INL/RPT-22-67144

User-oriented Improvements in the MOOSE Framework in Support of Multiphysics Simulation

May 2022

Nuclear Energy Advanced Modeling and Simulation M2 Milestone May 2022

Alexander Lindsay[1], Logan Harbour[1], Guillaume Giudicelli[1], Casey Icenhour[1], Fande Kong[1], Roy Stogner[1], Brandon Langley[2], Marco Delchini[2], Robert Lefebvre[2], Derek Gaston[1], and Cody Permann[1]

[1]*COMPUTATIONAL FRAMEWORKS*
[2]*OAK RIDGE WORKBENCH TEAM*

INL/RPT-22-67144

User-oriented Improvements in the MOOSE Framework in Support of Multiphysics Simulation

Nuclear Energy Advanced Modeling and Simulation M2 Milestone May 2022

Alexander Lindsay[1], Logan Harbour[1], Guillaume Giudicelli[1], Casey Icenhour[1], Fande Kong[1], Roy Stogner[1], Brandon Langley[2], Marco Delchini[2], Robert Lefebvre[2], Derek Gaston[1], and Cody Permann[1]

[1]COMPUTATIONAL FRAMEWORKS
[2]OAK RIDGE WORKBENCH TEAM

May 2022

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

Page intentionally left blank

CONTENTS

FIGURES

TABLES

ACRONYMS

HPC	High Performance Computing
INL	Idaho National Laboratory
MOOSE	Multiphysics Object-Oriented Simulation Environment
MPI	Message Passing Interface
NCRC	Nuclear Computational Resource Center
NEAMS	Nuclear Energy Advanced Modeling and Simulation
NEUP	Nuclear Energy University Program
TA	Technical Area
WASP	Workbench Analysis Sequence Processor

Page intentionally left blank

1. Executive 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 this ubiquity, improvements to the framework in support of modeling and simulation goals are critical to the program. These improvements can take many forms, including optimization, improved user experience, streamlined application programming interfaces (APIs), parallelism, and new capabilities. The work transcribed in this report was conducted in direct support of the simulation tools and has already been deployed or will be deployed in the coming months.

The capabilities were implemented in the same order as they are covered in this report: chainable execution objects or executors, support for transfers between same-level applications in a coupling scheme, support for boundary/subdomain restricted transfers, support for transfers between applications with different coordinate or unit systems, support for MOOSE applications in the NEAMS Workbench, deployment of the MOOSE application of the Idaho National Laboratory (INL) high-performance computing (HPC) OnDemand platform, the addition of a triangular meshing library in libMesh, and increased support of face variables.

2. Introduction

The 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 are responsible for delivering simulation tools to vendors, laboratories, and licensing authorities, while the Multiphysics Applications TA creates foundational capabilities for the program and exercises the tools to test them and ensure their usability for the intended purpose.

Reactors are inherently multiphysical: heat conduction, neutronics, solid mechanics, fluid flow, chemistry, and material evolution all combine to create a complex system that needs to be simulated. To tackle that problem, the NEAMS program utilizes the MOOSE platform [1] 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 [2] for nuclear fuel performance and Griffin [3] 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 the NEAMS applications, making for a large return on investment.

2.1 MOOSE

The MOOSE platform enables rapid production of massively parallel, multiscale, multiphysics simulation tools based on finite-element, finite-volume, and discrete ordinate discretizations. The platform was developed as open-source software on GitHub [4] and utilizes the LGPL 2.1 license, which allows for a large amount of flexibility. It is an active project, with dozens of code modifications 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/finite-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 the 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 this automation and reuse accelerates application development within the NEAMS program.

2.2 libMesh

Underpinning the MOOSE framework is the libMesh finite-element framework [5]. libMesh is an open-source software originally developed at the University of Texas at Austin (UT 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) [6]. MOOSE and libMesh were developed in lockstep with each other, with every change to either library 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.

2.2.1 TIMPI

The Templated Interface to MPI [7] is a C++-focused library based around the standard Message Passing Interface for distributed-memory HPC programming. Now an independent library, TIMPI was originally designed as part of libMesh, and its goals were to make the message-passing code there much simpler to read and write, less error-prone, and backwards compatible (when executed in serial) with non-MPI build environments. TIMPI has interfaces to MPI-standard features, but overloaded to communicate a variety of C++-standard templated containers. TIMPI generates MPI type descriptions and reduction operators for standard C++ classes, using an overloadable design that can also thus be extended to incorporate user classes (and nested combinations of any of the above). More recently, TIMPI was extended to add novel algorithms for large-scale non-blocking communication without an a priori sparsity pattern; the libMesh and MOOSE communication code has become much simpler and more scalable now that much of it has been migrated to the newest TIMPI methods.

2.2.2 MetaPhysicL

The Metaprogramming-for-Physics Library (MetaPhysicL) [8] provides a varied set of classes for compile-time and/or operator-overloaded evaluations that are also useful for other physics and engineering simulation codes. It was originally developed at the PECOS Center at UT Austin for use in Method-of-Manufactured-Solutions library calculations. MetaPhysicL was later made an optional libMesh dependency so that its dynamic sparse numerics could be used to enable projection-operator calculations for Geometric MultiGrid solves. After that, MetaPhysicL was made a required MOOSE dependency so that its numeric array and "dual number" types could be used in physics kernel residuals to produce automatic differentiation (AD) Jacobians.

MetaPhysicL has optional TIMPI overloads for its common classes, so projects built with both can seamlessly communicate MetaPhysicL data.

3. Flexible Executors

The Executioner system in MOOSE was among the first to be implemented, since even the most basic simulation requires an execution plan (i.e., Executioner) to function. It is also one of the most complex, as it must carry around a lot of state as well as a plan to execute a series of steps to carry out a simulation. On the most fundamental level, an Executioner contains a finite number of stages of a simulation, many of which farm out to other pluggable systems in MOOSE. Examples of these abstract stages include "pre-execute," "decide time step size," "take step," "check for termination condition,"and "post execute." For many years, MOOSE existed with just two main Executioners: `Steady` (a single steady-state solve) and `Transient` (a time-dependent series of solution steps), handling nearly all the needs of the vast collection of applications built on the framework. Later, the `EigenExecutioner` was added to solve Eigenvalue systems that appear in reactor transport applications. However, as the applications grew in complexity, so did the demands on the Executioner system.

The MultiApp system was one driver of the increasing complexity. The need to solve complex hierarchical arrangements of fully coupled applications made additional flexibility in the Executioner systems necessary. A modest redesign of this system occurred in 2018, with the existing core logic for moving transfer data and executing MultiApp solves being abstracted into yet another system (i.e., SolveObjects). The first and only instance of this system was simply called `PicardSolve`, in reference to Picard's fixed-point iteration method. While this necessary redesign accomplished significant cleanup in the code base, the design was far from being pluggable or extendable at that point in time. Later, additional `SolveObjects` were implemented to further support the increasingly complex simulation needs of the NEAMS program. The complexity continued to grow more unwieldy and difficult for analysts to use. To address these issues, a new, much more modular system was envisioned in which the Executioner system would be divided into several much more manageable blocks. The idea was that these blocks could be composed to form complex execution strategies while still remaining small, simple, and reusable for a wide variety of simulation needs. This new ″Executor″ system is the culmination of many years of experience and foresight—as was necessary for continued flexibility and growth.

The initial implementation of the Executor system did not include a replacement of the current `Executioner` objects. That is, the user had the option to either utilize a single `Executioner` or provide one or more `Executor` objects, but not both. This limitation made it impossible to combine `Executioner` and `Executor` objects. The next logical step for the Executor system is to completely replace the Executioner system. This replacement is the primary task for the Executor system in fiscal year 2022.

3.1 Implementation

The current `Executioner` objects will be completely replaced by `Executor` objects. For backwards compatibility, the `Steady`, `Transient`, and `Eigenvalue` objects still work with the previous `[Executioner]` syntax. The basic tasks in the previously discussed objects are separated into multiple `Executor` objects, such as `TransientExecutor` for the loop in time, `FEProblemSolveExecutor` for the single step solve, and `AdaptivityExecutor` for the adaptivity loop. This compartmentalization enables easier code maintenance as well as the use of fine control of the solve via user-defined `[Executor]` syntax.

The developer's construction of `Executor` objects from within an `Executor` was enabled in a fashion similar to how replacement of the Executioner system with the Executor system is intended to be as seamless as possible.

4. MultiApp Sibling Transfers

The MultiApp system in MOOSE is one of the key systems that set the MOOSE framework apart from other simulation packages. MultiApps allow MOOSE to solve complex multiphysics simulations with unprecedented flexibility, achieving high-fidelity results across a wide range of space and time scales. Independently built applications can be coupled using several different solver strategies, ranging from loose one-way coupling to fully coupled monolithic solves and everything in between. Moreover, this can be accomplished without writing any new code at all, meaning that application developers need not be involved in the framework development phase.

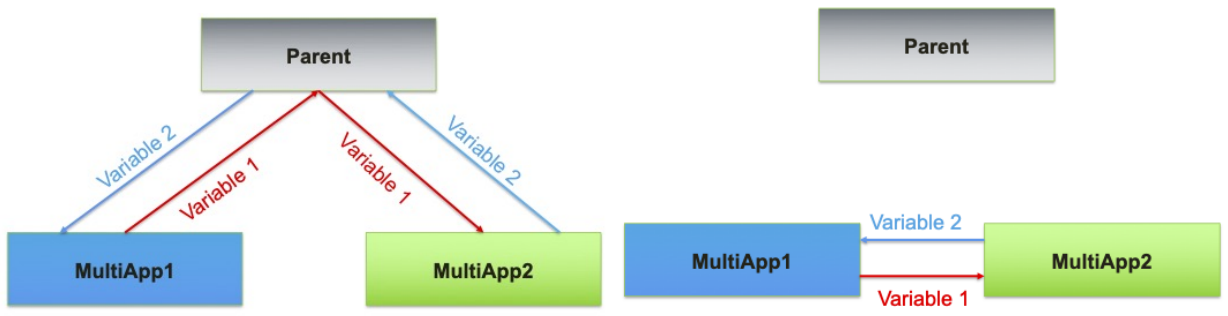


Figure 1: Conventional transfers and sibling transfers.

For several years, the MOOSE framework has been capable of coupling several applications together in hierarchical fashion to perform full-reactor analysis. Many of these models have routinely involved many applications working in concert to achieve realistic complex multiphysics simulations. One drawback to the prior approach was that all data movement was restricted to the structure of the hierarchical layout. That is, data transfers were only possible between ”parent” and ”child” applications. In simulations in which the natural layout consisted of a parent and two or more children applications, direct data exchange between the two children (or ”siblings”) was impossible. This meant that any data needing exchanged between siblings would first have to be sent to the parent before being directed back down to the other siblings in a separate transfer. This approach had two drawbacks:

- Two separate data transfers had to be set up to move data logically to another application.
- Discretization issues could cause loss of information in cases in which the intermediate application lacked sufficient mesh or other logical data structures to hold transitive data. For example, if the parent mesh is too coarse, the accuracy of the transferred variable will be lost. Sometimes, the parent application does not have a mesh or has a very different mesh, creating a barrier to data exchanges between siblings.

To address both issues, a new capability was added to the framework: ”Sibling Transfers.” As the name implies, this new capability enables the ability to move data between the sibling applications corresponding to a given parent, as shown in the right of Fig. 1.

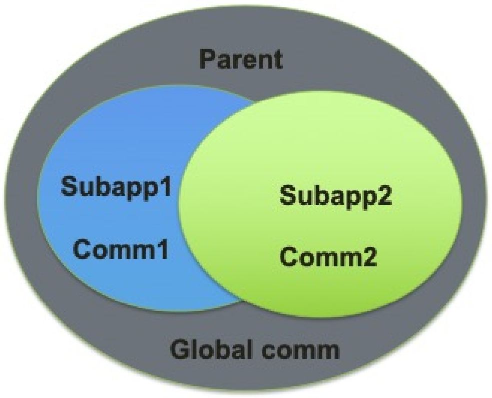


Figure 2: Communicator relationship between parent and child applications. A sub-application can have a communicator that differs from that of its parent, and different sub-applications can also have different communicators.

4.1 Sibling Transfers

As was mentioned, the MultiApp system is a widely employed multiphysics coupling simulation approach in which various physics components can be coupled either tightly or loosely. The efficiency and effectiveness of this coupling approach highly depends on the data transfers, in which various physics components interact with each other. While "conventional" data transfers in general are already challenging, sibling transfers introduce extra difficulties.

MOOSE is a massively parallel multiphyics simulation platform, and to take advantage of modern supercomputers, the MultiApp system has a configurable and flexible way to construct and execute sub-applications (instances of child applications). This adds complexity, since the communicators of different sub-applications may greatly vary, depending on the user input parameters, as shown in Fig. 2. In general, an individual sub-application is totally independent from the rest, and there is a limited number of ways in which these sub-applications can communicate with each other.

One additional challenge is that simulations may be concerned about data transfers from the subdomain/boundary of an application's computational domain to the subdomain/boundary of another application domain. In other words, we must transfer data from one arbitrary region of a sub-application to another region of another application. In addition, the data transfer task will be performed in parallel, requiring a parallel spatial search.

4.2 Implementation

To handle the communicator heterogeneity of sub-applications, we used the global parent communicator to manage data transfers, and not individual sub-application communicators. The parent communicator is a super-set of sub-application communicators and can safely cover all the sub-application subdomains. The implementation was based on TIMPI's parallel "pull" routine that has a flexible interface for sparse data transfers of any data type. To transfer the solution data from a selected region of the source application to the target region of the target application, we introduced subdomain and boundary restricted transfer strategies. The basic idea behind this approach was straightforward: during the transfer process, we constructed algorithms based on the selected region as if other the regions did not exist. This enabled an efficient and accurate solution transfer. The basic algorithm framework is shown in Alg. 1.

Algorithm 1 Restricted nearest node transfer

```
1: procedure TRANSFER(source mesh, target mesh)
2:     Construct bounding boxes from source meshes while considering subdomain and
       boundary restrictions
3:     Extract outgoing points based on the target meshes while considering subdomain and
       boundary restrictions
4:     Gather incoming points from remote owners
5:     Find nearest nodes by using KDTree for each incoming point
6:     Send nearest node values back to querying owners
7:     Set node values to solution vectors
8: end procedure
```

4.2.1 Input syntax for new features

The transfer shown in Alg. 1 is general and can be used to transfer data between the parent and the child applications, or from one child application to another. Data transfer between child applications at the same level is referred to as "sibling transfer." The following is an example of input when users wish to use sibling transfers:

```
[MultiApps/ma1]
  type = TransientMultiApp
  input_files = sub_between_diffusion1.i
  max_procs_per_app = 1
[]
```

```
[MultiApps/ma2]
  type = TransientMultiApp
  input_files = sub_between_diffusion2.i
  max_procs_per_app = 2
[]

[Transfers/from_ma1_to_ma2]
  type = MultiAppGeneralFieldNearestNodeTransfer
  from_multi_app = ma1
  to_multi_app = ma2
  source_variable = u
  variable = transferred_u
[]

[Transfers/from_ma2_to_ma1]
  type = MultiAppGeneralFieldNearestNodeTransfer
  from_multi_app = ma2
  to_multi_app = ma1
  source_variable = u
  variable = transferred_u
[]
```

The key parameters “from_multi_app” and “to_multi_app” are introduced to specify the transfer direction, “from_multi_app” indicates the source MultiApp, and “to_multi_app” is utilized to represent the target MultiApp. If we do not specify one of these parameters, the transfer assumes that the parent application will be used.

For boundary restricted transfers, we introduced the parameters “from_boundary” and “to_boundary” to specify source and target boundaries, respectively. A boundary restricted input example is shown below, and the result is shown in Fig. 3.

```
[from_sub]
    source_variable = sub_u
    direction = from_multiapp
    variable = from_sub
    type = MultiAppGeneralFieldNearestNodeTransfer
```

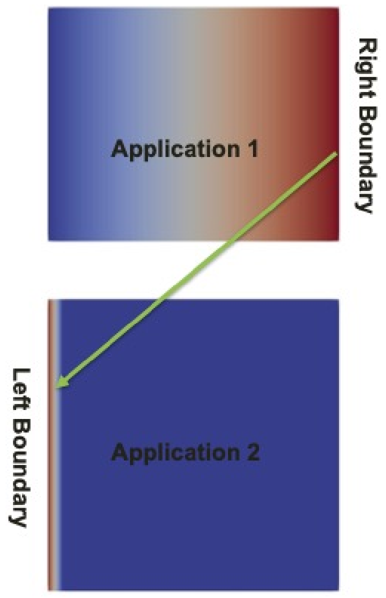


Figure 3: Boundary restricted transfer. A solution vector is transferred from application 1 to application 2, with specified boundaries.

```
    multi_app = sub
    from_boundary = 'right'
    to_boundary = 'left'
[]
```

Similarly, we introduced subdomain restricted transfers with the parameters "from_blocks" and "to_blocks," which are used to specify source and target blocks, respectively. As an example:

```
[from_sub]
    source_variable = sub_u
    direction = from_multiapp
    variable = from_sub
    type = MultiAppGeneralFieldNearestNodeTransfer
    multi_app = sub
    from_blocks = '1'
    to_blocks = '1'
[]
```

4.2.2 API changes

Two major API changes were implemented in addition to adding the new capability and the input file syntax changes. These API changes influence the applications' downstream use of these

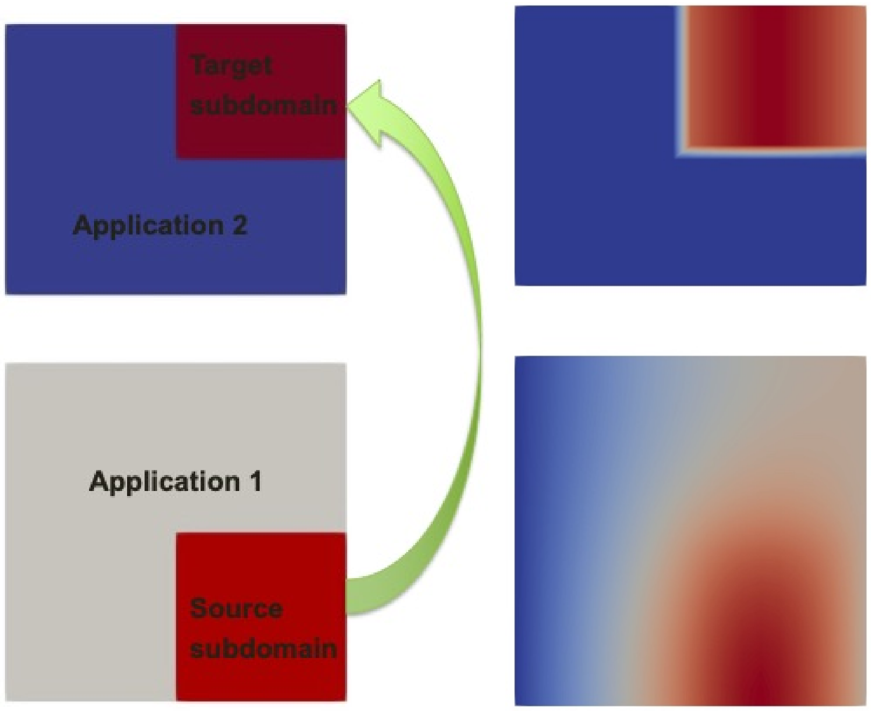


Figure 4: Subdomain restricted transfer. A solution vector is transferred from application 1 to application 2, with specified subdomains. The result is shown on the right.

systems. The patches were generated by the MOOSE team and proposed in pull/merge requests in all the affected downstream applications.

The first change was moving the information about the direction from the Transfer base class to the MultiAppTransfer class. This change was required because the direction is no longer an explicit parameter but instead determined from the from_multiapp and to_multiapp parameters. The direction parameter was deprecated and they can no longer be set in the constructor of the transfers for which only a single direction is implemented. The equivalent technique is to remove the MultiApp parameter for the unsupported direction.

The second change was to remove the _multiapp attribute. This attribute was unclear whenever there were two MultiApps for a transfer: one sending data and one receiving them. This attribute was replaced by two attributes, _from_multiapp and _to_multiapp, which were made private. The new attributes can now be accessed using the getFromMultiApp and getToMultiApp accessors. This allows for future changes without requiring patching of the underlying the applications.

The patched applications to adapt to the API changes are listed below. Both the source code and the input file were adapted. All but one patch were merged over the course of the same business day as when the changes in MOOSE were merged.

- MASTODON
- Isopod
- Blackbear
- Sockeye
- Griffin
- Pronghorn
- Pronghorn subchannel
- Mockingbird
- Bison
- SAM
- Cardinal
- Ferret

4.2.3 List of transfers supported

The following transfers can now perform sibling transfers natively in MOOSE.

- MultiAppCopyTransfer
- MultiAppPostprocessorTransfer
- MultiAppReporterTransfer
- MultiAppCloneReporterTransfer
- MultiAppVectorPostprocessorTransfer
- MultiAppScalarToAuxScalarTransfer

- MultiAppPostprocessorToAuxScalarTransfer

The sibling transfer capability was also implemented in the MultiAppGeneralFieldTransfer that features MeshFunction and nearest methods, and which is currently in a pull request to the framework. This later transfer encompasses the rest of the transfers of interest (projection, nearest node, interpolation, etc.).

4.3 Applications

4.3.1 Fluid-solid coupled problems

We added a 3-D sibling transfer example (shown below) with boundary restriction, demonstrating data transfer between a solid application and a fluid application. Sibling transfer between the fluid and the solid is required, since the parent can be a coarse-mesh application such as a neutron simulation. If the transfer was conducted through the parent application, accuracy would be lost. The region of interest is the interface between the fluid and solid domains. Here, we used the temperature as an example, though other variables are also possible. The transferred results are shown in Fig. 5. We observed that the temperature data were transferred correctly between the fluid and solid interfaces in both directions.

```
[MultiApps/solid]
    type = TransientMultiApp
    input_files = solid.i
[]
[MultiApps/fluid]
    type = TransientMultiApp
    input_files = fluid.i
[]
[Transfers/from_solid_to_fluid]
    type = MultiAppGeneralFieldNearestNodeTransfer
    from_multi_app = solid
    to_multi_app = fluid
    source_variable = temp
    variable = solid_T
    from_boundary = 'surface_surface100'
    to_boundary = 'surface1_surface2'
```

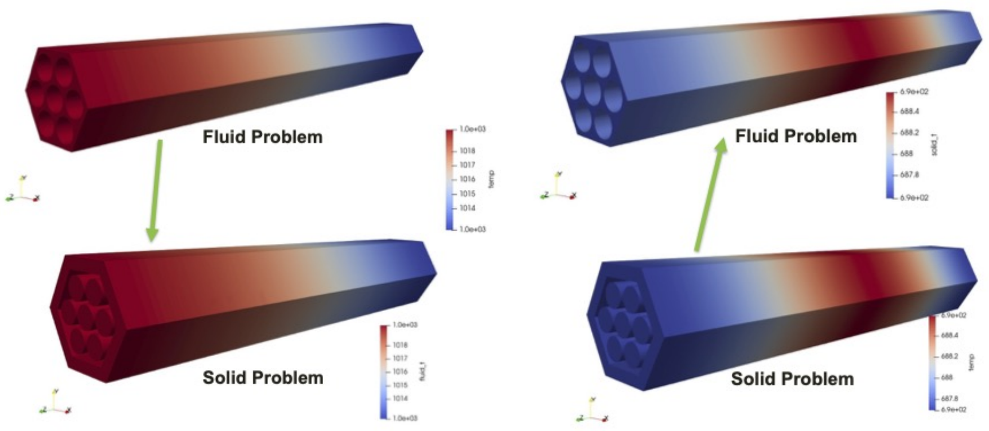


Figure 5: Boundary restricted sibling transfer. A solution vector is transferred from/to the solid domain to/from the fluid domain when solid and fluid problems are implemented as MultiApps.

```
[]
[Transfers/from_fluid_to_solid]
      type = MultiAppGeneralFieldNearestNodeTransfer
      from_multi_app = fluid
      to_multi_app = solid
      source_variable = temp
      variable = fluid_T
      from_boundary = 'surface1␣surface2'
      # The transfer with and without 'to_boundary'
␣␣␣␣␣#to_boundary␣=␣'surface surface100'
[]
```

4.3.2 Molten-salt reactor precursor modeling

We demonstrated sibling transfers for the Molten Salt Fast Reactor (MSFR) model [9, 10] hosted on the National Reactor Innovation Center Virtual Test Bed [11–13]. This model consists of a 2-D RZ coupled multiphysics simulation of a closed-loop MSFR. Neutronics is coupled to coarse-mesh thermal fluid flow, also solving the advection of the precursors in the system. The two codes used were Griffin [3] for neutronics and Pronghorn [14] for the fluid and precursors flow. Current results are currently unphysical; the temperatures are too high in the center of the core, due to the insufficiency of the coarse-mesh turbulence model. This falls outside the scope of this demonstration and will be improved in future iterations of the model. Decay heat precursor

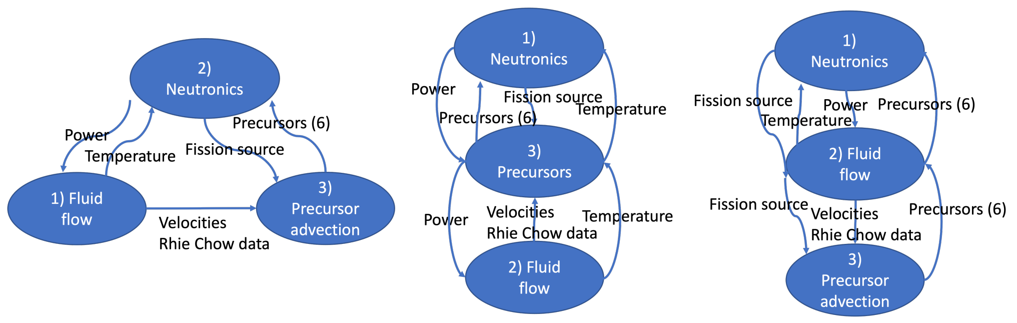


Figure 6: Possible coupling schemes for the MSFR multiphysics model of the core: pyramid with sibling transfer (left); linear with fluid flow, requiring smallest time steps (mid); and linear with a precursor advection-reaction, requiring smallest time steps (right).

advection is also not considered in this model.

Currently, this model is set up using MultiApps, with a neutron transport eigenvalue problem as the parent app and a fluid/precursor flow simulation as the MultiApp. While the fluid solution directly influences the precursor distribution, this distribution only influences the fluid solution through the neutronics solution and power distribution. Therefore, it is natural to separate those solves, and doing so strongly reduces the numerical difficulty of the coupled flow problem.

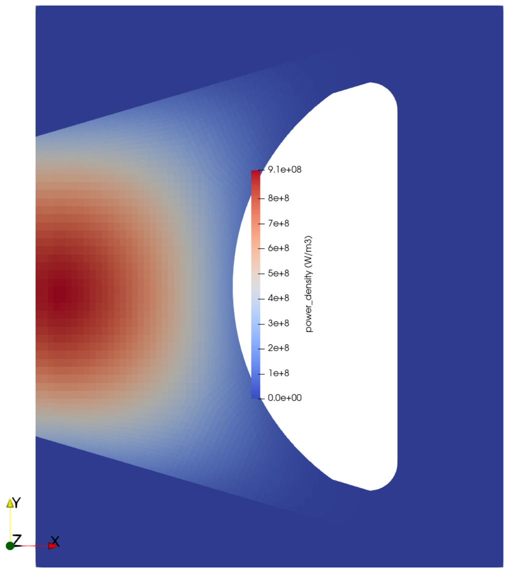
However, this was rather difficult to perform using the previous capability. If the precursors were split into their own separate applications, there would then be a total of three apps with cyclic needs for transfers. For example, the power distribution should be sent from neutronics to fluid flow, which should send velocities to the precursor solve, which should send the precursors' concentrations to the neutronics. There are also transfer needs in the other direction of the cycle (e.g., fission source from neutronics to the precursor solve).

Schematics for different MultiApp setups are shown in Figure 6. A linear setup of this coupling requires passing quantities through a MultiApp, thus necessitating storage in the intermediate application in between. A pyramid setup of this coupling requires storage of the exchanged variables between the MultiApps in the parent app. Either way, the memory cost is inflated prior to sibling transfers. The memory requirements in terms of the number of fields to store are shown in Table 1. The pyramid scheme's memory requirement needs are largely deflated by using sibling transfers.

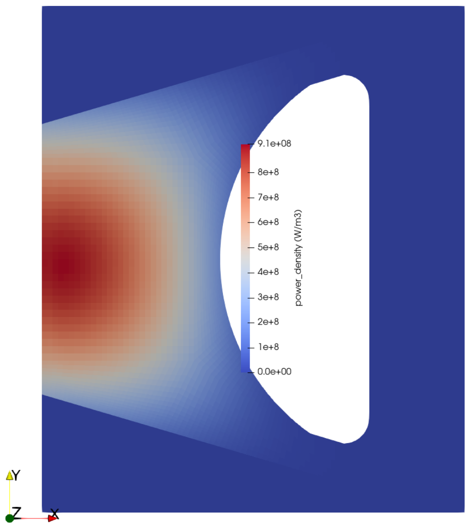
Table 1: MultiApp storage requirements of fields. Time stepping, not memory storage, is usually the main consideration when designing a MultiApp setup. Note that the future multi-system capability will entirely remove the need for this MultiApp setup, eliminating the duplicate storage entirely.

Application	Pyramid - no sibling transfer	Linear bottom = precursor	Linear bottom = fluid	Pyramid - siblings
Griffin	Fluxes (6)	Fluxes (6)	Fluxes (6)	Fluxes (6)
	Power	Power	Power	Power
	Fission source	Fission source	Fission source	Fission source
	Precursors (6)	Precursors (6)	Precursors (6)	Precursors (6)
	Temperature	Temperature	Temperature	Temperature
	Velocities			
	Pressure, RC Coeffs			
Pronghorn	Velocities	Velocities	Velocities	Velocities
Fluid and	Pressure, RC coeffs	Pressure, RC coeffs	Pressure, RC coeffs	Pressure, RC coeffs
energy	Temperature	Temperature	Temperature	Temperature
	Power	Power	Power	Power
		Precursors (6)		
		Fission source		
Pronghorn	Velocities	Velocities	Velocities	Velocities
precursors	Pressure, RC coeffs	Pressure, RC coeffs	Pressure, RC coeffs	Pressure, RC coeffs
	Precursors (6)	Precursors (6)	Precursors (6)	Precursors (6)
	Fission source	Fission source	Fission source	Fission source
		Power		
		Temperature		
Duplicates	18	22	14	14
Total fields	40	43	34	32

Figure 9 compares the results for the MSFR multiphysics model with two different numerical schemes: MultiApps with sibling transfer and the previous model (fluid flow strongly coupled with precursor advection). Strong agreement is shown, indicating the transfers are performing as



(a) Fully coupled fluid flow



(b) Segregated, with sibling transfer

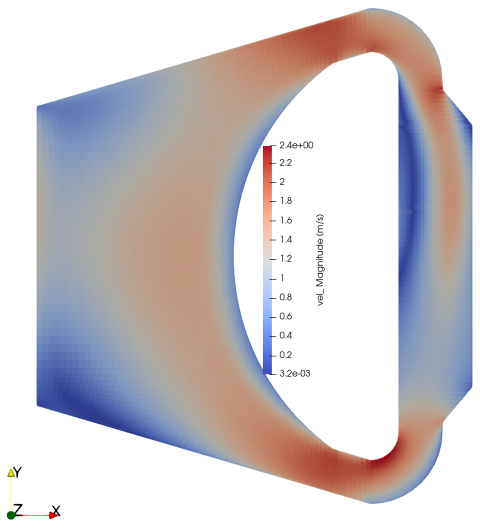
Figure 7: Power distribution.

expected. Some numerical artefacts can be seen in the precursor concentration near the bottom of the core in the segregated case. They are likely explained by the advection of the precursors with an unconverged or unsteady flow distribution. This is cause for further investigation of the model. In using a linear MultiApp structure instead of sibling transfers, the same artefacts are found, so the issue does not lie with the implementation of MultiApp sibling transfer. Transferring the flow fields at every time step in the relaxation transient instead of at the end will likely suffice to resolve this, but is akin to the subcycling capability described in Section 4.4.

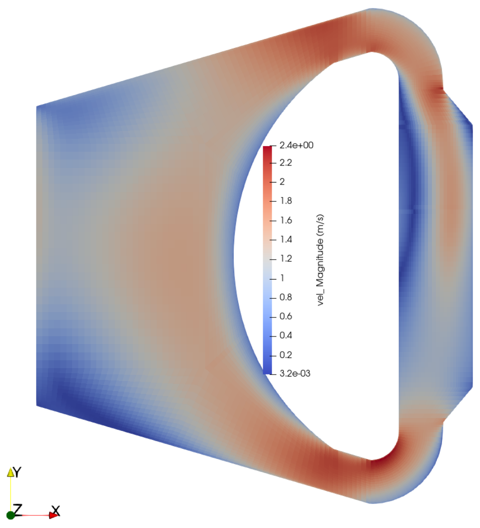
A notable difference between the two schemes is the simulation time: the pyramid MultiApp scheme with sibling transfer took ≈700 s to run on a 2020 fully speced MacBook©Pro. The same model with a different coupling scheme previously clocked over 1,200 s. This difference is due to segregating the precursor solve out of the fluid flow solve.

4.4 Future work

The added capability already significantly increases flexibility in MultiApp-based numerical schemes. It removes the need to add fields on the parent application to host variable data

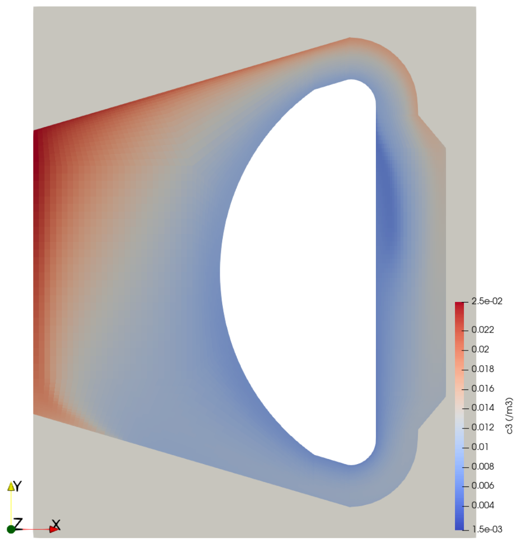


(a) Fully coupled fluid flow

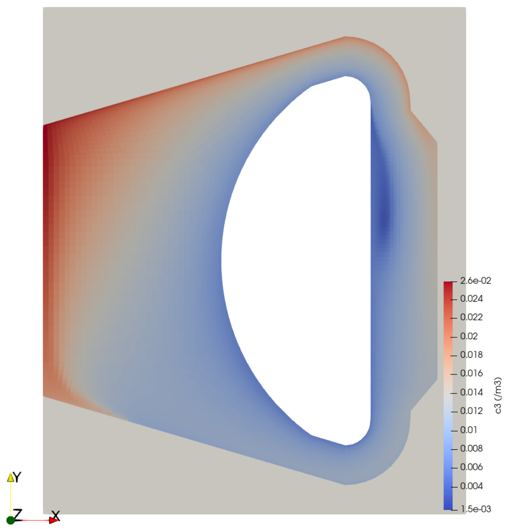


(b) Segregated with siblings transfer

Figure 8: Norm of the fluid velocity.



(a) Fully coupled fluid flow



(b) Segregated with siblings transfer

Figure 9: Group 3 delayed neutron precursors.

transferred from one MultiApp to another. However, further improvement to this new capability should be considered, most notably in two areas: dependency resolution and compatibility with subcycling.

Sibling transfers are currently executed either before or after the entirety of the MultiApps are run (on the same execute_on but with different execution parameters, staggering is possible). There is currently no option to transfer fields after one MultiApp is run—to possibly have an updated quantity, from a transfer, before the other MultiApp is run. This limitation was kept because there is currently no dependency resolution between transfers and MultiApps. Dependency resolution enables ordering of the transfers and MultiApps. The MultiApps that send data to another MultiApp should be executed first, then the sibling transfer, then the receiving MultiApp. In the case of bi-directional sibling transfers, the ordering should be specified explicitly.

Subcycling is the execution of multiple smaller time steps in the MultiApp while the parent app only performs a single time step. In the previous transfer paradigm, transfers may be executed either at the beginning or end of the succession of smaller time steps. In the sibling transfer paradigm, it is desirable to support transfers between two subcycled applications. The first major difficulty is that the parent app must regain control at every step of the subcycles to initiate the transfers. This is a rather large modification of the current workflow. The second difficulty is that the MultiApps may be on different schedules for the their subcycles, introducing additional needs such as the interpolation of transfers in time between uncoordinated subcycles. Even if the two subcycles are coordinated, a failed time step in one of the MultiApps may de-synchronize them.

Enabling these two features will provide the upmost degree of flexibility and perform the most logical workflows for multiphysics coupling. In the current state of the code, the limitations introduced were designed so as to maintain the intuitively expected behavior.

5. MultiApp Transfer Coordinate System Enhancement

New capabilities implemented in the Transfer System were introduced in Section 4. In this section, we introduce a complementary capability aimed at further simplifying transfers among separate MOOSE-based applications and reducing human error. As a framework, MOOSE has never prescribed a fixed convention for units or coordinate systems. While nearly every application built upon the platform uses SI units, even applications that model under similar time or space scales may choose different unit prefixes (e.g., centi- versus milli-) to follow the conventions of a given field. Likewise, a similar issue crops up when considering orientation within a coordinate system (i.e., which way is ”up”?). On the surface, these seem to be relatively insignificant issues that would not require a large effort to resolve, but many challenging issues arise when considering all the various ways in which one might choose to combine multiple applications. To understand some of the challenges in implementing such a system, consider this example: a 3-D model in which one application uses a Cartesian coordinate system, in which +Z is ”up” and the main unit of length is given in meters, coupled to a second application that also uses a Cartesian system but with +Y pointing up and a length measurement given in centimeters. Each data movement between these applications must go through both a scaling and rotation operation, but at least the mapping between these applications is one to one. Now consider a new case in which the first application has the same convention as before (Cartesian / +Z is up / length in meters), but the second application sits in a cylindrical or ”RZ” coordinate system with +Y as up. While one of these applications occupies true 3-D space, the other is only effectively 3-D. While the mapping to go from the 3-D to the 2-D space is just a series of linear mappings, reversing that mapping is non-trivial because it is not one to one. As an example, for each coordinate in the 2-D domain, there is effectively an entire ”edge” in 3-D space (e.g., a cylinder circumference of given radius) that maps to that location.

5.1 Implementation

To address the mapping or series of linear transformations that must be performed to map between any two applications, a new system was envisioned and implemented to create a universal coordinate system for all applications. The new system centers around the `MooseCoordTransform` class. Each `MooseApp` instance holds a coordinate transformation object in its `FEProblemBase` object. Users may specify transformation information about their simulation setup on a per-application basis or in the input file `Problem` block. The `coord_type` parameter specifies the coordinate system type (e.g., XYZ, RZ, or RSPHERICAL). Euler angles are available to describe extrinsic rotations. The convention MOOSE uses for a alpha-beta-gamma Euler angle rotation is:

1. Rotation about the z-axis by `alpha_rotation` degrees
2. Rotation about the x-axis by `beta_rotation` degrees
3. Rotation about the z-axis (again) by `gamma_rotation` degrees.

`length_units_per_meter` allows the user to specify how many mesh length units are in a meter. The last option that contributes to transformation information held by the `MooseCoordTransform` class is the `positions` parameter. The value of `positions` exactly corresponds to the translation vector set in the `MooseCoordTransform` object of the sub-application. The `alpha_rotation`, `beta_rotation`, `gamma_rotation`, and `positions` parameters essentially describe forward transformations of the mesh domain described by the MOOSE `Mesh` block to a reference domain. The `length_units_per_meter` parameter effectively represents an inverse transform (e.g., the meter is chosen as the reference scale), so a $1/l$ scaling transform (where l corresponds to `length_units_per_meter`) would be applied to the mesh domain in order to map to the meter-based reference domain. The sequence of transformations applied in the `MooseCoordTransform` class is:

1. rotation
2. scaling
3. translation
4. coordinate collapsing.

The last item in the list, coordinate collapsing, is only relevant when information must be transferred between applications with different coordinate systems. For transferring information from XYZ to RZ, we must collapse XYZ coordinates into the RZ space, since tthe mapping of XYZ coordinates into RZ coordinates is unique—but not visa versa (e.g., a point in RZ has infinite corresponding locations in XYZ space, due to rotation about the axis of symmetry). The table below summarizes the coordinate collapses that occur when transferring information between two different coordinate systems.

Table 2: Coordinate collapsing

Coordinate System	Paired Coordinate System	Resulting Coordinate System for Data Transfer
XYZ	RZ	RZ
XYZ	RSPHERICAL	RSPHERICAL
RZ	RSPHERICAL	RSPHERICAL

Let us consider an example. The below listing shows coordinate transformation in the `Problem` block of a sub-application:

Listing 1: sub-application transformation block

```
[Problem]
  alpha_rotation = -90
  length_units_per_meter = 5
[]
```

Here, the user is stating that a -90 degree alpha rotation should be applied to the sub-application's domain in order to map to the reference domain (which the user has chosen to correspond to the main application domain). Additionally, the user wishes the coordinate transformation object to know that five sub-application domain units correspond to a meter (whereas the main application is already in units of meters). This information from the sub-application's `Problem` block, combined with the translation vector described by the `positions` parameter in the main application `MultiApp` block allows MOOSE to directly map information between the disparate

Listing 2: main application MultiApp block

```
[MultiApps]
  [sub]
    type = FullSolveMultiApp
    app_type = MooseTestApp
    positions = '1 0 0'
    input_files = 'sub-app.i'
    execute_on = 'timestep_begin'
  []
[]
```

application domains. Fig. 10 shows the variable field v, which in both domains is a nonlinear variable in the sub-application and an auxiliary source variable in the main application, indicating a successful transfer of data after applying the transformation order outlined above (rotation, scale, translation).

Another example leveraging the `MooseCoordTransform` class is a simulation in which one application is in the 3-D XYZ space and another is in the 2-D RZ space. In this example, we wish to rotate the axis of symmetry, which is the Y-axis in the 2-D RZ simulation, in order to align it with the z-axis when transferring data between the 2-D RZ and 3-D XYZ simulations. This simple rotation is achieved by specifying the `beta_rotation` value below:

Listing 3: 2-D RZ sub-application transformation parameters

```
[Problem]
  coord_type = RZ
  beta_rotation = 90
[]
```

We can see that the rotation transformation was successful by examining the same field in both applications (in this case, the field is solved by the nonlinear solver in the sub-application [variable `u`] and transferred to the main application into the auxiliary field `v`). This is shown in Fig. 11.

We mentioned how forward rotation transformations can be achieved by specifying Euler angles. Another parameter that can be used to perform rotations is `up_direction`. As described in the parameter documentation string, if `up_direction` is specified, then in the `MooseCoordTransform` object we will prescribe a rotation matrix that corresponds to a single 90 degree rotation, such that a point lying on the `up_direction` axis will now lie on the y-axis. We have chosen the y-axis to be the canonical reference frame up-direction because it is the literal up-direction when opening an Exodus file in ParaView. Additionally, it is consistent with boundary naming for cuboidal meshes generated using MOOSE meshing internals in which the upper y-boundary is denoted as `top`.

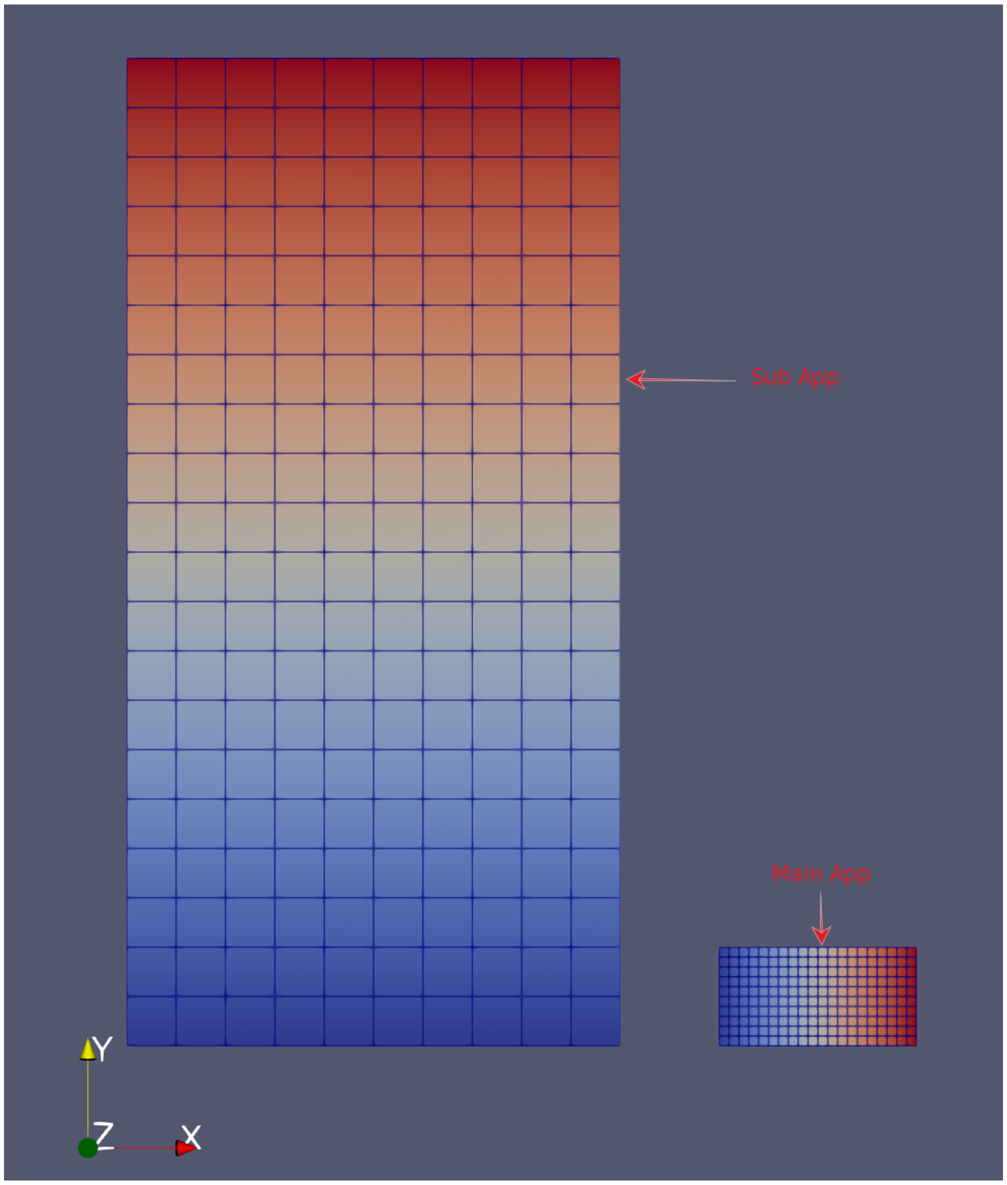


Figure 10: Example of rotation, scaling, and translation transformations between MultiApps.

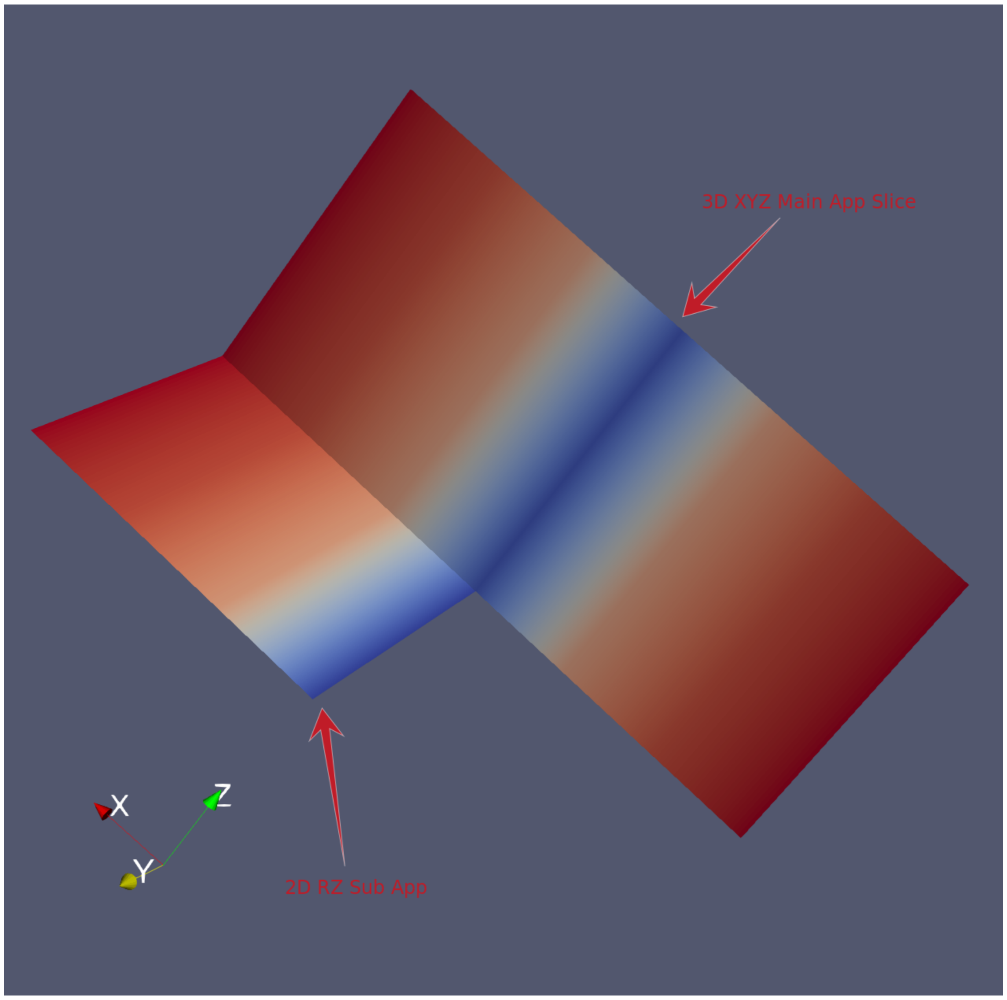


Figure 11: Example of information transformation between different coordinate system types.

6. MOOSE Language Improvements

Since 2016, improvements have been made to increase the access and usability of MOOSE application input via integration into the NEAMS Workbench [15]. More recent work has underscored the need for consolidating advanced language functionality into the MOOSE framework itself [16]. These improvements focus on implementing the Language Server Protocol [17] into the MOOSE framework.

There are two primary objectives for input processing to provide user-oriented improvements. The addition of the language server to the MOOSE framework will provide future releases with the ability to quickly communicate input options and diagnostics to the user via available integrated input development environments such as those available in the NEAMS Workbench. The second is the ability to allow the inclusion of parts of user input via a file-include capability.

6.1 Language Server Protocol

The language server protocol was integrated into the NEAMS Workbench via an NEUP collaboration [18]. This collaboration resulted in an open-source language server and client implementation in the C++ programming language. In combination with the language server, the NEAMS Workbench also includes a language processor for the MOOSE application syntax. As such, a benchmark study was performed [19] to determine the most suitable language processor on which to implement the MOOSE language server. The benchmark results indicated that the NEAMS Workbench language processor and language server, openly available via [20], were both the most performant and economical path forward.

As of this writing, the WASP is compiling in the MOOSE build environment, and all interface locations have been identified. Substitution of the MOOSE HIT parse logic with the WASP parse logic will likely be completed by the time this report is published. In addition, the locations for connecting MOOSE input processing (errors, warning, etc.) to the language server were identified. It is expected that further MOOSE input logic will need to be consolidated to fully enable language server capabilities with initial completion by the end of this fiscal year. Additional complexities of Actions and MultiApps are scheduled for fiscal year 2023 activities.

6.2 Multiple Input File Capability

Users have expressed a desire to consolidate problem inputs via the support of a file-include capability in the MOOSE application input. The ability to include files helps users reuse input file content and refactor larger input file setups into more understandable problem descriptions. With the adoption of the WASP as the MOOSE input processor, the file-include capability has been added and will be available by the end of FY22.

For example, for a set of application inputs that share Mesh and Materials, instead of repeating these definitions in each problem input, the definitions can be consolidated into files that can be included as needed.

```
[Mesh]
... mesh block definition ...
[]
```

Listing 4: Common Mesh definition *Mesh.i*

```
[Materials]
... material block definition ...
[]
```

Listing 5: Common Materials definition *Materials.i*

The application input can now include the de-duplicated components.

```
...
!include Mesh.i
...
!include Materials.i
...
```

Listing 6: Reuse of *Materials.i* and *Mesh.i* in an other MOOSE input file

The new file-include capability supports relative and absolute file paths and is not limited to the top-level of the problem input. It can be used at any location in the problem input and has an arbitrarily deep file inclusion hierarchy. The only restriction is that the syntax of components (blocks, arrays, parameters) are not split across file boundaries. At the time of this writing, the file-include capability is targeted for public availability at the end of the FY22.

7. INL HPC-OnDemand Workflow Improvements

The HPC-OnDemand capability at INL provides easy access to MOOSE applications and the HPC resources on which to perform modeling and simulation analysis. While the INL High Performance Computing (HPC) provides a central location and compute clusters, it still requires the user to have experience with Linux command line interfaces and PBS scheduler instructions. However, this system configuration, along with the provided Linux Desktop environment, has facilitated a new feature in the NEAMS Workbench to streamline interaction with the available multiphysics applications.

7.1 NEAMS Workbench updates

The configuration control of MOOSE applications on the INL HPC for Level 1 remote access has enabled a new logic module to be added to the NEAMS Workbench that enables the available applications to be activated and made ready for use in as little as three mouse clicks. Previously, users required the NEAMS Workbench to be installed on both their laptop/desktop (client) and the compute resource (server), and they had to configure the connection between the two to facilitate remote job launch [21]. While this affords great flexibility, it often entails great technical issues with regard to navigating network configurations. This is compounded by the fact that the application configuration and subsequent network knowledge is required from each user. This is where the HPC-OnDemand capability consolidates these issues and enables the NEAMS Workbench to be updated to automatically address them.

The first updates to the NEAMS Workbench enabled operation as a central installation. This consolidated the need for each user to perform an installation. The second update enabled the NEAMS Workbench to identify the available MOOSE applications and allow the user to activate the applications of interest via a simple menu item click of *File* $\gg$ *Localhost...* $\gg$ *Application* $\gg$ *Activate*. This application configuration loads the appropriate application environment (modules) and obtains the application's input schematic and syntax highlights, and subsequently creates a scheduler interface that greatly assists the user in creating, editing, and launching MOOSE application problem inputs.

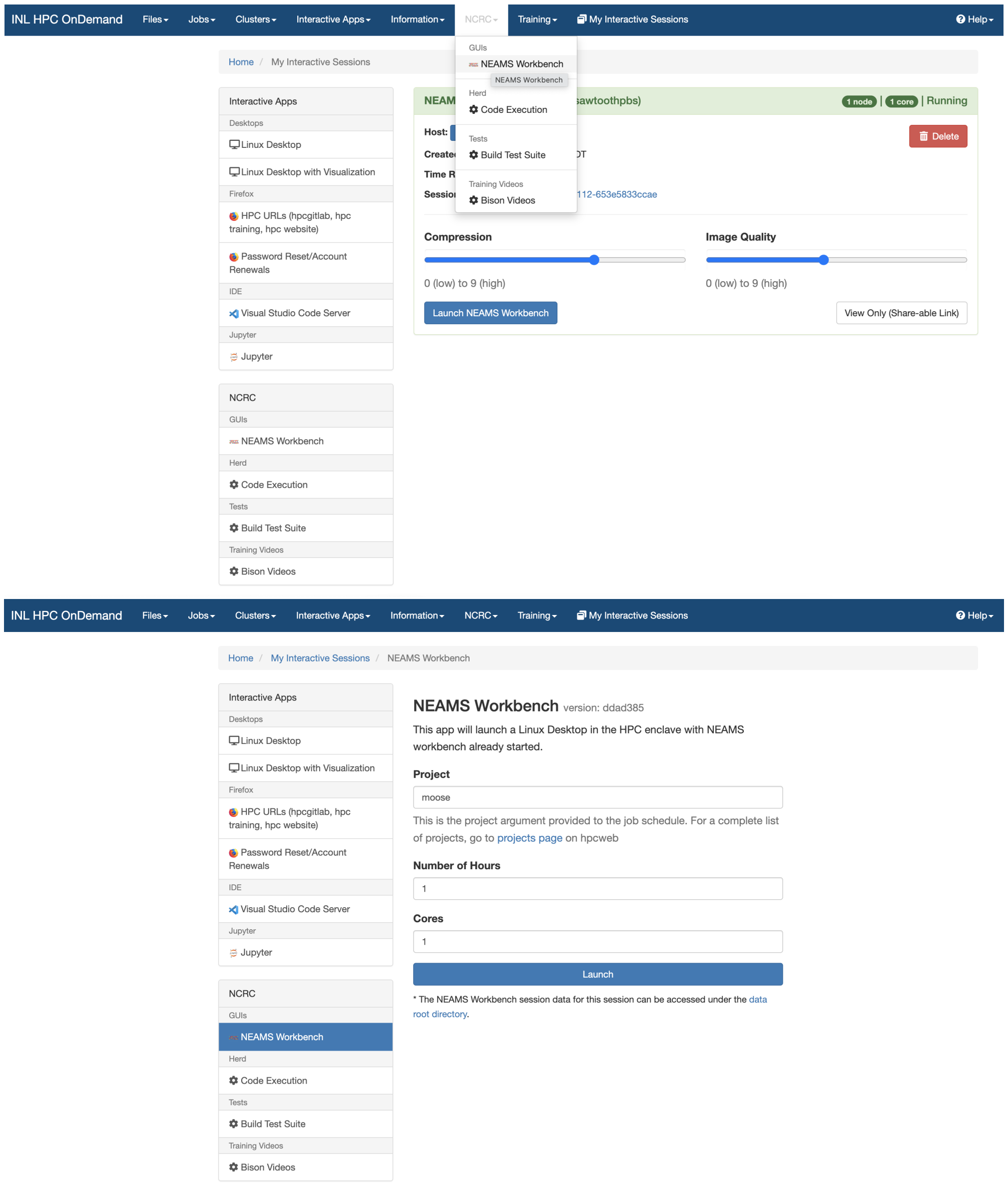
7.2 HPC-OnDemand Instructions

The central configuration control of MOOSE applications and the NEAMS Workbench on INL HPC-OnDemand allows users who have been granted access to log into *hpcondemand.inl.gov* and, via the following steps, launch a problem input to the NCRC cluster scheduler.

1. Log into *hpcondemand.inl.gov*.
2. Click the *NCRC* ≫ *GUIs* ≫ *NEAMS Workbench* menu item, specify the session time and desired core count, and click *Launch*, as shown in Figure 12.
3. With the NEAMS Workbench open, click the *File* ≫ *Localhosts...* ≫ *Application* ≫ *Activate* (Figure 13).
4. Open a desired application input (Figure 14).
5. With the activated application configuration selected, click the *Run* button and watch the problem be submitted to the cluster scheduler. The scheduler options can be modified by selecting *Customize Run Options* in the drop-down menu next to the *Run* button. Once the job completes, the Exodus output file is visualized with the embedded ParaView.

8. Triangulation Improvements

The libMesh library now incorporates Poly2Tri, a small open-source Delaunay triangulation library, as an optional sub-module. The basic triangulation capabilities therein have already been incorporated into the MOOSE reactors module, with new `MeshGenerator` classes that create such triangulations around—and stitches them together with—other user-constructed mesh geometries, allowing for simple creation of a reactor cross-section mesh that is then extruded into a simulation domain with hexahedra and triangular prism elements. The libMesh support for mesh stitching has also been broadened somewhat, allowing meshes to be stitched together when they have been only temporarily replicated between processors, with the library automatically



NEAM sawtoothpbs)

Host:

Create DT

Time R

Sessio 112-653e5833ccae

Compression

0 (low) to 9 (high)

Image Quality

0 (low) to 9 (high)

Home / My Interactive Sessions / NEAMS Workbench

NEAMS Workbench version: ddad385

This app will launch a Linux Desktop in the HPC enclave with NEAMS workbench already started.

Project

moose

This is the project argument provided to the job schedule. For a complete list of projects, go to projects page on hpcweb

Number of Hours

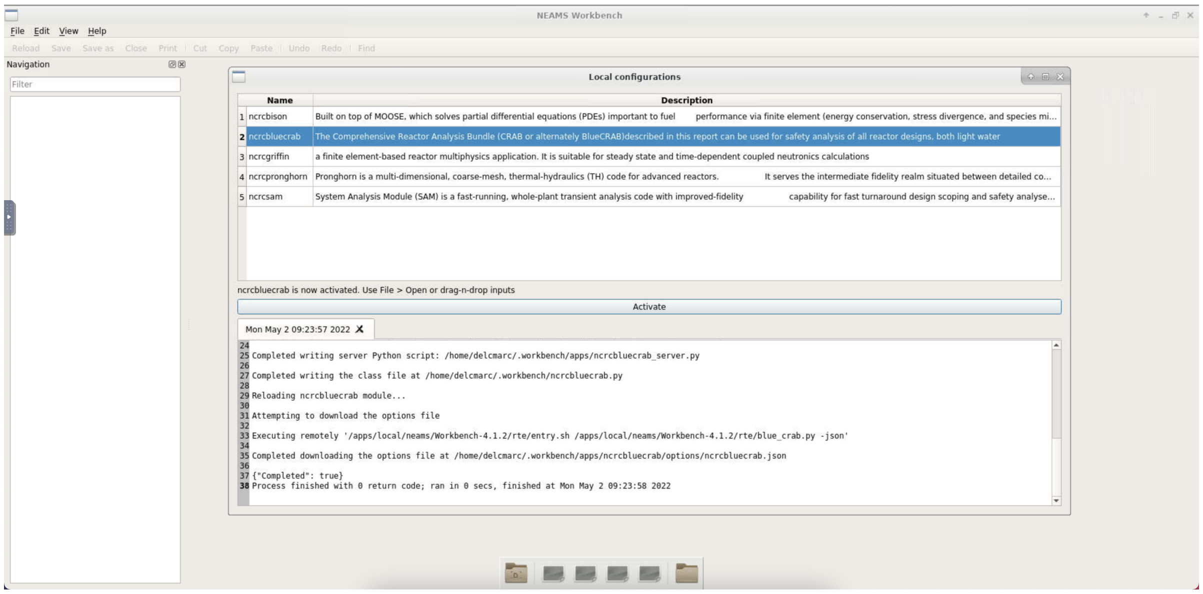
1

Cores

1

* The NEAMS Workbench session data for this session can be accessed under the data root directory.

Figure 12: NCRC GUI and selection of an interactive session with the NEAMS Workbench GUI.

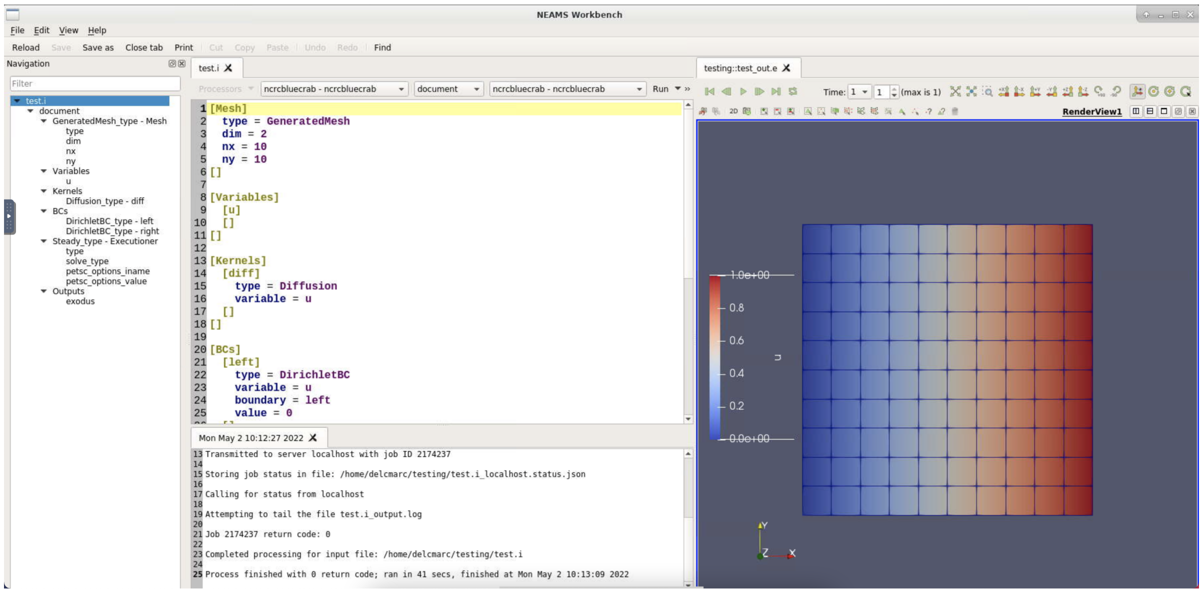


	Name	Description
1	ncrcbison	Built on top of MOOSE, which solves partial differential equations (PDEs) important to fuel performance via finite element (energy conservation, stress divergence, and species mi...
2	ncrcbluecrab	The Comprehensive Reactor Analysis Bundle (CRAB or alternately BlueCRAB)described in this report can be used for safety analysis of all reactor designs, both light water
3	ncrcgriffin	a finite element-based reactor multiphysics application. It is suitable for steady state and time-dependent coupled neutronics calculations
4	ncrcpronghorn	Pronghorn is a multi-dimensional, coarse-mesh, thermal-hydraulics (TH) code for advanced reactors. It serves the intermediate fidelity realm situated between detailed co...
5	ncrcsam	System Analysis Module (SAM) is a fast-running, whole-plant transient analysis code with improved-fidelity capability for fast turnaround design scoping and safety analyse...

ncrcbluecrab is now activated. Use File > Open or drag-n-drop inputs

```

Completed writing server Python script: /home/delcmarc/.workbench/apps/ncrcbluecrab_server.py

Completed writing the class file at /home/delcmarc/.workbench/ncrcbluecrab.py

Reloading ncrcbluecrab module...

Attempting to download the options file

Executing remotely '/apps/local/neams/Workbench-4.1.2/rte/entry.sh /apps/local/neams/Workbench-4.1.2/rte/blue_crab.py -json'

Completed downloading the options file at /home/delcmarc/.workbench/apps/ncrcbluecrab/options/ncrcbluecrab.json

{"Completed": true}
Process finished with 0 return code; ran in 0 secs, finished at Mon May 2 09:23:58 2022
```

Figure 13: Screenshot of the NEAMS Workbench with the localhost configurations window open.



```
[Mesh]
  type = GeneratedMesh
  dim = 2
  nx = 10
  ny = 10
[]

[Variables]
  [u]
  []
[]

[Kernels]
  [diff]
    type = Diffusion
    variable = u
  []
[]

[BCs]
  [left]
    type = DirichletBC
    variable = u
    boundary = left
    value = 0
```

```
Transmitted to server localhost with job ID 2174237

Storing job status in file: /home/delcmarc/testing/test.i_localhost.status.json

Calling for status from localhost

Attempting to tail the file test.i_output.log

Job 2174237 return code: 0

Completed processing for input file: /home/delcmarc/testing/test.i

Process finished with 0 return code; ran in 41 secs, finished at Mon May 2 10:13:09 2022
```

Figure 14: Screenshot of the NEAMS Workbench with a MOOSE input file open, run time output tab and ParaView visualization tab.

handling communication as needed if the meshes are in a distributed state when stitching is requested. This was prompted to support workflows such as the above, where stitching is done on a relatively small mesh (where the replicated stitching algorithm in libMesh is appropriate), but the eventual simulation is performed on a much larger (extruded and/or refined) mesh. Distributing the Mesh object, leaving only local elements and a a thin layer of "ghost" elements in memory on each processor, can now be done after stitching is complete but before the mesh is enlarged, making the enlargement much more CPU-efficient and the subsequent simulation much more memory-efficient.

Within libMesh, the old interface to Triangle (a popular triangulation library, unfortunately unusable with many MOOSE and other applications, due to licensing conflicts) was refactored to allow multiple concrete implementations of the same abstract interface, and an implementation using Poly2Tri at the base level was written to finally enable the use of this interface within MOOSE applications. The MOOSE framework user interface remains under development, but at the libMesh level, test coverage now exists for triangulations with boundaries and (optionally) holes defined by irregular polygons or by other meshes, with independent options for refining each outer or hole boundary, as well as interior refinement controlled by either a global or a spatially varying user-specified element area constraint.

9. Library Interfaces

9.1 libMesh

libMesh now supports a `SIDE_HIERARCHIC` finite element type, which implements a hierarchic space of discontinuous polynomial (in "master space") basis functions defined specifically on element sides rather than in element interiors. When a user instantiates it with polynomial order p, this basis is the product of, on each element side, the tensor product polynomial space Q^p. Support for triangular-sided elements with P^p function spaces may be added through future work. Domain-internal element side values are shared between the two elements that meet at that side, but the values are discontinuous between different sides meeting at the same mesh edge (or in 2-D, vertex).

The initial use case for this finite element basis was simply an easier-to-use and more automated form of data storage: in the base case of $p = 0$, the basis is piecewise constant, with a single real-valued (or complex-valued, in those configurations) datum associated with each side of each variable. By adding some number of `SIDE_HIERARCHIC` variables to an explicit auxiliary system associated with a mesh, an application code obtains data storage for the many variables on each mesh side. Unlike application-level storage, however, these variables are automatically kept in sync via mesh distribution and redistribution, and are made available to kernels at each side quadrature point, just as with any other solution or auxiliary variable, and require no C++ data structure coding to manage. libMesh support for the visualization output of these variables is also in development.

For future use cases, however, the ability to elevate the polynomial degree becomes important. Simple discontinuous-Galerkin-type finite element methods have always been supported in libMesh via a few discontinuous L_2 finite element types, but more sophisticated stabilized formulations that require, for example, inter-element flux variables defined on element sides are now attainable.

Extensive unit testing of these new spaces has been added to the libMesh test suite.

9.2 MetaPhysicL

This year, the `TIMPI::StandardType` interface to MPI data type creation, the `TIMPI::Packing` functions for variable-size data and non-contiguous container serialization, and the `TIMPI::OpFunction` interface to distributed-parallel data reduction operations were all overloaded within MetaPhysicL—specifically to support the numeric classes used for automatic differentiation within MOOSE. Much of the impetus for this work was specific support requirements of NEAMS Thermal/Hydraulics applications, but because the parallel capabilities were implemented at this low level, they are easily usable by any MetaPhysicL+TIMPI application, including any MOOSE application that performs parallel reduction operations on AD data types or containers including such types. The same interface is used for MetaPhysicL data as for any other numeric data, so a generic code calling `sum(x)` or `min(y)` will work as expected, even if invoked with both a simple floating-point number in one context and a `DualNumber` with a sparse gradient for AD in another context.

9.3 Templated Interface to MPI

When doing application testing with the expanded MetaPhysicL parallel capabilities, using various combinations of C++ containers along with various MOOSE automatic differentiation options, larger tests revealed some limitations in the TIMPI variable-data-size packing implementation. These have now been fixed, and additional test coverage added to the MetaPhysicL and TIMPI test suites to prevent regressions.

10. Initial Support of New Architectures

The Mac computing platform is a significant one for NEAMS tool development, and recent changes to the Mac ecosystem with the inclusion of ARM64-based processors developed by Apple (known as Apple Silicon) revealed implementation gaps and breakages within MOOSE, MOOSE-based tools, and dependencies for this architecture. As ARM64 is also an established architecture in the Linux computing space (and growing with the development of ARM-based platforms from vendors), it is important to gain initial support for this platform for scientific computation using MOOSE tools.

Leveraging the conda package management system [22] as the basis for MOOSE software dependencies has enabled the MOOSE ecosystem to react swiftly to platform changes and library breakages, as the third-party packages provided by the open-source `conda-forge` project are built with the widest possible compatibility [23]. This included initial support for Apple Silicon in 2021. Some software dependencies, such as the Message Passing Interface (MPI) C and FORTRAN compiler wrapper MPICH, were built manually at first in order to control architecture tuning and compiler settings more closely. However, as support matured in third-party packages, pre-built and widely available versions of MPICH and other dependencies such as the HDF5 data format could be used. This greatly simplified MOOSE package management and support, providing a more consistent software experience for all supported computing systems. Furthermore, greater consistency and reliability has enhanced the MOOSE development team's ability to provide support and enhancements to developers and end users of the MOOSE-based NEAMS tools, especially as those groups obtain new Apple Silicon platforms.

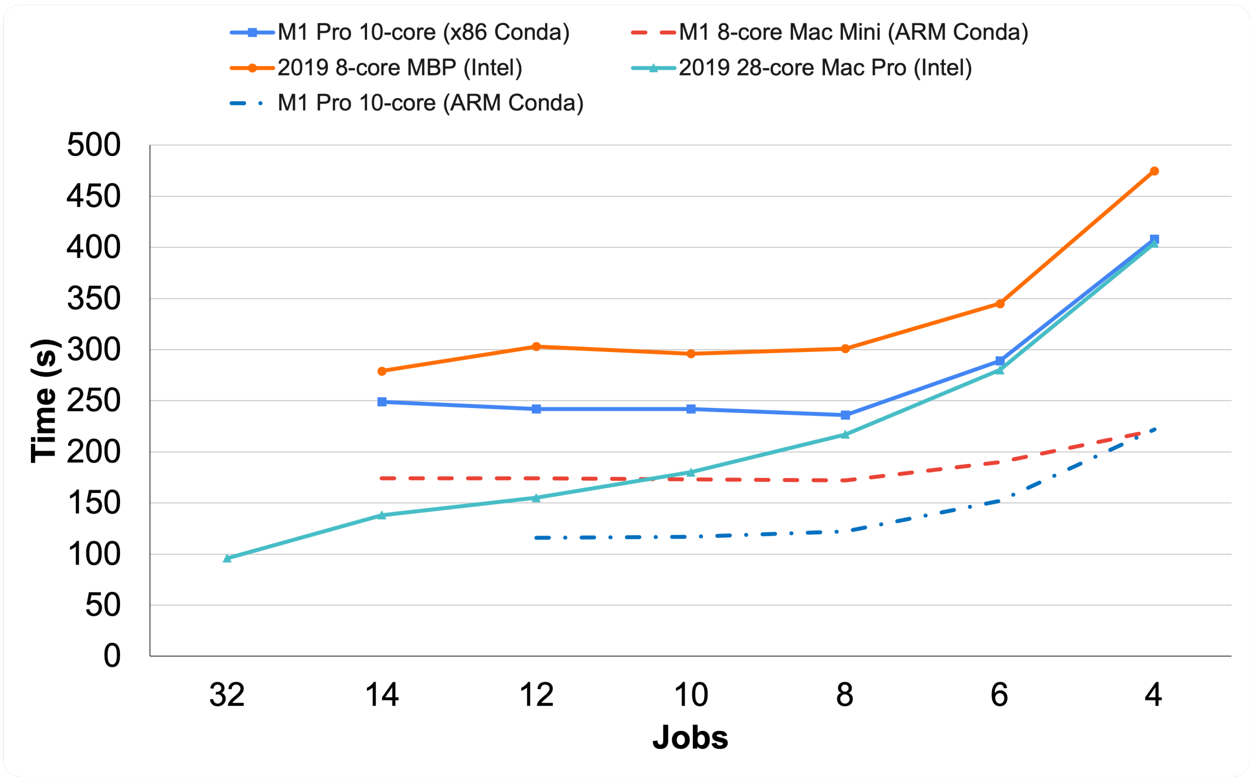


Figure 15: Comparison of Intel-based and Apple Silicon Macs for building the MOOSE framework. Solid lines represent the conda distribution for the x86 architecture, while dashed lines represent conda for ARM64.

10.1 System Profiling

To determine the impact of these new improvements on the user and developer experience, the time required to build MOOSE was profiled and compared across x86 and ARM64 architecture Macs. On the Intel side, this included comparisons of the 2019 MacBook Pro laptop (with an mobile 8-core Intel Core i9 processor), the 2019 Mac Pro desktop (with an 28-core Intel Xeon processor), the 2020 Mac Mini desktop (with an 8-core Apple M1 processor), and the 2021 MacBook Pro (with a mobile 10-core Apple M1 Pro processor). Because the Apple M-series processors can run programs based on x86 architectures using the Apple Rosetta translation and emulation layer, both x86 and native ARM64 versions of the conda distribution were compared. The results of this profiling are shown in Figure 15.

As shown in the results, the new 2021 MacBook Pro with the M1 Pro mobile processor running x86 software is very comparable to the desktop 2019 Mac Pro at up to the former's maximum number of cores (i.e., 10). This is particularly remarkable when remembering that the M1 Pro is running under emulation for this task. When running native ARM64 compilers, building MOOSE is achieved up to 45% faster (compared to the desktop), and up to 53% faster (compared to mobile), than both the Intel platforms and Apple x86 emulation. This performance boost will be vital for both users and developers as the Apple ARM64-compatible MOOSE software ecosystem grows.

REFERENCES

[1] C. J. Permann, D. R. Gaston, D. Andrš, R. W. Carlsen, F. Kong, A. D. Lindsay, J. M. Miller, J. W. Peterson, A. E. Slaughter, R. H. Stogner, and R. C. Martineau, "MOOSE: Enabling massively parallel multiphysics simulation," *SoftwareX*, vol. 11, p. 100430, 2020.

[2] 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.

[3] 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.

[4] MOOSE Team, "Moose github page." `https://github.com/idaholab/moose`, 2014.

[5] 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.

[6] 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.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-performance, portable implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, pp. 789–828, Sept. 1996.

[8] A. Lindsay, R. Stogner, D. Gaston, D. Schwen, C. Matthews, W. Jiang, L. K. Aagesen, R. Carlsen, F. Kong, A. Slaughter, *et al.*, "Automatic differentiation in metaphysicl and its applications in moose," *Nuclear Technology*, pp. 1–18, 2021.

[9] R. Freile, S. Harper, G. Giudicelli, and A. Abou-Jaoude, "Coupled griffin and pronghorn simulation of the molten salt fast reactor (msfr) for the virtual test bed," in *Transactions of the American Nuclear Society*, vol. 125, 2021.

[10] A. Abou-Jaoude, S. Harper, G. Giudicelli, P. Balestra, S. Schunert, N.Martin, A. Lindsay, and M. Tano, "A workflow leveraging moose transient multiphysics simulations to evaluate the impact of thermophysical property uncertainties on molten-salt reactors," *Annals of Nuclear Energy*, vol. 163, p. 108546, 2021.

[11] National Reactor Innovation Center, "Virtual test bed documentation." `https://mooseframework.inl.gov/virtual_test_bed/`, 2021.

[12] A. Abou-Jaoude, D. Gaston, G. Giudicelli, B. Feng, and C. Permann, "Overview of the virtual test bed," in *Transactions of the American Nuclear Society*, vol. 125, 2021.

[13] 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.

[14] G. L. Giudicelli, A. D. Lindsay, R. Freile, and J. Lee, "Neams-th-crab," Tech. Rep. INL/EXT-21-62895, Idaho National Laboratory, 2021.

[15] R. A. Lefebvre, B. R. Langley, L. P. Miller, M. L. Baird, and B. S. Collins, "M3ms-16or0401086 - Report on NEAMS workbench support for MOOSE applications," Tech. Rep. ORNL/TM-2021/2141, Oak Ridge National Laboratory, 2021.

[16] R. A. Lefebvre, B. R. Langley, and A. B. Thompson, "M3ms-16or0401086 - Report on NEAMS workbench support for MOOSE applications," Tech. Rep. ORNL/TM-2016/572, Oak Ridge National Laboratory, 2016.

[17] "LSP / LSIF. Official Page for Language Server Protocol." https://microsoft.github.io/language-server-protocol.

[18] P. J. Kowal, J. A. Eugenio, K. A. Dominesey, W. Ji, R. A. Lefebvre, and F. B. Brown, "Development of a syntactic validation capability for the use of MCNP," *Nuclear Science and Engineering*, 2021.

[19] B. R. Langley and A. R. Lefebvre, "Workbench analysis sequence processor and MOOSE framework parser benchmarks," Tech. Rep. ORNL/LTR-2022/16, Oak Ridge National Laboratory, 2021.

[20] R. A. Lefebvre, B. R. Langley, and J. P. Lefebvre, "Workbench analysis sequence processor," tech. rep., Oak Ridge National Laboratory, 10 2017.

[21] K. Cunningham, R. A. Lefebvre, J. Powers, L. P. Miller, M. L. Baird, and B. R. Langley, "NEAMS workbench and BISON fuel performance remote application configuration," Tech. Rep. ORNL/TM-2019/1208, Oak Ridge National Laboratory, 2019.

[22] "Anaconda software distribution," 2020. Computer software. Vers. 4-4.12.0. https://anaconda.com.

[23] conda-forge community, "The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem," July 2015.