The Journal of Open Source Software

GLUE Code: A framework handling communication and interfaces between scales

Aleksandra Pachalieva [1,2], **Robert S. Pavel** [3¶], **Javier E. Santos** [1,2], **Abdourahmane Diaw** [4], **Nicholas Lubbers** [3], **Mohamed Mehana** [2], **Jeffrey R. Haack** [3], **Hari S. Viswanathan** [2], **Daniel Livescu** [3], **Timothy C. Germann** [5], **and Christoph Junghans** [3]

1 Center for Non-Linear Studies, Los Alamos National Laboratory, Los Alamos, 87545 NM, USA **2** Earth and Environmental Sciences (EES) Division, Los Alamos National Laboratory, Los Alamos, 87545 NM, USA **3** Computer, Computational and Statistical Sciences (CCS) Division, Los Alamos National Laboratory, Los Alamos, 87545 NM, USA **4** Fusion Energy Division, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN 37831, USA **5** Theoretical Division, Los Alamos National Laboratory, Los Alamos, 87545 NM, USA ¶ Corresponding author

DOI: 10.21105/joss.04822

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @govarguz
- @keipertk

Submitted: 20 September 2022
Published: 23 December 2022

Summary

Many scientific applications are inherently multiscale in nature. Such complex physical phenomena often require simultaneous execution and coordination of simulations spanning multiple time and length scales. This is possible by combining expensive small-scale simulations (such as molecular dynamics simulations) with larger scale simulations (such continuum limit/hydro solvers) to allow for considerably larger systems using task and data parallelism. However, the granularity of the tasks can be very large and often leads to load imbalance. Traditionally, we use approximations to streamline the computation of the more costly interactions and this introduces trade-offs between simulation cost and accuracy. In recent years, the available computational power and the advances in machine learning have made computing these scale-bridging interactions and multiscale simulations more feasible.

One driving application has been plasma modeling in inertial confinement fusion (ICF), which is fundamentally multiscale in nature. This requires deep understanding of how to extrapolate microscopic information into macroscopically relevant scales. For example, in ICF one needs an accurate understanding of the connection between experimental observables and the underlying microphysics. The properties of the larger scales are often affected by the microscale behavior incorporated usually into the equations of state and ionic and electronic transport coefficients (Liboff, 1959; Rinderknecht et al., 2014; Rosenberg et al., 2015; Ross et al., 2017). Instead of incorporating this information using reliable molecular dynamics (MD) simulations, one often needs to use theoretical models, due to the inability of MD to reach engineering scales (Glosli et al., 2007; Marinak et al., 1998). One approach to resolve this issue is by coupling two MD simulations of different scales via force interpolation, e.g., the AdResS method (Krekeler et al., 2018; Nagarajan et al., 2013). Another approach, which we will pursue in the scope of this work, is by enabling scale bridging between MD simulations and meso/macro-scale models through the development and support of application programming interfaces that these different applications can interact through.

State of the art

Traditionally, multiscale simulations combine multiple simulation methods that need to be simultaneously executed and coordinated. To achieve this, we use asynchronous task-based

runtime systems that include load balancers. Such load balancers can schedule and migrate tasks to maintain throughput; however, the issue of fault tolerance remains (Cappello, 2009). To avoid this, checkpointing is often used (Koo & Toueg, 1987), but it could be prohibitively expensive when its output frequency is high. Other works such as the `mystic` framework (Michael McKerns & Aivazis, 2019) is more geared towards large-scale machine learning techniques (McKerns et al., 2011). Such codes are not designed toward coupling multiscale simulations but have been demonstrated to be highly effective at solving hard optimization problems.

In this work, we propose the Generic Learning User Enablement (GLUE) Code to facilitate the coupling between scales. The GLUE Code builds upon previous work on multiscale coupling (Haack et al., 2021; Karra et al., 2022; Lubbers et al., 2020; R. Pavel et al., 2017; R. S. Pavel et al., 2015). At its simplest, we determine what physical properties must be exchanged between the various scales and derive application programming interfaces (APIs) from these.

To take advantage of modern supercomputing and exascale platforms and we couple our framework directly with high-performance-computing-friendly job schedulers. This allows us to use one framework to generate training data with a minimal footprint (task scheduler can fire off all jobs) and perform "hero-runs" of multiscale simulations. At every point, we perform a fine grain simulation or utilize an upscaled result from the active learning model. The GLUE Code uses active learning to evaluate the level of uncertainty and executes fine grain simulations when more training data is required.

Statement of need

The GLUE Code is a modular framework designed to couple different scientific applications to support use cases, including multiscale methods and various forms of machine learning. The workflow was initially designed around supporting active learning as an alternative to coupled applications to support multiscale methods but has been written in a sufficiently modular way that different machine learning methods can be utilized. Application programming interfaces (API) are available for C, C++, Fortran, and Python with support for various storage formats. The code also provides direct coupling with high performance computing job schedulers such as SLURM (Yoo et al., 2003) and Flux (Ahn et al., 2018). An overview of the GLUE Code implementation is shown in Fig. 1, where the meso/macro-scale solvers BGK, CFDNS, and LBM are also defined.

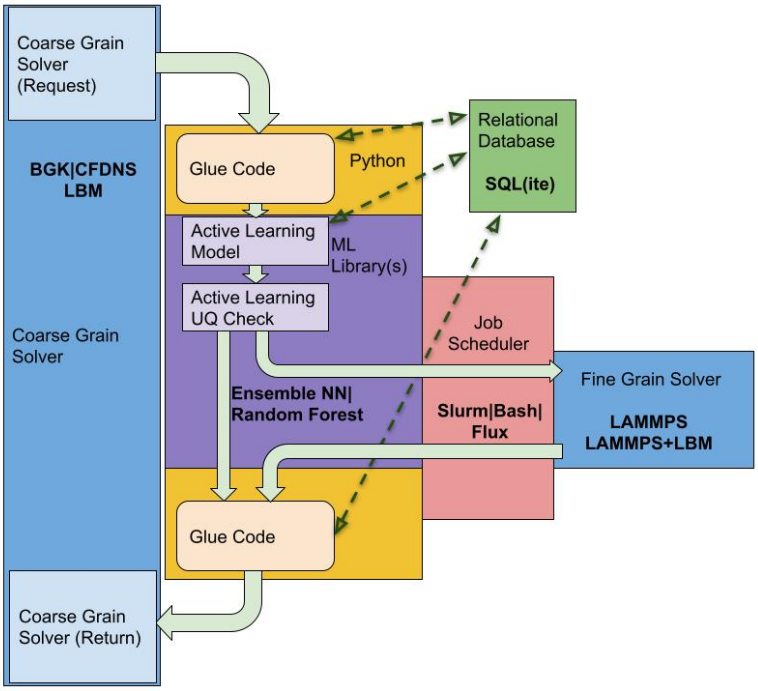

Figure 1

Figure 1: Sample of the GLUE Code implementation of our microscale-macroscale coupling. On the macroscale simulation (left) sending a request, the GLUE Code (center) uses the active learning algorithms (center purple) to determine if the model's uncertainty quantification is such that a new fine grain simulation (right) needs to be called. Then either the result of the fine grain simulation or the model's prediction is returned to the macroscale simulation. This figure is adapted from Fig. 2 in Diaw et al. (2020).

The aim of the GLUE Code is to efficiently handle the communication and interfaces between the computing platform, surrogate model, coarse-scale code, and fine-scale code (Diaw et al., 2020). At its core, the GLUE Code determines what physical properties need to be exchanged between the scales of interest and spawns APIs using this information. The GLUE Code ensures that these physical properties are communicated between the various scientific codes needed for the given multiscale problem. The GLUE Code uses active learning algorithms to evaluate if the model has a high level of uncertainty, and thus, a fine grain simulation needs to be executed. At the end of each cycle, the framework returns either the result of the fine grain simulation or the model's prediction of the macroscopic simulation.

Characteristics of the GLUE code:

- The GLUE Code has a modular lightweight structure, where each component can be swapped out for a different implementation. This is possible due to the APIs structure of the framework.
- The modularity of the GLUE code allows us to replace a scientific solver with a machine learning solution. The scale bridging structures are explicitly defined, which allows us to switch between a fine grain MD simulation and an ML solution. If a neural network is capable of reproducing the same outputs as an MD simulation from the same inputs, then they would be functionally identical.
- As shown in Fig. 1, the main components of the GLUE Code are a Coarse Grain Solver, an active learning backend, a relational database, a job scheduler, and the coupling and coordination logic.

- The GLUE Code is written with a collection of commodity software as the backend to most of these components. Currently, the relational database is supported via SQL(ite), while the job scheduler has support for SLURM (Yoo et al., 2003), Lawrence Livermore National Laboratory's Flux scheduler (Ahn et al., 2018), and a rudimentary serialization model for debugging. Active learning is provided through PyTorch (Paszke et al., 2019) and Scikit-Learn (Pedregosa et al., 2011).
- This approach relies on SQL(ite) for the communication as it is a guaranteed atomic read and write that simplifies a lot of our efforts at the cost of performance.
- For coarse grain simulations, we currently couple with the Multi-BGK (Los Alamos National Laboratory, n.d.) as well as additional ICF codes. Preliminary studies have been made of coupling with more general Lattice Boltzmann simulations for other fields of study.
- For fine scale simulations, we use a mixture of LAMMPS (Plimpton, 1995) and proven analytic solutions; however, one could easily switch to another MD solution as long as they have similar capabilities.
- Since the effectiveness of machine learning methods varies depending on the available training data and problem, an additional check for any machine learning solution is added to the framework. In the GLUE Code, we query the models provided by the machine learning solution for the output associated with our inputs. The surrogate model then provides both the expected outputs and the uncertainty quantification to indicate its confidence that this specific model gave a valid answer. If the confidence is too low, the code falls back to calling the actual MD simulation and providing the data for the machine learning solution to retrain and generate a new surrogate model for future use.

Concurrent multiscale simulations

As depicted in Fig. 1, the parallelism of the Coarse Grain Solver is fully dependent on the solver of choice; however, the `GLUECode_Library` provides MPI interfaces to support distributed applications and all couplings thus far (barring hand-written tests for CI purposes) were with distributed coarse grain simulations. Initial efforts were made to make the `GLUECode_Library` calls non-blocking (similar to non-blocking communication in MPI), but this was given low priority as the coarse grain solvers considered thus far have comparatively fast time steps relative to the fine grain solvers and do not have a significant amount of work that can be overlapped with the cost of the GLUE Code. Similarly, there was work on providing a thread-safe interface for on-node parallelism (e.g. OpenMP), but this was similarly given low priority.

A single `GLUECode_Service` is primarily a series of sequential operations that occur in a persistent service and is built around monitoring a task queue. Said task queue allows for parallelism via the use of HPC Job Schedulers, but the actual computation/generation of ML models, as well as processing of requests and results, is sequential. That said, multiple `GLUECode_Service` instances can be run in parallel, and this has been done as a way to lessen congestion for particularly large simulation.

The Fine Grain Solver is once again dependent on the solver. We have worked with both GPU-enabled LAMMPS and MPI-enabled LAMMPS and the SlurmScheduler and FluxScheduler parts of our json schema (https://github.com/lanl/GLUE/blob/1.0/docs/inputSchema.json) are specifically set up to provide these configurations.

The overall GLUECode has a high degree of concurrency and resembles fork-join parallelism/MapReduce in practice, even if it consists of sequential tasks/stages.

Load imbalance

We have implemented traditional approaches for load balancing used by asynchronous task-based runtimes that rely on some form of a task queue and then various work-stealing algorithms.

However, due to the granularity of our tasks, we offload these tasks to HPC job schedulers like Slurm and Flux. By putting this work into a job queue, we can use proven tools to schedule those in an efficient manner.

To reduce the load that need to be balanced, we preprocess the requests prior to calling functions/subroutines like `bgk_req_batch_subr_f()`. Aspects of this can be found in the function `preprocess_icf()`. Future work will include better domain-aware scheduling and optimizations but, at this stage, we have minimized this so as to stress-test the overall GLUECode interface as well as evaluate the benefits of active learning-based models.

Multiple formats and parsing tools

We only provide the interfaces required for the coupling that we are working on, as one of the primary goals of this project was to provide a minimally invasive tool. Domain experts are the most knowledgeable in terms of what they need from each level of the simulation; therefore, we largely use co-design to define interfaces (e.g., https://github.com/lanl/GLUE/blob/1.0/GLUECode_Library/src/include/allInterface.h) as well as using control variables in the overall system configuration to utilize the appropriate tools (e.g., https://github.com/lanl/GLUE/blob/1.0/GLUECode_Service/processBGKResult.py) to generate and post-process results.

The existing tools will be generalized as necessary as new couplings are added. However, we aim to keep the interface relatively clean, allowing domain experts to take advantage of their knowledge and experience with tools like LAMMPS rather than rely on us to provide all the interfaces they need.

Code structure

The `GLUE` code is organized as follows:

- `GLUECode_Library` contains the C++ library that is meant to be linked to the coarse grain solver. This allows existing applications to couple to the `GLUECode_Service` with minimal code alterations.
- `GLUECode_Service` contains the Python scripts that the library communicates with and uses a combination of active learning and spawning of fine grain simulation jobs to enable and accelerate multiscale scientific applications.
- `docs` contains documentation for the Flux scheduler.
- `examples` contains sniff tests for the serial and the MPI versions of the code, and utility scripts, such as `pullICFData.py` that demonstrate how the `GLUE` Code framework can be used to access training data to perform deeper analysis.
- `jsonFiles` contains example input decks for the `GLUE` Code service.
- `lammpsFiles` contains LAMMPS example scripts (`in.Argon_Deuterium_masses` and `in.Argon_Deuterium_plasma`) that compute the mutual diffusion for a plasma composed of argon and deuterium at different concentrations and temperatures. More information about the test cases can be found within the files.
- `training` contains training data required for active learning.

Acknowledgements

The research presented in this article was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory (LANL) under project number 20190005DR and used resources provided by the LANL Institutional Computing Program.

We acknowledge the support of the U.S. Department of Energy through the LANL/LDRD Program for this work. AP and JS also acknowledge the Center for Non-Linear Studies at LANL. LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). Assigned LA-UR-22-29746.

Ahn, D. H., Bass, N., Chu, A., Garlick, J., Grondona, M., Herbein, S., Koning, J., Patki, T., Scogland, T. R., & Springmeyer, B. (2018). Workflows in support of large-scale science (WORKS'18). *Proceedings of the 2018 ACM/IEEE Workflows in Support of Large-Scale Science (WORKS'18)*, 10–19.

Cappello, F. (2009). Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, *23*(3), 212–226. https://doi.org/10.1177/1094342009106189

Diaw, A., Barros, K., Haack, J., Junghans, C., Keenan, B., Li, Y., Livescu, D., Lubbers, N., McKerns, M., Pavel, R., & others. (2020). Multiscale simulation of plasma flows using active learning. *Physical Review E*, *102*(2), 023310. https://doi.org/10.1103/physreve.102.023310

Glosli, J. N., Richards, D. F., Caspersen, K. J., Rudd, R. E., Gunnels, J. A., & Streitz, F. H. (2007). Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability. *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 1–11. https://doi.org/10.1145/1362622.1362700

Haack, J., Diaw, A., Pavel, R., Sagert, I., Keenan, B., Livescu, D., Lubbers, N., McKerns, M., Junghans, C., & Germann, T. (2021). Enabling predictive scale-bridging simulations through active learning. *APS Division of Plasma Physics Meeting Abstracts*, *2021*, ZO03–012.

Karra, S., Mehana, M., Lubbers, N., Chen, Y., Diaw, A., Santos, J. E., Pachalieva, A., Pavel, R. S., Haack, J. R., McKerns, M., & others. (2022). Predictive scale-bridging simulations through active learning. *arXiv Preprint arXiv:2209.09811*. https://doi.org/10.48550/ARXIV.2209.09811

Koo, R., & Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, *1*, 23–31.

Krekeler, C., Agarwal, A., Junghans, C., Praprotnik, M., & Delle Site, L. (2018). Adaptive resolution molecular dynamics technique: Down to the essential. *The Journal of Chemical Physics*, *149*(2), 024104. https://doi.org/10.1063/1.5031206

Liboff, R. L. (1959). Transport coefficients determined using the shielded Coulomb potential. *The Physics of Fluids*, *2*(1), 40–46. https://doi.org/10.1063/1.1724389

Los Alamos National Laboratory. (n.d.). LANL/Multi-BGK: Conservative multispecies kinetic equation solver. In *GitHub*. https://github.com/lanl/Multi-BGK

Lubbers, N., Agarwal, A., Chen, Y., Son, S., Mehana, M., Kang, Q., Karra, S., Junghans, C., Germann, T. C., & Viswanathan, H. S. (2020). Modeling and scale-bridging using machine learning: Nanoconfinement effects in porous media. *Scientific Reports*, *10*(1), 1–13. https://doi.org/10.1038/s41598-020-69661-0

Marinak, M., Haan, S., Dittrich, T., Tipton, R., & Zimmerman, G. (1998). A comparison of three-dimensional multimode hydrodynamic instability growth on various National Ignition Facility capsule designs with HYDRA simulations. *Physics of Plasmas*, *5*(4), 1125–1132. https://doi.org/10.1063/1.872643

McKerns, M. M., Strand, L., Sullivan, T., Fang, A., & Aivazis, M. A. (2011). Building a framework for predictive science. *Proceedings of the 10th Python in Science Conference, arXiv Preprint arXiv:1202.1056*. https://doi.org/10.48550/arXiv.1202.1056

Michael McKerns, P. H., & Aivazis, M. (2019). *Mystic: Highly-constrained non-convex optimization and UQ*. https://uqfoundation.github.io/project/mystic

Nagarajan, A., Junghans, C., & Matysiak, S. (2013). Multiscale simulation of liquid water using a four-to-one mapping for coarse-graining. *Journal of Chemical Theory and Computation*, *9*(11), 5168–5175. https://doi.org/10.1021/ct400566j

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). *PyTorch: An imperative style, high-performance deep learning library*. 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pavel, R. S., McPherson, A. L., Germann, T. C., & Junghans, C. (2015). Database assisted distribution to improve fault tolerance for multiphysics applications. *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, 1–8. https://doi.org/10.1145/2834899.2834908

Pavel, R., Junghans, C., Mniszewski, S. M., & Germann, T. C. (2017). *Using Charm++ to support multiscale multiphysics on the Trinity supercomputer*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., & others. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830.

Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, *117*(1), 1–19. https://doi.org/10.2172/10176421

Rinderknecht, H., Sio, H., Li, C., Zylstra, A., Rosenberg, M., Amendt, P., Delettrez, J., Bellei, C., Frenje, J., Johnson, M. G., & others. (2014). First observations of nonhydrodynamic mix at the fuel-shell interface in shock-driven inertial confinement implosions. *Physical Review Letters*, *112*(13), 135001. https://doi.org/10.1103/physrevlett.112.135001

Rosenberg, M., Séguin, F., Amendt, P., Atzeni, S., Rinderknecht, H., Hoffman, N., Zylstra, A., Li, C., Sio, H., Gatu Johnson, M., & others. (2015). Assessment of ion kinetic effects in shock-driven inertial confinement fusion implosions using fusion burn imaging. *Physics of Plasmas*, *22*(6), 062702. https://doi.org/10.1063/1.4921935

Ross, J., Higginson, D., Ryutov, D., Fiuza, F., Hatarik, R., Huntington, C., Kalantar, D., Link, A., Pollock, B., Remington, B., & others. (2017). Transition from collisional to collisionless regimes in interpenetrating plasma flows on the National Ignition Facility. *Physical Review Letters*, *118*(18), 185003. https://doi.org/10.1103/PhysRevLett.118.185003

Yoo, A. B., Jette, M. A., & Grondona, M. (2003). Slurm: Simple Linux utility for resource management. *Workshop on Job Scheduling Strategies for Parallel Processing*, 44–60. https://doi.org/10.1007/10968987_3