Balay, S. Abhyankar, S. Benson, J. Brown, P. R. Brune, K. R. Buschelman, E. Constantinescu, A. Dener, J. Faibussowitsch, W. D. Gropp, *et al.*, "Petsc/tao users manual," tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2022.

[10] G. L. Giudicelli, C. Permann, D. Gaston, B. Feng, and A. Abou-Jaoude, "The virtual test bed (VTB) repository:a library of multiphysics reference reactor models using neams tools," in *Proceedings of PHYSOR*, 2022.

[11] A. Novak, D. Andrs, P. Shriwise, J. Fang, H. Yuan, D. Shaver, E. Merzari, P. Romano, and R. Martineau, "Coupled Monte Carlo and Thermal-Fluid Modeling of High Temperature Gas Reactors Using Cardinal," *Annals of Nuclear Energy*, vol. 177, p. 109310, 2022.

[12] P. Fischer, S. Kerkemeier, M. Min, Y. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, and T. Warburton, "NekRS, a GPU-Accelerated Spectral Element Navier-Stokes Solver," Apr. 2021. arXiv:2104.05829.

[13] P. Romano, N. Horelik, B. Herman, A. Nelson, B. Forget, and K. Smith, "OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development," *Annals of Nuclear Energy*, vol. 82, pp. 90–97, 2015.