INL/RPT-22-69424

NEAMS Technical Area Support in MOOSE

Sept. 2022

Nuclear Energy Advanced Modeling and Simulation M3 Milestone September 2022

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

[1]*COMPUTATIONAL FRAMEWORKS*

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

INL/RPT-22-69424

NEAMS Technical Area Support in MOOSE

Nuclear Energy Advanced Modeling and Simulation M3 Milestone
September 2022

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

[1]**COMPUTATIONAL FRAMEWORKS**

Sept. 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

Page intentionally left blank

CONTENTS

FIGURES

ACRONYMS

AD Automatic Differentiation

MOOSE Multiphysics Object-Oriented Simulation Environment

NEAMS Nuclear Energy Advanced Modeling and Simulation

TA Technical Area

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. The capabilities outlined in this report include a factor of 10^4 improvement in dependency-resolution speed, sorting of user objects, ability to compute residuals and Jacobians together, transfer fixes, support for the mortar method in finite volume discretizations, addition of generalized advection schemes for fluid simulations, 10^2 speedup in some Griffin simulations due to a new matrix-only solve type, and much more.

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.

3. GENERAL SUPPORT

3.1 Dependency Resolution Algorithmic Changes

Prior to #21108 MOOSE used an algorithm of unknown complexity in order to perform dependency resolution. The expense of this algorithm became clear when a user submitted an input with many initial conditions, which took 76 hours of computer wall-time to finish the simulation's initial setup. #21108 introduced a depth-first search algorithm for resolving

dependencies which is known to be of linear complexity based on the number of nodes and edges in the graph. After adopting the depth-first search algorithm, initial setup for the problematic input was reduced to a handful of seconds, a reduction in wall time by a factor of 1e4-1e5. The algorithmic change for the dependency resolver benefits all MOOSE applications.

3.2 User Objects Now Sorted

Back in 2016 a developer filed an issue that user object execution was determined by their input file ordering regardless of any dependency relation between the objects. This issue has been fixed in fiscal year (FY) 2022 after #21198. The key change is that user objects now inherit from the `DependencyResolverInterface` and consequently are sorted in contexts where sorting matters (e.g. when calling `execute` methods).

3.3 Computing the Residual and Jacobian Together

#19971 added the ability to compute the residual and Jacobian simultaneously. The motivation for this addition to MOOSE is based on the automatic differentiation capability which allows simultaneous computation of a function and its derivatives. Simultaneous computation of residual and Jacobian can be triggered for most MOOSE simulations simply be setting `residual_and_jacobian_together=true` in the `Executioner` block. Setting this parameter to `true` has shown 20% computational gains for some finite-volume Navier-Stokes simulations in which computation of Rhie-Chow coefficients requires derivative evaluation for every function evaluation. All other simulation types tested (e.g. Automatic Differentiation (AD)-based tensor mechanics and phase field simulations) have shown no gains from using the new option; in fact most have been slower. This can be attributed to the large cost of computing derivatives, which is traditionally dominated by sparsity union calculations. Normally, when MOOSE computes residuals (without simultaneous Jacobian calculation), `DualNumber` derivative computations are disabled by setting a static `do_derivatives` flag to `false`, so the "cost" of AD is negligible during residual evaluation. Consequently, for MOOSE's traditional nonlinear solution scheme, the cost of AD is N - 1 Jacobian evaluations, where N is the number of iterations taken by the nonlinear solver. However, when the residual and Jacobian are evaluated together, the AD cost is N Jacobian evaluations, or worse if a line search method is used. The cost of this additional

Jacobian evaluation can be amortized if N is large.

It is hypothesized that simultaneous computation of the residual and Jacobian may still be advantageous if the expense of material property evaluation is very high, in which case simultaneous evaluation of these properties would cut this cost roughly in half (assuming that the material properties are not functions of the nonlinear degrees of freedom; if they are, then the added cost of `DualNumber sparsity_union` calculations must be taken into account). In a situation where material property evaluation cost is high, the potential benefits of simultaneous residual and Jacobian evaluation would be felt regardless of whether the residual objects in the simulation are based on AD or have hand-coded Jacobians.

3.4 Transfers Fixes

#21748 fixed the caching of nearest nodes between the main app mesh and multiple subapp meshes. Previously, only the lowest rank subapp was properly cached, and this pull request expanded the cache to all subapps. This fix unearthed further issues when the number of processes involved in the simulation is greater than the number of subapps involved in the `MultiAppNearestNodeTransfer`, with mesh mapping caching. This was documented issue #20962 which will be fixed in future efforts. #21490 fixed a seg fault in the nearest node transfer that was happening when a rank did not own any subapp. The communication of bounding boxes (boxes formed around an entire app's mesh) was happening regardless of the ownership of a subapp. This pull request also enabled transfers to main variables and not just auxiliary variables through the `MultiApp appTransferVector` application programming interface (API). This allows modelers to leverage the `MultiApp` system to initialize nonlinear variables with additional transfers.

Preventing querying of degrees of freedom unavailable on a process within `MultiAppNearestNodeTransfer`. This bug was unearthed in trace-trace coupling tests in BlueCRAB after some unrelated refactoring of `MultiAppNearestNodeTransfer` to support MOOSE coordinate transformation objects. The bug was only triggered when the user set the `fixed_meshes` parameter to `true`.

3.5 Memory Pools for Automatic Differentiation

MOOSE currently uses an automatic differentiation derivative container that is statically sized (e.g. its size is determined during the configuration part of the build phase as opposed to at run time with a dynamically sized container). The statically sized container leads to optimal computer performance as no dynamic memory creation or management is involved; however, it requires users to be aware of the sparsity pattern sizes in their simulation, and to configure MOOSE accordingly. When a user does not consider their sparsity pattern in their MOOSE configuration, they will have exceptions thrown during their simulation when their derivative container size is exceeded. In FY-22 we wanted to explore a dynamic container that would allow users to execute application binaries without consideration of their sparsity pattern size (e.g., we wanted user AD simulations to "just work"). In past explorations, dynamic container sizes had performed very poorly relative to statically sized containers with the former taking more than ten times longer on a Navier-Stokes test case. Profiling suggested that dynamic memory allocation every time a dual number was constructed was the major contributor to the dynamic container's poor performance. To improve performance we hypothesized that using a memory pool, to which we would return properly sized dynamic containers post-dual number destruction and to which we would draw from when constructing new dual numbers, would drastically reduce dynamic memory allocation expense. In fact, we were successful in this regard as shown through comparison of Figure 1 and Figure 2. The former figure shows a CPU profile executing with a pool-based dynamic container while the latter figure shows a CPU profile with a "naive" `std::vector`. The naive dynamic implementation, `operator new`, represents dynamic memory allocation as a significant callee of `std::vector::resize`, whereas it does not appear as a significant timing fraction of `std::vector::resize` for the pool-based implementation. However, CPU profiling of the naive dynamic container shows, at least for the tensor mechanics problem tested, that dynamic memory allocation is no longer an important factor in the simulation time, although it was a dominant factor in previous years. And because of the significant fraction of time spent in acquiring containers from the memory pool, the pool-based dynamic container is actually two times slower than the naive dynamic container implementation. Based on these findings we decided not to merge the pool-based dynamic container into MetaPhysicL. However, the memory

pool exploration task was not without success: its exploration showed that the naive dynamic container implementation performance is now within reason relative to the static container currently used in MOOSE (within 40% on the test problem; compare Figure 2 and Figure 3). Based on this finding, we may decide to add the dynamic container as a configuration option available to users. Users who configure with the dynamic option would experience slightly slower run-times but would have greater flexibility in the simulation types they can run. That is, a user could run with an arbitrarily large (or small) sparsity pattern and have no exceptions due to exceeded (static) container size.

3.6 Miscellaneous

Miscellaneous work in general support of applications includes:

- A number of vector postprocessors did not have proper re-initialization of their data vector, which led them to have the default "complete history" output. This was fixed across the framework in #20810 and #20956

- A number of mesh generators were not checking properly for the parallelism of the mesh. They did not support distributed execution, but failed to inform the user and risked improper execution. This was fixed by #21173.

- Among other documentation improvements, #20956 added documentation for loading split meshes with a `FileMeshGenerator`. This enables the execution of mesh generators on split meshes, or the execution of re-partitioners on those splits. This is largely enabling simulations of very large 2D-extruded-to-3D geometries.

- Implementing variable checks for boundary restricted objects. This new integrity check examines whether a variable has degrees of freedom along the entire boundary of a boundary restricted object. The desire for this check came from a user who wasted a lot of time unsuccessfully trying to determine why they were getting out of bounds indexing in `MooseArray`, when the root cause was use of a block-restricted variable on a boundary-restricted object for which the variable only had degrees of freedom along

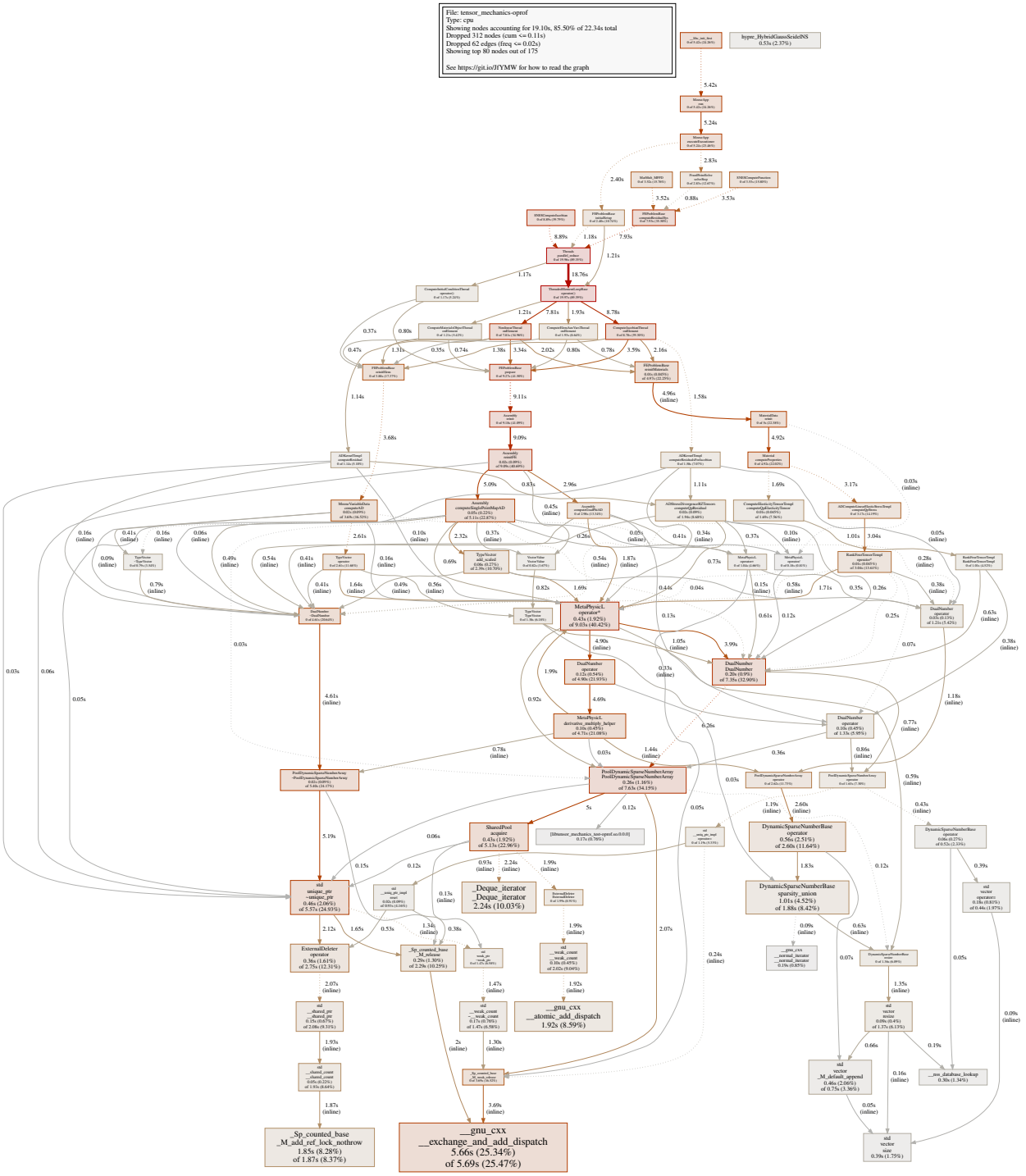

Figure 1: CPU profiling graph for a tensor mechanics simulation using the memory pool-based dynamic sparse number array derivative container. Overall simulation time for this container type was 22 seconds.

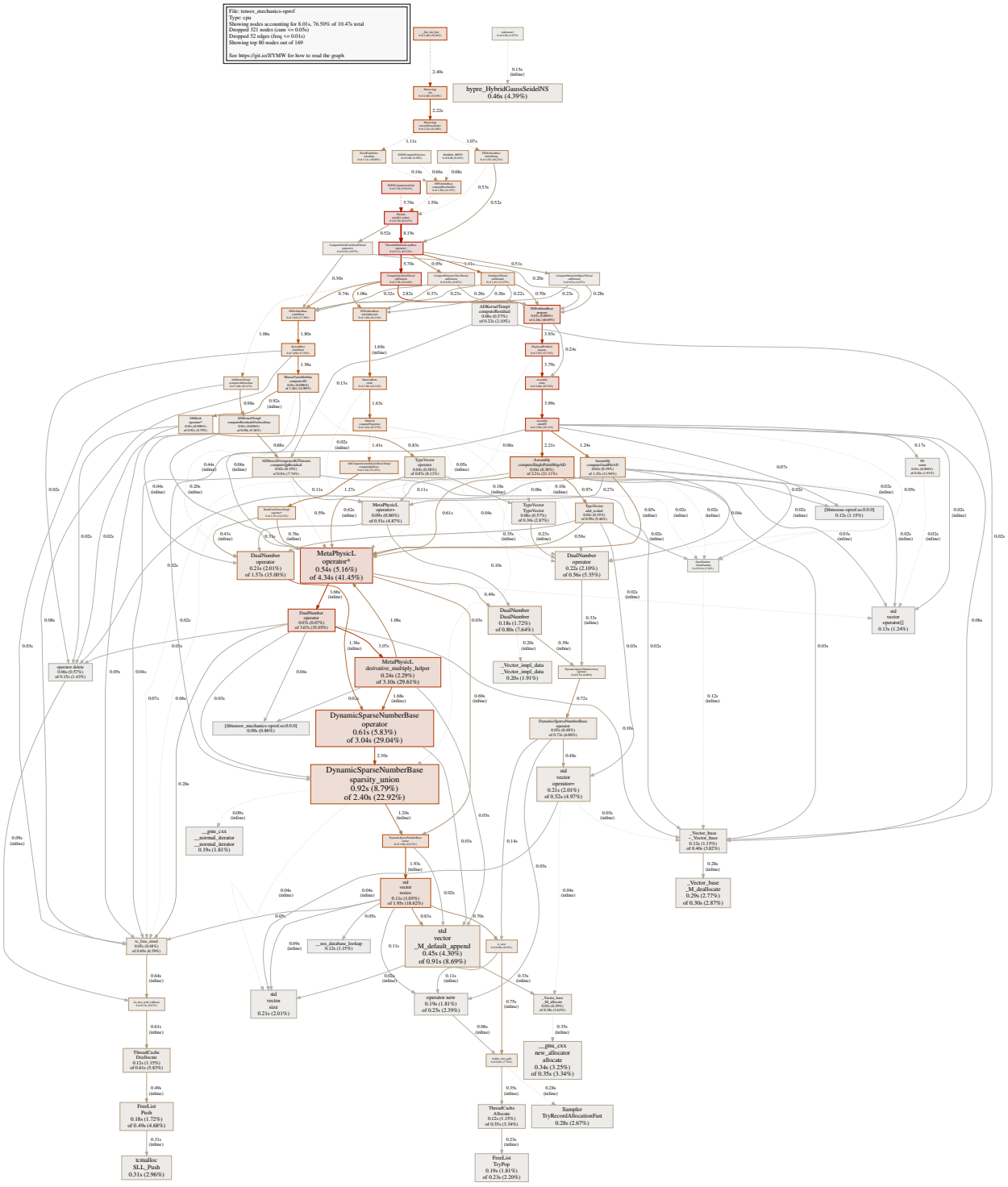

Figure 2: CPU profiling graph for a tensor mechanics simulation using the dynamic sparse number array derivative container. Overall simulation time for this container type was 10 seconds.

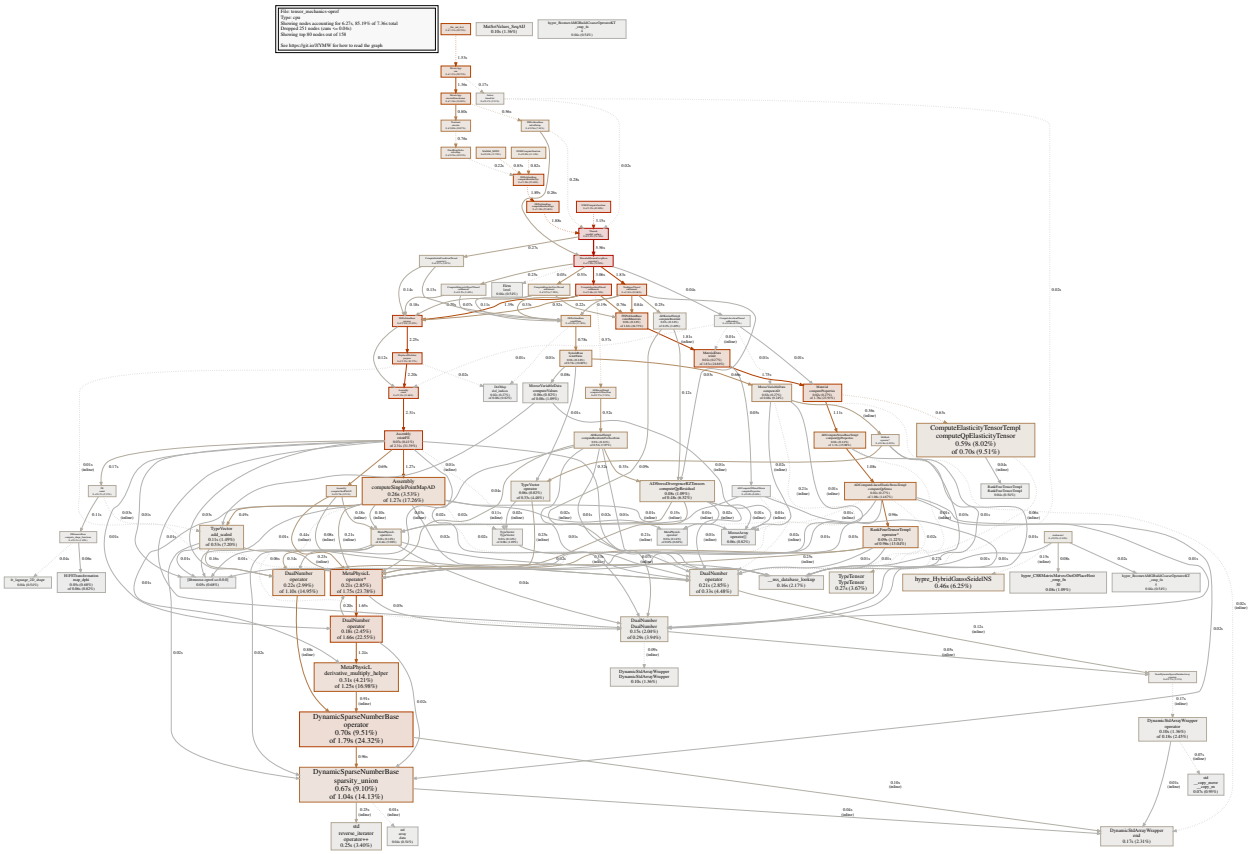

Figure 3: CPU profiling graph for a tensor mechanics simulation using the statically sized sparse number array derivative container. Overall simulation time for this container type was 7 seconds.

part of the boundary. Objects included in these integrity checks are `AuxKernel`, `NodalBC`, `IntegratedBC`, `NodalUserObject`, and `SideUserObject`.

- #20068 added documentation for installing MOOSE on Idaho National Laboratory (INL) High-Performance Computing (HPC). Help installing MOOSE on HPC was a recurrent request of INL users, as the build environment is significantly different from one's personal workstation or laptop.

- #20068 also made sure that users could always use block names for mesh generation, instead of only working with block IDs. Block IDs are generally less intuitive than names and often require the user to examine the mesh before making a determination for every input.

- A number of pull requests completed this year requested the object documentation of framework objects in MOOSE. Over one-quarter of all MOOSE object documentation pages were turned from helpful but minimal stubs to full-blown documentation, with detailed explanations and examples where required.

- Created a command line parameter for specifying which application to run an input file. This capability is important for large conglomerate applications like BlueCRAB. Some applications' syntax do not play well with other applications, such that if a user conceptually wants to execute application A, application B's syntax, if pulled into the simulation via aggregate registration, may degrade or lead to unexpected results. By using the `--app FooApp` command line parameter, users can be confident that they will only be using the registered objects and syntax associated with `FooApp` and consequently will have identical behavior using an aggregate application binary like BlueCRAB as they do when using a standalone application binary such as SAM, Pronghorn, or Griffin.

4. PRONGHORN SUPPORT

4.1 Support For Mortar With Finite Volume Discretizations

The mortar method has been supported for finite elements in MOOSE since 2019 and has been used successfully for thermal and mechanical contact. With the rise of the finite volume method in MOOSE and Pronghorn for modeling fluid flow, performing thermal contact within finite volume

discretizations has become necessary to accurately account for energy transport within advanced reactor designs. In that vein, support for finite volume within the mortar method was added in #21658. Adding this new capability was fairly straightforward, requiring only two significant additions.

The first addition was a new spatial argument type for calling MOOSE `Functor` objects. The new type, `ElementPointArg`, is a composition of an element and a point within the element. When called with an `ElemPointArg`, variable functors perform a two-term Taylor expansion corresponding to

$$f = f(\vec{r}_C, t) + \nabla f(\vec{r}_C, t) * (\vec{r}_P - \vec{r}_C) \tag{1}$$

where $\vec{r}_C$ is the element vertex-average (roughly the element centroid) and $\vec{r}_P$ is the user-supplied evaluation point. Function functors simply evaluate the function at the provided point, foregoing the two-term expansion.

The second addition was a `ghostHigherDNeighbors` method on the mortar relationship manager class `AugmentSparsityOnInterface`. This method ghosts the higher dimensional neighbors of elements along the mortar boundary, which is necessary for computing the Green-Gauss gradient for the two-term expansion in Equation (1), which renders the finite volume mortar discretization second order accurate. Verification of second order accuracy is shown in Figure 4 for a gap conductance problem. A plot of the temperature field for a gap conductance problem on a relatively coarse mesh is shown in Figure 5.

4.2 Generalized Advection Schemes For Rhie-Chow-Based Navier-Stokes Simulations

Prior to FY-22 the only supported interpolation schemes for evaluating face quantities for advection were upwind and central-difference. However, in FY-22 we added support for some common interpolation schemes, including:

- van Leer
- minmod

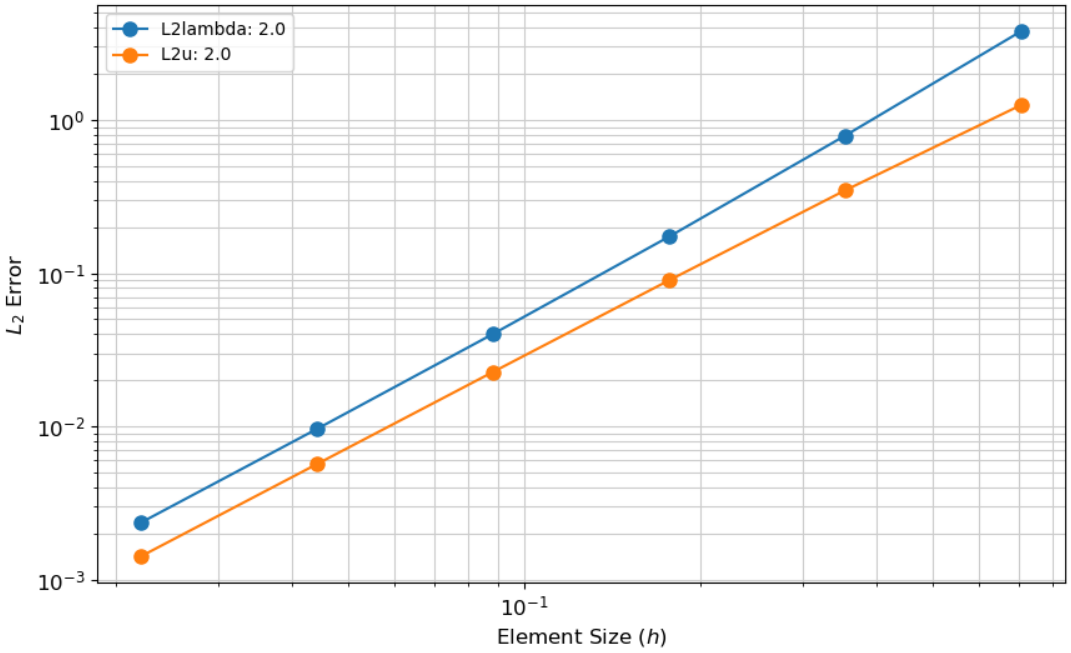

Figure 4: Verification of the second-order accuracy of a finite volume mortar discretization of a gap conductance problem

- second-order upwind (SOU)
- QUICK.

Total variation diminishing schemes like van Leer and minmod are necessary for problems where solution fields have steep gradients and high Reynolds numbers for which respectively upwind is too inaccurate and central differencing is too unstable and oscillatory. Figure 6 illustrates solution of a Ra $= 10^6$ problem (where Ra denotes the Rayleigh number) using a minmod limiter. The minmod scheme accurately captures the steepd velocity gradients near the domain boundary. Note that the implemented limiters are scalar limiters, so limiting the velocity field is done on a component-by-component basis. Vector limiting schemes exist and are more stable (e.g., more diffusive). We plan to implement vector limiting in future work.

4.3 Miscellaneous

Miscellaneous work supporting Pronghorn in FY-22 includes:

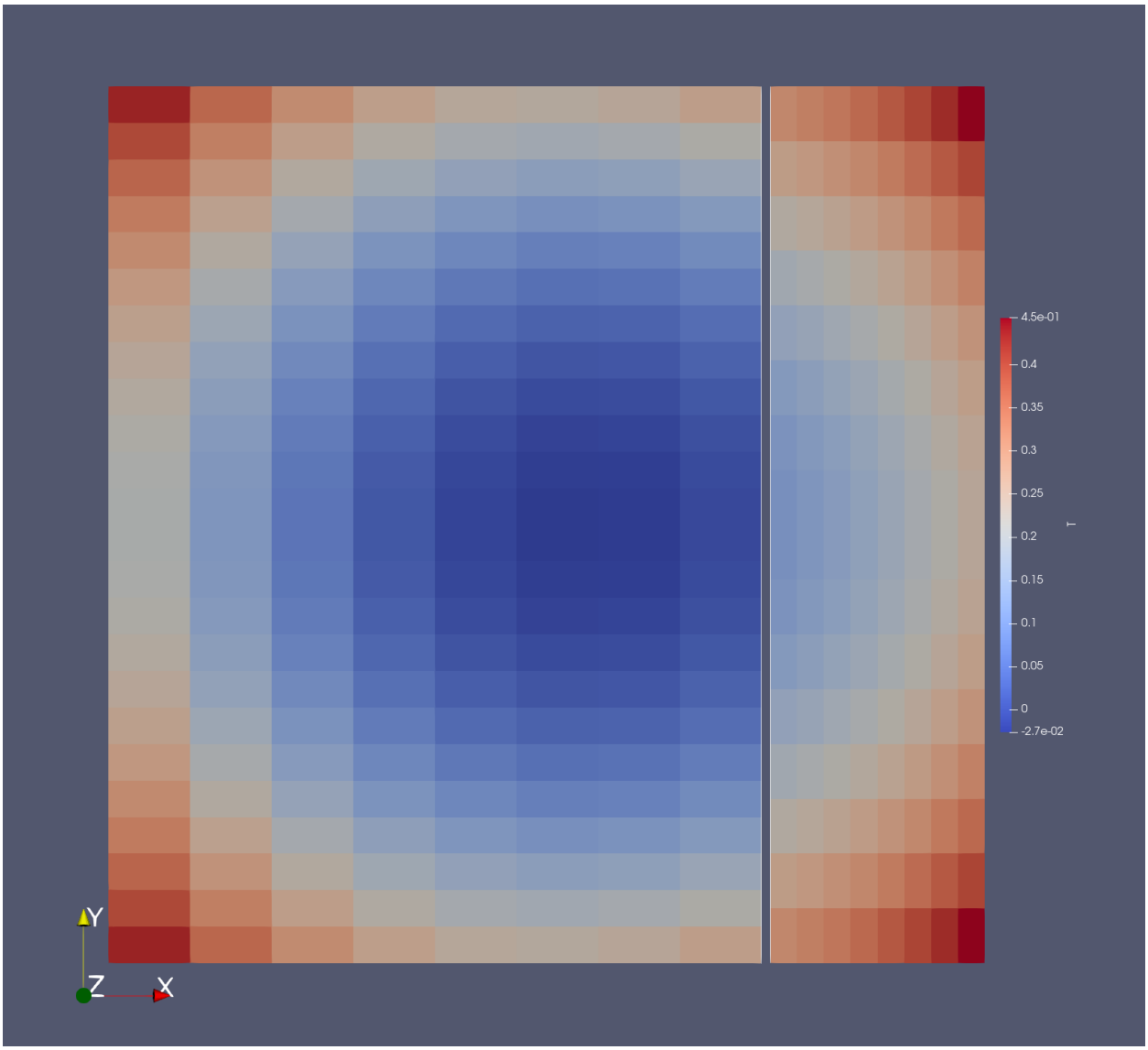

Figure 5: Temperature plot for basic gap conductance problem on a coarse mesh

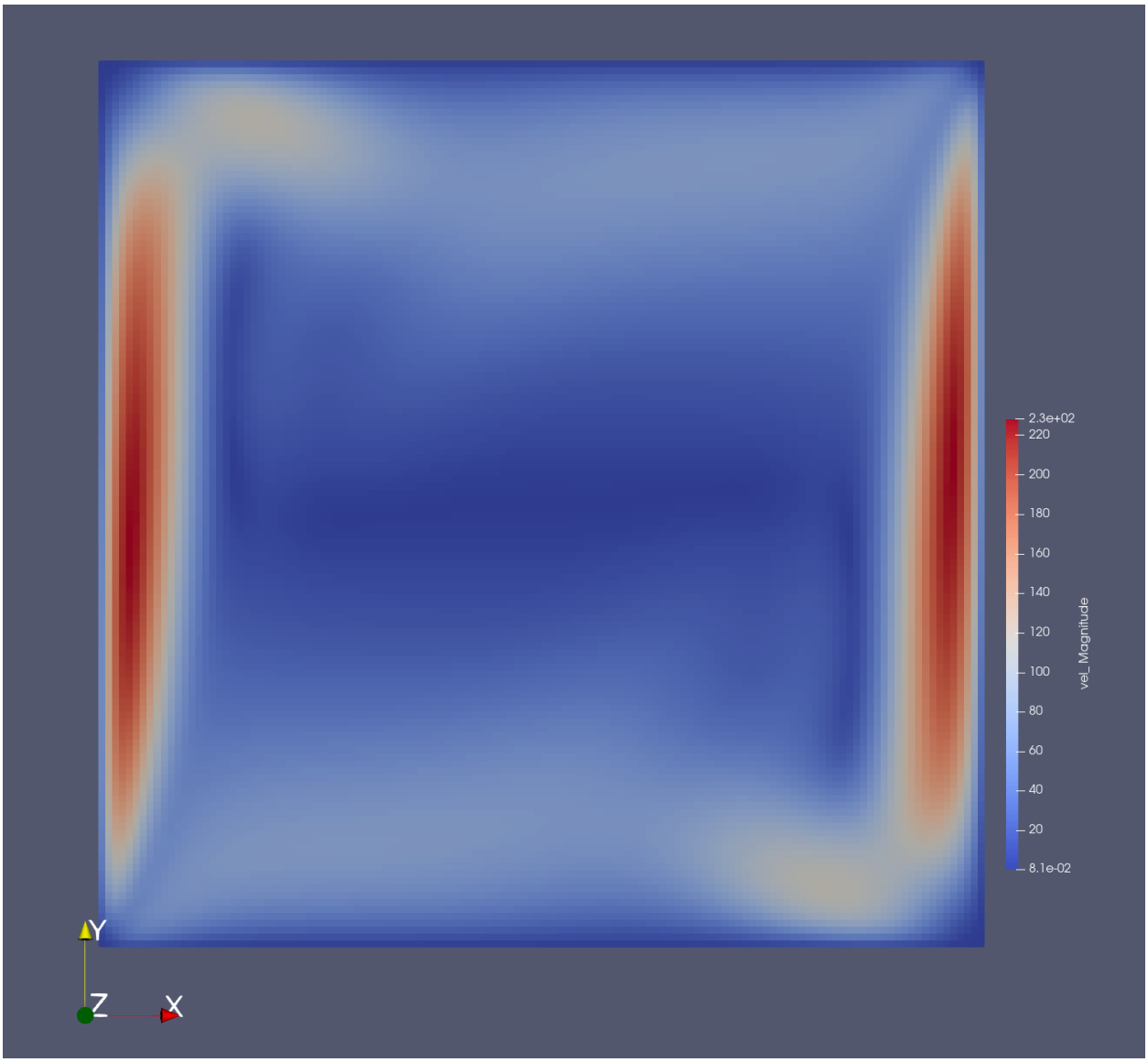

Figure 6: Plot of the velocity magnitude for Ra $= 10^6$ in which advected quantities are interpolated using a minmod limiter

- Allowing use of the `GradientJumpIndicator` with finite volume variables. The change in gradient, a measure of the Laplacian, is a common metric on which to base mesh adaptivity. In areas of high second derivative, it can be helpful to refine the mesh to accurately capture the physics while leaving regions in which the solution field has low second derivative (smoother regions) relatively coarse. This adaptivity strategy allows accurate simulations without incurring inordinate computational cost. The changes introduced in this pull request were leveraged in natural convection models of spent nuclear fuel cask storage.

- Added `ReynoldsNumberFunctorAux`. The Reynolds number is a defining quantity of fluid flow modeling. It informs decisions on whether to include turbulence models or stabilization in a discretization scheme. For instance, for element/cell Reynolds numbers much above one, discretization of the advection operator should include upwinding to ensure stability and reduce or remove oscillations in the solution field. With the addition of this new auxiliary kernel, users of Pronghorn can make more-informed decisions on turbulence and advection discretization schemes.

- Generalization of the finite element Navier-Stokes implementation to support an x-axis axis-of-symmetry for RZ simulations. Previously only a y-axis axis-of-symmetry was supported and user selection of an x-axis axis-of-symmetry silently led to incorrect results or poor nonlinear convergence.

- Implemented caching for finite-volume-based argument evaluation of functors. This can be advantageous if functor evaluation is very expensive although it uses non-trivial amounts of memory.

5. GRIFFIN SUPPORT

5.1 Enabled Matrix-Only Solve Type For Eigenvalue Executioner

In #18527 we added a new option for computing matrix-vector products when performing preconditioned Jacobian-Free Newton-Krylov Eigenvalue solves. Instead of approximating matrix-vector products by doing differencing of various user-provided function evaluations, the matrix-vector product is explicitly computed using the user-provided A and B matrices.

This avoids calling the potentially expensive user residual function. This option is only relevant for applications in which the A and B matrices are constant. Its computational savings are proportional to the number of linear iterations in a linear solve/nonlinear iteration and consequently has led to 100–1000× speedups in some Griffin simulations.

5.2 Robust Ghosting of Lower Dimensional Variables for SideUserObjects

In FY-22, Griffin developers noticed segmentation faults in parallel when attempting to access lower-dimensional degrees of freedom from "downwind" elements. This bug was due to inadequate ghosting of lower-dimensional elements. To fix this issue we created a new `GhostLowerDElems` relationship manager class in #21699. Addition of the `GhostLowerDElems` relationship manager triggered remote element deletion for more distributed mesh simulations, which unearthed inadequate ghosting for Bison simulations using mortar constraints. This led to creation of another new relationship manager class `GhostHigherDLowerDPointNeighbors` as well as more robust communication of required element couplings within the mortar mesh generation class. We also moved creation of the initial mortar mesh to before initialization of the libMesh `EquationSystems` object, making it possible to avoid calling the expensive `EquationSystems::reinit` method. Motivated by the discovery that adding new relationship managers incurred remote element deletion, which had not occurred before, we attempted to programatically delete remote elements for all distributed mesh simulation types; however, this incurred multiple failures in the MOOSE test suite. In the future we plan to pursue this further, as described in issue #21798.

6. BISON SUPPORT

6.1 Dependency-Based Reinitialization of Material Properties For Mortar Constraints

Materials that use stateful material properties are not evaluated when computing mortar constraints due to the nature of the mortar mesh; the number of quadrature points is constantly changing as the mesh displaces and the projections and inverse projections of secondary and

primary mortar interfaces change. However, a user reported that they were hitting segmentation faults due to execution of materials which did not use stateful properties but *did* use non-stateful material properties computed by materials that also used stateful properties (e.g., materials were executing which did not have their dependency materials executed). We were able to circumvent this issue by adding utility functions that query consumer objects for required material properties, and then building a list of materials which provided the required properties, as well as materials that provided properties used by the consumer-required materials. In summary we were able to selectively build a list of materials to execute based on need as opposed to blindly reinitializing all materials. These utility functions were added in pull request #20104. In the future we plan to incorporate these utilities into all places where we evaluate materials to avoid performing unnecessary, potentially expensive, material property evaluations.

6.2 Miscellaneous

Miscellaneous support tasks for BISON in FY-22 included:

- Not bypassing threads during initial stateful material property evaluation. A Bison merge request showed an exodiff on an initial auxiliary variable computation when run with threads due to the fact that thread ID 1 was retrieving a stateful-property dependent porosity which had not been initialized on thread ID 1. By running the initial stateful property initialization on all threads (as opposed to just thread 0), the bug was fixed.

- Allowing coincident nodes on mortar boundaries. Disconnected meshes may be generated using the `BreakMeshByBlockGenerator` which takes an initially connected mesh and breaks it along a boundary, creating duplicate overlaid nodes in the process. Before the fix in #21683, the mortar mesh was not being created correctly because, due to the perfect overlay of nodes, a primary element candidate kitty-corner to a given secondary element was considered and an invalid orientation was determined, generating a hard error. To fix the issue, instead of generating a hard error, the bad primary element candidate was simply rejected and eventually the primary element coincident with the secondary element was correctly selected for pairing.

- Adding back higher-dimensional secondary-primary coupling to the mortar sparsity pattern. This change was necessary to prevent computationally expensive matrix reallocations when performing penalty-based mortar. In the past all mortar constraints had been based on Lagrange multipliers, which does not have coupling between the primal degrees of freedom on the secondary and primary sides.
- Adding more potential matches for regex of input files. This made sure that example input files are properly attributed to the corresponding documentation pages for templated classes, especially with AD (automatic differentiation) and non-AD versions of the same class. This templating is increasingly spread across Bison and the relevant MOOSE modules.
- Allowing existence of mortar meshes without corresponding mortar constraints. This was a necessary change in order to allow users to execute mortar-based auxiliary kernels in a reference mesh configuration while running mortar constraints on the displaced mesh.

7. Conclusion

This report highlighted various improvements made to the MOOSE framework in direct support of NEAMS applications. Some of the highlights include a factor of 10^4 improvement in dependency-resolution speed, sorting of user objects, ability to compute residuals and Jacobians together, transfer fixes, support for the mortar method in finite volume discretizations, addition of generalized advection schemes for fluid simulations, 10^2 speedup in some Griffin simulations due to a new matrix-only solve type, and much more. Due to the quickness and number of improvements, technical area support remains a very popular item among application developers in the NEAMS program.

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.