LLNL Response to the DOE ASCR RFI, "Stewardship of Software for Scientific and High-Performance Computing"*

J. Hittinger, G. Abdulla, D. Ahn, C. Follett, J. Foraker, S. Futral, T. Gamblin, M. Gamboa, M. Goldman, K. Halliday, J. Hill, T. Kolev, A. Kupresanin, I. Laguna, I. Lee, M. Legendre, E. Leon, T. Mendoza, M. Miller, K. Mohror, D. Quinlan, B. Springmeyer, V. Sochat

Introduction

For decades, Lawrence Livermore National Laboratory (LLNL) has been engaged in significant research, development, and support for software to enable scientific computing and, particularly, the use of high performance computing (HPC) in the NNSA mission space. In particular, the move in the mid-1990's to simulation as a leading component of stockpile stewardship through the ASCI and the successor ASC programs, as well as the need for reliable data acquisition and control software for the National Ignition Facility, have been important drivers in building expertise in production-quality software development at LLNL. LLNL has also been a leader in the DOE SciDAC FASTMath Institute and the DOE Exascale Computing Project (ECP), both of which have striven to make scientific computing software – in particular, the enabling technologies underpinning simulation capabilities – more widely adopted and sustainable. As such, we believe that our experience can inform the broader goal of software stewardship for scientific and high-performance computing.

LLNL strongly supports the formation of a new DOE ASCR program element in software stewardship and sustainment. Historically, DOE ASCR has funded applied mathematics and computer science research that has led to the development of important new capabilities and algorithms that are expressed as artifacts in research software. Such frameworks, libraries, and tools have seldom been directly funded to address the important issues of code maintenance, documentation, robustness, and community building. Software engineering and support have typically been done on the side in support of the ASCR-driven research products. DOE funding priorities have been slow to recognize that good software engineering, the kind that ensures research investments have more adoption and longevity, requires significant resources.

Based upon our experiences, we have prepared this response to highlight the concerns and issues we believe to be important as DOE ASCR considers its role in scientific software stewardship. We believe that role is important and will require a significant investment of new funding to legitimately support the technologies past and future DOE ASCR investments have and will produce to facilitate their uptake and adoption in the broader scientific computing community. Following a summary of our involvement in scientific software development, the remainder our response is organized around the nine topics specifically identified in the RFI.

Background

For any institution, developing an HPC code implies a significant investment, and basing such an investment on outside technology (libraries, frameworks, etc.) carries a substantial risk. At minimum, one wants confidence that the software libraries and tools on which a code is built will be

* Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear SecurityAdministration under Contract DE-AC52-07NA27344.

supported for the foreseeable future. Even better is that one will receive active support in how to use the software, to address bug fixes and porting issues, and to add desired new features. Scientific software that has broad community support is a low-risk choice, but it takes years, significant resources (funding and effort), and too often, the persistent effort of an evangelist for the software to build such community support. This model is not efficient, predictable, reliable, or sustainable.

The SciDAC program was successful in bringing ASCR technologies to the other Office of Science Programs, particularly in providing resources to help science application codes make better use for HPC resources. However, while there was some support for software sustainability, it was limited. The ECP has successfully placed an emphasis on better software development practices, fostering and promoting software productivity and sustainability for example, through the IDEAS-ECP. The DOE laboratories engaged in ECP have demonstrated the ability to execute a more principled software engineering approach to produce production-quality software. This effort has paid dividends – already, under ECP, there is interest from industry players (engineering firms and cloud providers) in using the ECP software stack – and broad usage could easily turn into contributions from these companies. When the ECP ends, however, there is no ASCR program element currently positioned—and funded at the necessary level—to maintain and, ideally, build upon, this ECP success. Software engineering and development is an activity distinct from, but complementary to, research, computational partnerships, and the DOE ASCR computing facilities.

The stockpile stewardship programs have provided decades of experience at the NNSA labs in the development, use, and sustainment of simulation codes on which the mission relies. Through the NNSA ASCI and ASC programs, we learned how to assemble multidisciplinary teams to create the reliable, supported technologies for the entire HPC software stack, including system software, programming models, programming environments, math and data management libraries, application codes, and data analysis and workflow tools. In LLNL's Computing Directorate, we have an entire division of computer scientists, the Applications, Simulations & Quality (ASQ) Division, dedicated to support the weapons programs, of which ASC is a major component; this includes nearly 150 employees engaged in production-quality software development, 30 contributing to software development environment support, and another 20 providing software quality assurance expertise (SQA). Our National Ignition Facility Computing (NIFC) Division similarly provides the computer science expertise to support operationally critical system control and the massive data acquisition and management, with over 60 employee developing software, four supporting the software development environment, and another six addressing SQA. Livermore Computing (LC), which is funded primarily by ASC and institutional investments, has not only deployed world-leading HPC resources, but it has played a significant role in the deployment of high-end Linux-based clusters and, importantly, the production-quality system software (e.g., OpenZFS for Linux [1], SLURM [2], Flux [3], Spack [4]) needed to enable high productivity of such systems in a shared computing center environment. Through LC, LLNL has a long history of working in public/private partnerships with software and hardware vendors (e.g., IBM, HPE, NVIDIA, AMD, RogueWave) to produce the systems and software needed to make these systems useful for our mission. Finally, while LLNL's Center for Applied Scientific Computing (CASC) is the focal point of math and computer science *research* at the lab, many research projects that emerged in CASC have also grown software development teams funded by ASC and other program elements. Spack, MFEM [5], *hypre [6]*, SUNDIALS [7], and ROSE [8] all emerged as cutting-edge research projects within CASC, but as these projects grew, they were able to hire software developers into CASC and other divisions to sustain their projects while continuing basic research efforts. Close connections to supportive programs, along with institutional support

and recognition for strong software development, have enabled projects at LLNL to bridge the so-called "Valley of Death".

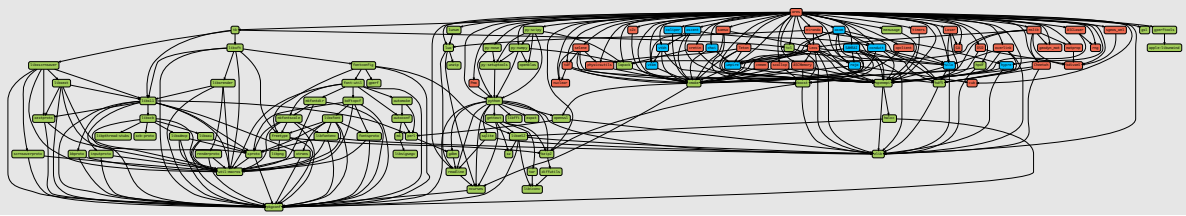

Figure 1: LLNL multi-physics code showing LLNL-developed internal (red) packages, LLNL-developed open-source packages (blue), and external open-source packages (green).

Software dependencies and requirements

LLNL's scientific application codes make extensive use of software packages and libraries from both within the lab and from the greater community; we have a strong culture of both creating and making use of open-source scientific software. As such, we have a great deal of experience in managing dependencies, modular software development and design, and maintaining software stacks on our HPC clusters from system software up to the applications codes.

Dependencies

There is insufficient space in this response to list all the dependencies, APIs, and standards upon which LLNL application codes rely. If we think about the absolute minimum requirements, then we might say that we can make do with a C/C++ compiler (preferably supporting C++11 or later), an MPI implementation; math libraries like FFTW, BLAS, and LAPACK; GPU programming models like CUDA and HIP; and suitable debuggers and profilers. That would enable us to develop minimal versions of software for exascale machines.

However, LLNL's mission is much larger than the libraries and languages required to build a single high-performance application code. Stockpile stewardship and other missions at the lab require that we *sustain* a wide range of multi-physics code projects over time, in many cases for decades. These codes are written in more than simply C and C++; they make use of C, C++, Fortran, Python, Lua, CUDA, HIP, and sometimes other languages. Libraries evolve along with the codes they serve, and codes adopt features from libraries as they emerge. Figure 1 shows the component packages in ARES, an exemplar of LLNL's multi-physics codes. ARES relies on 114 different component libraries to function, a number that has grown significantly over time. Over 60% (72) of its dependencies are external (potentially) open-source libraries, including an MPI implementation, Python packages, boost, build tools such as CMake, and I/O libraries such as HDF5. Just under 40% (42) of the dependencies are LLNL-developed, and of these, 12 are open source. We can see from the ARES example that LLNL software is a complex ecosystem. Even proprietary codes rely on hundreds of open-source components, many developed outside DOE or at labs other than from that of the application code itself. Sustaining mission codes requires us to sustain this *entire* ecosystem of dependencies, which is a challenging maintenance and porting issue in and of itself. Tools to simplify the installation of software packages, such as Spack [4] and BLT [9], are important to manage these complex ecosystems.

Modular Code Design

In 2015, an internal study [10] was undertaken to solidify LLNL's next-generation code strategy, and it identified modularity as fundamental to our approach to simulation code development. The need for modularity stemmed from many factors, but the key aspects were flexible physics, sustainability, and the then-impending transition to GPU computing. Changing mission needs demanded flexible physics: the ability to leverage one model across many problems, and the ability to use similar methods on the same problem to increase certainty were deemed critical. The study also found that around 40% of the code in our applications was common CS infrastructure that could be consolidated to reduce each team's maintenance burden. The study pushed for code teams to maintain a common infrastructure stack, especially since all codes would need significant new infrastructure to facilitate the transition to GPUs.

This effort led to the development of performance portability libraries like RAJA, CHAI, and Umpire, which allow codes to effectively use multiple modes of parallelism on the CPU and GPU. Math libraries such as MFEM were reworked to exploit GPUs and other hardware with specialized back-ends, and the development of Axom and other middleware layers provided code teams with common geometry, logging, and meshing models for integrated codes. Physics packages have been reworked to use these frameworks instead of coding directly to specific hardware. The modular code strategy was revisited in 2019, and despite some challenges with integration, the review still determined it was the right approach—that integration costs are a price worth paying for modularity. Without this common base, our codes could not have easily adopted GPU portability frameworks. Moreover, the transition to GPUs would have been a new effort for each team—with it, teams could leverage the performance tuning work of others. Reworking of LLNL codes to leverage this common base has separated the high-level concerns of applications from the low-level concerns of mapping loops to specific GPU hardware. As we port our stack from NVIDIA GPUs to AMD GPUs in preparation for the El Capitan machine, we can (mostly) port the infrastructure, not the applications, to leverage the new hardware. We expect this approach will continue to pay dividends beyond the current set of architectures well into the future.

One feature of LLNL's modular approach is an emphasis of interoperability over dependency. Most LLNL software components are light weight, in the sense that they provide functionality while being as stand-alone as possible, i.e., not requiring the inclusion a host of additional packages that may be of little interest to the application. Thus, one can pick and choose the software packages that provide only the needed functionality. The use of multiple packages is then facilitated by interoperability between the packages.

Software Stacks

LLNL code teams are not alone in shifting towards modularity in recent years. The complexity of the modern HPC software environment, catalyzed by the transition to GPUs, has driven the creation of many different "software stacks" – collections of software built and tested together, aimed at creating a solid foundation on which to build scientific codes. Within ECP, the E4S [11] stack has emerged to fulfill one of the main charges in ECP's mission statement: *to create an "integrated software stack*". Similarly, and partly inspired by E4S, LLNL has initiated its own software stack effort, RADIUSS [12]. The goal of RADIUSS is to create an integrated, welltested set of core libraries that codes across LLNL and outside of LLNL can easily use. The stack contains portability libraries, build tools such as Spack and BLT, LLNL's next-generation Flux resource manager, visualization tools such as VisIt, and math libraries such as MFEM. Similar efforts, for example, EESSI [13], the Compute Canada software stack

[14], and a stack for the European Joint Undertaking (JU) for exascale are also emerging. In all cases, the goal is to reduce the total maintenance effort for teams by consolidating build, integration, and testing efforts.

Software stacks broaden the reach of software by bridging the gap between upstream developers and the software's ultimate consumers. Stacks provide support for exotic platforms (such as exascale machines and other one-of-a-kind DOE environments) where the upstream developers either have not or cannot test their code. They ensure that the critical DOE projects continue to work together and, perhaps more importantly, they ensure that the many *dependencies* of those projects also continue to work. E4S comprises 91 DOE-developed packages, but when dependencies of those packages are also included, the number increases to over 550 packages. These dependencies are analogous to the green nodes in the ARES diagram in Figure 1; there is a mountain of open-source software upon which we rely to keep our software functional.

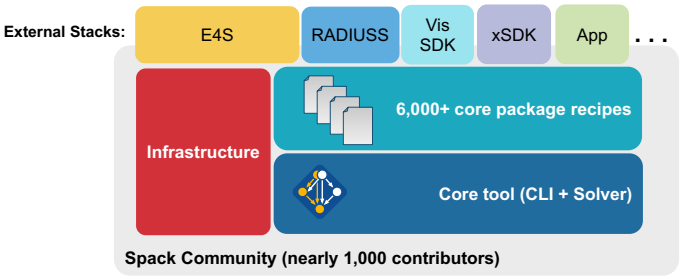

Figure 2: Structure of the Spack software ecosystem.

Package Management and Software Ecosystems

The emergence of software stacks has been driven by need and has been enabled by capable package managers for HPC software. Software stacks such as E4S and RADIUSS can manage hundreds of components because they leverage the efforts of a much larger open-source community. Figure 2 shows the structure of the *Spack* software ecosystem. Spack consists of a core package management tool, roughly 6,000 software recipes, and build infrastructure. All of this is maintained by a large community of contributors—nearly 1,000 people worldwide have contributed to Spack's package descriptions (Figure 3 shows the volume and diversity of contributions over time), and a core team of 30-40 maintainers from around the world review and manage these contributions. The team ensures that packages continue to work with a large continuous integration system—Spack's infrastructure currently runs around 100,000 builds per month, both in the cloud and on local hardware resources, to ensure that the stacks continue to work.

Stacks such as E4S, RADIUSS, and others sit on top of the 6,000 core package descriptions in Spack; they are curated sub-sets of the larger Spack stack. Other stacks, such as ECP SDK teams as well as application-specific software stacks are maintained in Spack. This firm foundation of maintainers and automation keeps these stacks working together; they critically rely on Spack and its broader open-source ecosystem.

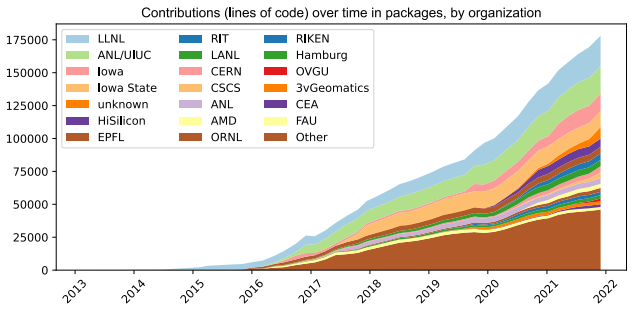

Figure 3: Contributions to packages in Spack.

With many software stack efforts emerging, there is an opportunity to reduce duplication further. LLNL relies on ~42 of the 91 packages in E4S (~340 if we include dependencies). Of these, 21 are LLNL-developed. Other labs are similar – they both develop and use a significant fraction of the same

software stack. LLNL's RADIUSS stack also has significant overlap with E4S. The critical need, at this point, is that these stacks remain reliable. The component packages will change over time as new technical challenges (like GPUs and other accelerators) emerge, but the users should be able to rely on a living, evolving, tested stack to ensure that their software continues to work. A dedicated sustainability program would save tremendous amounts of duplicated integration effort, and a consolidated stack would have broad reach outside DOE—if it becomes stable and, therefore, reliable enough for adoption by other government agencies, industry, and academia.

Key gaps and future capabilities

LLNL is starting a new initiative to develop a *Virtual Design Assistant*, or ViDA, which will automate many currently manual processes used by analysts and designers at the lab (the users of our application codes). ViDA will require workflow tools (like LLNL's Flux scheduler), and it will require LLNL to integrate its multi-physics codes with AI frameworks—both mainstream tools (PyTorch, TensorFlow) and in-house AI frameworks (LBANN [15]). We expect that we will need to ensure that mainstream AI frameworks can build and run at scale on HPC resources, and we expect a new class of workflows will emerge for training and managing ML models. Furthermore, independent of the tools used, HPC applications are increasingly evolving into loosely coupled workflows rather than the monolithic applications of traditional codes. This evolution is due to changes in workload, i.e., a focus on ensemble calculations and the continuing trend towards multiscale integration, but also as a response to heterogenous hardware and the integration of cloud tools and infrastructure. However, developing, maintaining, and optimizing the underlying workflow infrastructure (Flux [3], Maestro [16], Merlin [17], Themis, PARSL [18]) has proven challenging and has led to significant duplications in efforts.

Currently, support for ML frameworks is limited in our software stacks, and keeping mainstream ML tools working in our environment can be difficult, as they are designed for a limited set of cloud environments. At a higher level, orchestration frameworks, like Kubernetes, do not yet map well to an HPC environment – they can be used to run large TensorFlow jobs, but our fast HPC networks do not support Smart NICs, Software Defined Networking (SDN), or isolation features needed to run Kubernetes [19] effectively.

There is a critical need to make it easier to run AI and cloud services in the HPC center, alongside traditional application codes. Most industrial work in AI is done in the cloud, using cloud frameworks, and there are billions of dollars invested each year in improving industry AI and infrastructure tools. HPC and cloud have evolved largely independently of each other, but with the emergence of AI, industry workloads are starting to look more like HPC. The question for DOE is whether we will leverage the investments made by industry or whether we will redevelop many of these technologies ourselves. Adopt and integrate are the clear answers, as it is unlikely that we can sustain both an internal HPC stack and an internal ML for stack AI frameworks; technologies are simply evolving too quickly. However, if funding in DOE emphasizes novelty and new research over development and adoption, we may find that we do not have money that can pay for the cheaper and more fruitful path.

Risks and Mitigation Strategies

The benefits of relying on large software stacks are clear: we share a common software base with other DOE labs and with the rest of the open-source scientific software community, which has reached a critical mass under ECP that enables the inclusion large volumes of external contributions. We also have an acute need to leverage key technologies from industry, rather than reinventing them

ourselves. The benefits of leveraging a common stack are clear from Figure 3; DOE contributions account for only a small portion of the total *packages* in the Spack ecosystem, though core developers still spend considerable effort *integrating* outside work. Ongoing development efforts on this and other infrastructure projects provide significant spin-in benefit to DOE, and they allow us to leverage others' work.

If we are unable to converge on a common stack, or if we are unable to continue the development efforts started under ECP that enable these benefits, our mitigation would likely be to focus on ASC mission-relevant packages, at the expense of other packages that are key to LLNL and other labs. It is the only way we can continue to meet mission needs in the absence of dedicated development and sustainability funding. While this may seem like a pragmatic and frugal choice, it ignores the network effects and economies of scale inherent in community development efforts. Software sustainment efforts pay greater dividends over time, with scale, and with greater community buy-in. Cutting them to bare-minimum funding levels would have far greater consequences than the reductions to DOE-funded efforts—we would lose the communities, external contributions, and collaborations that we have been able to build under ECP and ASC ATDM.

Key takeaway: *Developing modular software and adopting 3rd party libraries where possible allows for rapid development but comes with risk and challenges. LLNL mitigates these through judicious use of tools (e.g., Spack) and developing a culture that rewards reuse. Changes to the way we use computing – the increase in the use of AI and cloud services as parts of larger, federated workflows – will require new infrastructure and tools to enable the future DOE computing needs.*

Security and integrity of software and data

From our experience, there are several key aspects to consider for the security and integrity of software and data: security of shared compute environments, protection of source code and binaries, protection of shared data sets for collaboration, and security of sensitive data.

Security of Shared Compute Environments

It is common for HPC scientific software developers and users to utilize shared, multi-tenant compute resources, e.g., gateway or login nodes where multiple users log in concurrently, request resources, and analyze results. Additionally, the increased use of AI/ML for science is driving the need for new multi-tenant compute resources that allow for user-defined software stacks and development environments to rapidly prototype new solutions and to produce more reproducible and portable solutions. While we have well-known and practiced security measures in the traditional HPC environment, these new AI/ML needs often lead to the use of technologies, such as containers, that may contain software that does not adhere to the predefined and approved software stacks. It is imperative to monitor the activity on these shared resources to ensure that users do not execute tools or programs that may result in exposing sensitive user data to others on the shared resource.

While Livermore Computing provides a large suite of tools for our users to perform their work, there are an ever-growing number of publicly available software tools that can assist developers and users in their daily tasks, ranging from performance tools to data analysis tools. Users of shared, multi-tenant systems commonly request new software tools to aid them in their work and, sometimes, even download and install them in user space without permission from the center. Although these software tools can be useful and users are simply trying to get their work done efficiently, such installations can pose a risk in shared environments due to their inner workings. For example, some

tools may open unencrypted ports, other tools may require sensitive credentials that are then leaked via the global process table, and yet other tools may simply assign an overly permissive access control list, all of which can expose sensitive information about the current user as well as other concurrent users of the shared system. At LLNL, we employ multiple safeguards to mitigate the risk of software tools, including performing automated scans and monitoring of user software to provide a solid baseline view into the system, curating known-good software and configurations with which users can work, and our last line of defense, educating users about potential risks. Together, these practices help us mitigate the risks of ever-changing software. Moving forward, new scanning, interrogation, deployment, and monitoring approaches need to be investigated to ensure nefarious software has not been deployed through container technologies while ensuring that users are able to fully leverage the power containers offer and while treating them like traditional predefined software stacks and processes.

Protection of source code and binaries

In addition to monitoring the runtime activities of scientific programs, it is also imperative to ensure that the source code and binaries used do not contain vulnerabilities that can allow them to be attacked or their integrity to be undermined. Vulnerability detection tools can be used to identify and even remove known vulnerabilities within a repository, however additional tools are needed to ensure that third-party software conforms to institutional security and data protection policies and to identify and remove zero-day vulnerabilities.

Tools that identify zero-day risks and potential vulnerabilities in source code and binaries reduce the attack surface of HPC software. Analysis to detect possible malicious behaviors in software reduces the risk that users may inadvertently execute infected software. These tools follow the zero-trust methodology by assuming all parts of software and their dependencies (i.e., supply-chain) may be infected and must be screened for both vulnerabilities and malicious behaviors before being added to the software repository. Furthermore, periodic scanning of software within the repository for undocumented changes is another best practice.

Moving forward, with the growing use of package management systems for reproducible software builds, we believe that deep integration of cyber security analysis tools within these systems can be used to enforce security and data protection policies. Tools available for this screening include commercial-grade source code static analysis for C, C++, and Fortran. Additional prototype tools have been implemented for other languages commonly used in scientific and ML software, such as Python. Additional work is needed in program analysis tool development. We need efforts to match the analyses that these tools can perform with system, site, and organization-specific policies. It is expected that policies will vary regarding data protection, acceptable third-party dependencies, and assurance levels. Separate from cyber security analysis results being folded into package management systems, the analysis for both vulnerabilities and malicious behaviors at scale can be posed as an HPC problem, one which could favorably impact broad areas of the government.

Protection of Shared Data Sets

In addition to protection of programs used on HPC systems, we also need to protect the integrity of scientific data used in collaborative research among national and international teams. To facilitate data sharing and collaboration, federated data systems or cloud architectures are being used the scientific community. Although scientific data is considered less vulnerable to malicious attacks than other kinds of data, there are cases where the data could have been maliciously manipulated or where the hosted servers are compromised for various reasons.

As an example, the Earth System Grid Federation (ESGF) [20] successfully operated a globally distributed data infrastructure for over 15 years [21] and did not see a security accident until June of 2015 (the same attack affected Equifax [22]). The attack on the ESGF servers did not affect the data, but the attackers used the NASA/NCCS, LIU (Sweden), and IPSL (France) servers as resources for crypto mining, as software specifically for that purpose was discovered at the IPSL site. Data was not compromised at the LLNL site due several measures already in place: access to the Struts-based webapp was restricted and ESGF data disks were mounted read-only. The affected sites were fortunate to not have their data corrupted, as data manipulation was not in the interest of this attacker.

Lessons learned from this example (and others experienced by LLNL staff) include:

- A key strength has been the implementation of a system for federated search and replication of datasets amongst participating sites with user login and read only access to the data.
- Data servers must have proper security scanning and timely patching for vulnerabilities. Project teams should invest in long-term planning for data security.
- Legacy data access interfaces should be replaced with new interfaces with a modern authentication [23].
- Data may need to be sanitized to prevent sharing of sensitive data, e.g., telescope images may inadvertently reveal satellite positions. However, care must be taken in sanitizing data sets, because the sanitization may lead to loss of information that makes analysis impossible. Additionally, care must be taken to ensure the use of aggregated sanitized data sets does not lead to a mosaic effect that can reveal sensitive information [24].
- Security tools should not hinder collaboration and should not violate FAIR data principals.

Security of Sensitive Data

In the healthcare and biomedical data science domain, data security is directly linked to individual privacy. Multiple approaches are employed to protect the privacy of individuals. Whenever possible, using de-identified data is preferred to using identifiable information. De-identified data has had identifiable features removed, including name, social security number, medical record number, and other such information that could link records back to an individual. The use of synthetic data can aid with algorithm and tool development, but generating synthetic data with realistic properties can be challenging.

There are complex regulatory requirements for different types of sensitive data. Healthcare data can be subject to the Health Insurance Portability and Affordability Act (HIPAA). Other countries and some U.S. states have additional privacy regulations. There are many other types of Controlled Unclassified Information (CUI) [25] that generally require need-to-know protection. Best practices for protecting sensitive datasets involve the use of both administrative controls and technical controls. Administrative controls include training, user agreements, and other documentation. Data Use Agreements (DUAs) are documents that describe the permissible use of a dataset are often used. Administrative controls are contractual in nature and do not by themselves prevent unauthorized access or accidental misuse. Technical controls provide these security mechanisms through software. A properly configured system suitable for processing sensitive data may implement hundreds of technical controls. Many of the technical controls are applicable across the different types of sensitive datasets. Federal information systems follow the Federal Information Systems Modernization Act (FISMA) [26], while non-Federal systems follow the NIST 800-171 guidelines for protecting CUI [27].

LLNL has developed HPCrypt, software for protecting sensitive datasets on High Performance Computing (HPC) systems. This software is actively being beta tested on HPC clusters at LLNL and Argonne National Laboratory. Additional research in related approaches including homomorphic encryption, differential privacy, and federated learning would provide great value for data protection. While security controls such as encryption are becoming ubiquitous, edge cases such as HIPAA require additional compliance requirements that can be challenging to understand and implement. Simplifying and standardizing sensitive data protection regulations would be broadly beneficial.

Key Takeaways: *Developers must treat security of their code and data with an importance akin to functionality and performance for software to be used and adopted broadly. Tools for scanning and detection of threats embedded in the developer workflow (e.g., continuous integration or package management) have proven to be an effective first step toward continuous monitoring of threats. More effort is needed in practices for ensuring the security of multi-tenant systems, in the development of program analysis tools to detect program vulnerabilities, and in standardizing data protection regulations and processes.*

Infrastructure requirements for scientific software development

Both HPC hardware and software environments are growing more heterogenous and complex, which is driving new challenges in building and maintaining HPC software. On the computing environments side, we face new paradigms such as HPC-integrated machine learning, cloud computing, and increasingly complex HPC applications. On the hardware side, the HPC market currently presents choices between four major CPU vendors and three major GPU vendors, which is in addition to various non-traditional accelerators such as neuromorphic systems, quantum, and FPGAs.

The uncertainty and complexity have significant implications for HPC software development and use. We focus on resource management, traditional HPC tools (debuggers and performance tools), developer tools (source management, continuous integration tools, etc.), and system software. The software challenges in these areas have a common theme of struggling to handle growing heterogeneity and new computing environments, though the details in each vary.

Development Tools and Best Practices

Multiple revolutions have occurred in the last decade in software engineering for handling large, distributed software projects. Resources such as git [28], GitHub [29], GitLab [30], continuous integration (CI) frameworks, and wiki documentation make it easy for distributed software developers to coordinate and build software. CI tools make testing, issue tracking, and management of external contributions tractable for large projects. Containers allow us to package, distribute, and reproduce complex application workflows. Silicon-Valley-style data analytics infrastructure, such as Jupyter [31] or Pandas [32], can be applied to large HPC data sets for new insights.

These development tools are all freely available in the cloud. GitHub, for example, provides free repository hosting, project management, and compute cycles for developing and testing open-source projects. Modern software developers have thus come to expect that these types of tools are available for their projects, and transitioning to a closed environment, like most HPC centers, can be shocking—the tools available do not match what is available for free for open projects. Moreover, because our software relies on hundreds of open-source packages, keeping HPC projects up to date

with dependencies can be particularly difficult. There is an acute need for better and more secure integration of internal and external developer tools.

The primary challenge of adapting these tools to HPC centers around security. Cloud CI services run in sandboxed environments, where code changes can be run and tested from completely untrusted contributors to a project. Cloud CI services typically run very basic Linux (and sometimes Windows and macOS) environments, but they do *not* provide hardware resources to test the latest HPC architectures. Ideally, HPC centers could fill this gap, but nearly all HPC centers have stringent security requirements that prevent them from running CI for cloud projects. Ironically, at least half of LLNL's software is open source, and proprietary internal products rely heavily on these and other open-source dependencies. We cannot continuously test our own software on the HPC platforms of greatest concern for us. To adequately sustain a DOE software ecosystem, DOE needs to provide not only internal compute cycles for production runs, but also compute resources for testing and sustaining its own open-source software ecosystem. Without such resources, projects will founder, as they will not be able to ensure that they continue working on HPC machines. Security technologies discussed in the previous section will be critical for enabling CI on HPC, as we will need to scan and have confidence in code from untrusted sources.

While HPC resources are the main pain point for cloud-developed codes, we still need developer tools and resources for our own internal applications. Services such as GitLab need to be adapted to work in multi-tenant HPC environments—they often are written assuming execution in single-user or already-secure environments or assuming that the user has root access. Containers require HPC-focused effort to work in rootless environments [33, 34]. Jupyter and CI tools need security adaptions to work in the common HPC multi-user environments. Unfortunately, these adaptions of development tools can sometimes be site-specific. Work by LLNL to deploy a web application into its security model does not necessarily help LANL do the same. The HPC community and laboratories lack common infrastructure or standards that would make deploying these tools easy, which leads to patchwork deployments across computing centers.

In addition to the technical challenges, there are engagement challenges in getting developers to adopt the best practices that accompany new development tools. It is easy (and perhaps even common) to dismiss errors that creep into CI systems as just testing idiosyncrasies. A team can adopt git without adopting its powerful branching and tracking capabilities. Developer tool deployments need to come with training and engagement with users. LLNL has had success with dedicated application-engagement tiger teams that reached out to application developers and to help them learn and adopt tools and best-practices (along with other challenges, such as porting and tuning).

Resource Management and Workflows

Both application complexity and system heterogeneity are driving new challenges in resource management and workflow tools. Many HPC applications are moving away from single-application workflows to incorporating AI/ML, in-situ visualization, non-traditional datastores, and other new software technologies. These software technologies can also rely on new hardware, such as neuromorphic systems and accelerators. Composing, optimizing, and scheduling these complicated software stacks across systems is a challenge.

Numerous projects have built complex workflow tools to handle these challenges, which each have their own strengths and specific goals. There are ongoing efforts to break workflow tools into best-of-field components [35], which can then be reassembled into workflow technologies for specific

problems. Still, the best practices for composing and optimizing these composite workflows are still unresolved. How should users build and optimize these combined workflows?

The cloud community is building powerful software that can be useful for HPC, but managing cloud-style resources in HPC remains a challenge. For example, the Ray infrastructure (a powerful framework that can do hyperparameter tuning) requires Kubernetes and allocates resources with an elastic and descriptive model. Ray could be useful for HPC, but it is an open question how HPC resource managers could run it with their traditional descriptive and fixed allocation models.

Traditional Tools

The traditional HPC tool software (debuggers, performance analysis, correctness tools) ecosystem faces different challenges than development tools, which are discussed in the next subsection. These traditional tools can be thought of as helping the user understand how an application runs and how it interacts with hardware, e.g., why do some code paths lead to bugs or why do some loops run slow on large inputs, etc. As such, these tools need a deep understanding of both applications and hardware to tie them together. Unfortunately, heterogeneous hardware and increasingly complex applications make that a challenge.

The traditional HPC tool challenge was making tools scale to very large numbers of nodes. That remains important, but both debuggers and performance tools were demonstrating full system runs on ORNL's Jaguar fifteen years ago [36]. Instead of tools monitoring 10,000 nodes, they now need to monitor 10,000 threads on a node. Some tools [37] have started to address this scaling twist with redesigned data structures and in-tool parallelism, but the problem remains generally open and has a significant impact on tool performance across the tools community.

Traditional tools need a deep understanding of the underlying hardware, and the diverse hardware market makes this a significant challenge. For each major hardware platform and each tool, someone could invest in software engineering to port the tool to the hardware, but in our experience, these are large multi person-year efforts for each tool and architecture combination. This is not a scalable solution. Application developers have tackled this problem by building standards to separate themselves from the hardware [38, 39]. Tool developers have several programming model standards [40, 41], but would benefit from hardware-level standardization efforts.

Application complexity is another challenge for tools. A common tool operation involves attributing some collected information to a point in an application (e.g., a performance sample to a line of code or a debug value to a variable). To do so, the tool must understand application-level structures. However, HPC applications are growing larger, using new programming models, integrating with scripting languages, and incorporating new computing paradigms such as machine learning. These changes make it more challenging to understand an application code. In some cases, programming models (OpenMP, MPI, Kokkos [42]) have added tool interfaces. Python has added both performance profiling and debugging interfaces into the interpreter. Nevertheless, most new software technologies take years to incorporate tool support, and maintaining tool support for too many technologies is not viable. Some tools [42, 43] have had success asking application developers to manually encode attribution information into a program, but that is a significant hurdle to tool adoption and not ideal. HPC tools have also not significantly broached into new computing paradigms such as machine learning or cloud computing. It is neither clear how tools should help someone debug a large machine-learning/simulation integrated workflow nor how to best performance analyze a cloud-offloaded task.

System Software

The system software that runs our supercomputers is aging. Linux is thirty years old, and GCC and glibc are even older. Even as the software ages, it is growing more complex to support new hardware and workflows. We need to identify ways to both maintain and to update our system software stack for modern complex HPC systems.

Within operating systems, managing HPC driver dependencies has become a significant challenge. Traditional operating system design involved isolating drivers for safety and security, but modern HPC operating systems require integrated drivers. To optimize networks, GPUs, NVME devices, accelerators, and other devices vendors have started to interlink drivers. A GPU, for example, may send data directly to a network card. When a single vendor provides all drivers, this is a tractable problem, but managing the driver integration concerns from multiple vendors remains an ongoing issue. Even if one finds a set of consistent and compatible driver versions, a security update may topple the fragile stack.

HPC system software is also complicated by not controlling its own destiny. Most system software comes from outside the HPC community. Companies like Amazon, Google, and RedHat are leading the open-source system software movements, and they do not always have HPC needs at the forefront of their efforts. This situation leads to incompatibilities, such as containers struggling to use HPC file systems and network interfaces efficiently. We can maintain patches on open-source system software, but are there better approaches and designs for adapting modern system software techniques to HPC?

Key takeaway: *Scientific software developers demand and deserve tools on par or exceeding what opensource development in the cloud can offer for free. But these tools must be customized for our HPC environments that DOE invests so heavily in. DOE can leverage a lot of great work in the broader cloud software tool ecosystem, but those will not solve all of our problems in HPC without our participation*

Developing and maintaining community software

At LLNL, we have over 40 years of experience developing production-quality HPC software as part of the ASCI and ASC programs and their predecessors. Many of the software packages developed as part of these efforts have existed for multiple decades and have grown to be used by the wider community, outside of their original internal applications. Examples of such packages include: OpenZFS, *hypre* , SUNDIALS, SAMRAI [44], ROSE, VisIt [45], and MFEM. Several newer LLNL-developed packages like Spack, RAJA [38], ZFP [46], HiOp [47], Conduit [48], Flux, and SCR [49] are also on the same trajectory. At LLNL, we care deeply about software and have one of the longest track records in software development and stewardship in the DOE.

What we have learned from our history is that, since many HPC applications in the DOE and the broader scientific community live for decades, they are reluctant to take on external dependencies that may not be around long-term. The decision to adopt a particular scientific software package in an HPC application is often determined by both the technical capabilities of the software package as well as by the expected level of support and the longevity of that package. *Therefore, to ensure that our software can impact the wider community it needs to "live long and prosper", i.e., it needs to continuously improve, grow, and be used in applications that are actively supported. Further, users need to have confidence that the software package will have support beyond the current funding cycle.*

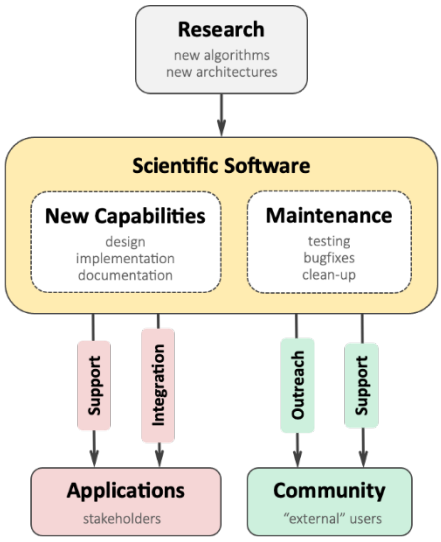

Figure 4: Scientific software activities

Figure 4 shows a simplified diagram of some of the major activities in a healthy scientific software project. To avoid stagnation, the software should continuously grow its **capabilities,** which often (e.g., for packages such as *hypre* and MFEM) requires **research** on new algorithms, e.g., to improve quality/robustness or to target new architectures. This evolution cannot be accomplished without maintaining strong connections between software development and support and the algorithmic expertise on the team. For example, post-ECP, it is critical that we maintain strong collaborations between GPU algorithmic expertise and the software teams so we can support current and future applications that want to use GPU capabilities. At LLNL, we have had found integrated teams of researchers and developers to be highly effective.

Without user adoption, software dies, and so it is also critical to have funds for both initial **integration** in new applications and for continued long-term **support** of current applications. For example, it is important to provide resources for the current ECP scientific software technologies so they can continue to support the ECP science applications after the end of the project.

Any project needs to grow its wider **community** with **outreach**, via workshops, tutorials, publications, etc., as well as with short-term **community support** that does not yet have application funding attached to it. Such growth is critical for recruitment, future funding, and longevity and requires dedicated time from team members, that is not currently funded directly.

Finally, software **maintenance** – the testing, documentation, porting, distribution, and installation, to ensure the software package continues to function and is ready for use – is rarely directly funded. Without dedicated support for maintenance, code quality is likely to deteriorate, particularly as the project grows. Documentation, for example, takes a lot of time and can be critical for user adoption. There can be benefits if a lot of this maintenance is performed by the software team, but some of it could be automated and centralized in the DOE. We emphasize that while maintenance is important, it is not enough: *continued application support and growth are critical for a healthy scientific software project.*

To summarize, each scientific software package requires additional effort, beyond what is needed to develop and maintain it internally, in the following four areas:

1. **Application Support** – Both initial integration for new applications and ongoing support for current customers. *Requires 0.5-1.0 FTEs per application, currently funded by application*

customers. To maximize impact, it is important that the apps also have support to engage with software technology teams.

2. Development of **New Capabilities** – Motivated by the needs of both future applications as well as new upcoming architectures. *Requires 1.0-1.5 FTEs, currently funded by application customers.*

3. **Community Outreach** – Grow the project community, foster collaborations, influence standards, and attract users to become developers and/or future customers. *Requires 0.5-1.0 FTEs, currently funded by projects like ECP or internal funding, implicitly funded by application customers.*

4. Software **Maintenance** – Ensure code quality and ease of use. *Needs 0.5-1.0 FTEs, currently funded by projects like ECP or internally funded, implicitly funded by application customers.*

All of these tasks are important, but the first three are critical. The FTE amounts above are approximate and based on an informal survey at LLNL. The exact amounts will depend on the software project complexity and the complexity of its userbase.

As a specific example, here is how these tasks look for the post-ECP needs of one of the LLNL-developed ECP scientific software libraries:

- **Retain the GPU algorithmic expertise on the team so we can support applications that are already using the code – 1.0 FTEs (ongoing).** *The shift to GPUs was a huge change, and we will feel the aftershocks for a long time. These include answering questions/consulting, bugfixes and extensions, porting and tuning of additional code components, developing new GPU-specific capabilities, documentation, and other GPU-related tasks. The ECP allowed us to hire experts in this area. Without post-ECP funding for GPU support, we may not be able to retain this GPU expertise.*
- **Extend our algorithms to impact non-ECP applications that want to use the exascale systems and/or transition to high-order algorithms – 1.0 FTEs (ongoing).** *Several applications in SciDAC, other parts of the DOE, and industry are just beginning to explore GPUs and can very much benefit of our ECP work, if we add GPU support for additional physics and the missing features they need (e.g., automatic differentiation, hp-adaptivity optimized support for high-order methods on simplex meshes, parallel anisotropic AMR, etc.). In addition, modern hardware will lead more applications to switch to high-order methods, which would require both algorithmic and software support. We strongly believe that it is important to fund not just code development but also application engagement.*
- **Improve and maintain the code quality of our software at professional level. Help users and the broader community by improved documentation, tutorials and outreach – 1.0 FTEs (ongoing).** *We have accumulated a significant technical debt during the ECP: while many cutting-edge performant algorithms have been implemented and integrated in applications, the documentation, clarity, and user-friendliness of these algorithms have not always kept up and can be improved. We would like to take some time to re-engineer, polish, and document the code without adding new features so it is easier to maintain it long-term. We also want to increase the impact of our ECP work tutorials and community outreach and to remove any remaining barriers to adoption (e.g., Windows support, pre-build binaries, understanding-oriented documents, optimized ARM support, etc.).*

In the above case, the required ongoing post-ECP funding is 3.0 FTEs/year. The typical range found in an informal survey of LLNL software packages was 1.0-3.0 FTEs/year. We further note that funding uncertainty has a significant cost. It impedes growth, improvement, staff retention, and adoption by applications. Reducing this uncertainty with longer-term funding and/or block-funding that is

managed locally, with appropriate oversight, will be beneficial both for the software developers (who can focus on their work) and for the users (who will know that they can rely on us long-term).

Besides funding, we consider the following to be the largest challenges to maximizing impact:

- **Scaling the development as the project grows.** This concern affects all four areas but is particularly challenging for community outreach (which often requires developer expertise, and for successful open-source projects can grow and evolve substantially without a matching increase in resources) and for maintenance (due to increasing size and complexity of the code base and the architectures it needs to be tested and deployed to).
- **Hiring and retention of experts on the team.** This concern affects all four areas but is particularly challenging for the first two, since we often need advanced expertise in multiple fields (e.g., MATH + GPUs + Electromagnetics) and since there is a lot of competition for software talent in growing areas such as GPU programming. The technical debt when developers leave can be significant, so sustaining both the software and the team is important. While not discussed here, the planning and coordination of large team efforts is also a challenge.
- **Integration, coordination, and collaboration with other scientific software packages,** particularly with respect to building as part of automatic deployment and coordination when used in the same application (e.g., in terms of passing data to each other). We have found that grouping projects by computational motif, where the research topics and application targets are shared, to be beneficial for improved collaboration [50].

Key takeaways: *For scientific software to be adopted, users (application developers) will require that the project will have ongoing support so that key functionality can be relied on for the lifetime of their application. In addition to research and new capabilities, sustainability demands support for users, community outreach, and standard maintenance.*

Challenges in building and maintaining a diverse workforce

Diversity, Equity, and Inclusion activities have steadily increased in recent years, but within the HPC community, we have much work remaining to create and maintain an inclusive professional environment even without the challenge of building a diverse workforce in a field that includes companies and institutions infamous for sexist and unwelcoming behavior. Fortunately there are pockets of diverse HPC communities where management has some amount of diversity and the culture engages with and appeals to individuals outside the norm, including individuals of any race, color, caste, economic status, gender expression, gender identity, sexual orientation, disability, age, religion, national origin, and ethnicity.[†] Emulating and scaling the practices of these diverse and inclusive communities can improve life not only for those in the minority, but for everyone. A diverse and equitable community leads to more innovation, creativity, and motivation in our work. In our response, we first answer the questions about challenges, then offer successful strategies that we have employed. We address first the challenge of building a diverse workforce, and then building

[†] The following inclusive wording is taken from the RSE DEI Mission statement [51]: We welcome and respect individuals of all dimensions of diversity, including but not limited to race, color, caste, economic status, gender expression, gender identity, sexual orientation, disability, neurocognitive differences, age, religion (or lack thereof), national origin, and ethnicity.

and maintaining an inclusive professional environment. Achieving both of these goals strongly supports our ability to design, develop, support, and maintain software with an inclusive vision created by a collaborative workforce.

Challenges

Recruiting and retaining the talent we need to develop scientific software is a challenging problem for multiple reasons. Complex, production-quality multi-physics applications software requires skills and processes beyond what individuals trained on research software or coursework projects typically acquire. We need individuals that have multidisciplinary training that often is beyond the scope of what undergraduate and even graduate curricula provide. Summer and year-round internships are one answer to this problem but require filling the pipeline very early in their academic career.

Unfortunately, it is a challenge to retain those we have trained because they have valuable skills that are in demand by other companies. It is difficult for the national labs to compete with the pay and bonuses that companies provide to software engineers/developers. Other retention challenges include providing a good work-life balance, articulating clear career paths and providing advancement opportunities, and providing options for developers to change assignments and learn new skills.

In addition, HPC development environments can lag behind other environments in terms of using the latest tools and techniques because the tools need to be adapted to function securely within a complex, center-wide environment. Thus, we require skilled professionals who translate between HPC and cloud, web, and other technologies. Simplifying our developer and user workflows can go a long way towards retaining developers who can then focus on the scientific software rather unneeded complexity or environments requiring arcane knowledge.

Recruiting and retaining talented professionals from groups historically underrepresented in STEM and/or individuals from underserved communities is difficult. These individuals are in high demand throughout industry as the value of diversity is increasingly recognized, but in comparison to the demographics, we need to provide additional support to make sure they enter the pipeline and stay in the pipeline. There are many other challenges, including unconscious bias from even well-meaning recruiters and those who unconsciously behave in ways that do not create a psychologically safe space in meetings. When underrepresented minorities do not see people like themselves in positions of power, that signals a lack of equity and inclusion. While there are women in management, it is much less likely to find other kinds of diversity in HPC management.

Our traditional approaches to scientific and HPC software development do not promote inclusivity and equity. Modern, comfortable developer environments and good software engineering practices could lead to more inclusivity. Programming culture can feel exclusive and foreign to those who enter the field from less conventional backgrounds. Good onboarding and technical mentoring help people ramp up more quickly. Having software engineering metrics to measure impact would be helpful (and not just lines of code and bugs solved). Many organizations fail to value teamwork and the problem solving and social skills required to be successful in software projects. Technical skills are only one aspect; it is a challenge to ensure that other aspects are valued and rewarded.

Building a Diverse Workforce

Seeking to increase diversity in the HPC workforce often centers around recruiting at the college level, but it is widely acknowledged that we never attract to computer science the breadth of candidates that we need at either the college or high school levels. As early as middle school, students who appear not to fit the stereotype of the nerdy programmer or whiz at robotics, math, or science have been influenced to steer clear of STEM areas supposedly out of their reach. It is our aim to attack this challenge with two approaches: recruiting from institutions with more minority candidates and investing even more in STEM outreach for underrepresented populations so that they remain open to pursuing careers that lead them to scientific software development.

Recruiting

Effective recruiting for a diverse workforce requires broadening the search to include minority institutions and making connections with organizations that can help match candidates from underrepresented populations with job opportunities. At LLNL, we engage our workforce managers in outreach to universities with larger percentages of minority students, including UC Merced, Cal State East Bay, San Francisco State University, and University of the Pacific. We participate in road shows to these universities to visit with professors and students to hear about their university and to give presentations on LLNL and our internship programs. This effort has enabled us to build a pipeline of students who return multiple summers and some who stay on part-time during the school year. The students in turn recommend our internship programs to others. The Lab has a high rate of offers accepted from students in this pipeline. Livermore Computing has become known as an organization that accepts pipeline students and provides extra support in the first crucial months of employment. The new employee program helps new hires to quickly learn about the Lab and to build a network of co-workers, mentors, and friends in affinity groups such as the Employee Resource Groups (ERG), which are voluntary, employee-led organizations connected by common interests and bonds, and similar backgrounds. They offer professional development, recruitment, networking, mentoring, internships, scholarships, and laboratory committee and leadership opportunities, and they contribute to community outreach activities. Examples include African American Body of Laboratory Employees, American Indian Activity Group, Amigos Unidos Hispanics in Partnership, Asian-Pacific American Council, the Lawrence Livermore Lab Women's Association, and Livermore Pride. Through this extra support for new employees, we have addressed some of the challenges of retaining talented professionals from groups historically underrepresented in STEM. The ERGs provide leadership opportunities. In fact, our current and first female Lab Director had an early leadership position on the Parent Council for the Children's Center, and several women Division Leaders gained early leadership experience running the Lawrence Livermore Laboratory Women's Association. It is important in recruiting and retention to be able to reflect the diversity of the Laboratory in our management, which is a goal not yet reached, but these programs and groups are helping to change that through recruiting, mentoring, and sponsoring diverse employees for leadership roles.

STEM Outreach

LLNL works actively on community engagement through STEM education to positively impact STEM nationally, regionally, and locally. The Office of Strategic Diversity and Inclusion Programs partners with and supports many efforts, including America's Promise [52], Black Girls Code [53], California Alliance of African American Educators [54], EmpowHer [55], Engineer Girl [56], Expanding Your Horizons Network [57], Girls Who Code [58], Global Tech Women [59], and Hidden Genius Project [60]. Within the HPC community at LLNL, we actively support several of these organizations and

activities as mentors. We give presentations, mentor students, and develop programs to engage youth in learning about STEM content and careers. We participate in hands-on workshops, and we serve as mentors and role models.

Maintaining an Inclusive Professional Environment

Towards achieving an inclusive professional environment, LLNL aims to include underrepresented people in management, on committees, and in decision making; acknowledges the need for inclusive language; and introduces training on DEI as well as on how to react and interact in the moment that a negative behavior or situation occurs. LLNL fosters a culture that embraces inclusiveness and diversity as well as curiosity and learning because these qualities help us solve challenging problems. We strive to ensure that our employees are engaged and feel respected, accepted, included, and encouraged to participate. We recognize employee-led initiatives and behaviors that promote our culture's values through our Director's diversity and inclusion awards program. We encourage external community service that provides visibility for our diverse population and helps to promote positive change in HPC.

We strive to create a workplace that encourages and inspires our staff to promote personal and professional growth that assists in human understanding, creativity, empathy, innovation, and team engagement. LLNL develops staff in the diversity, equity, and inclusion areas through awareness, education, communication, and community experiences. Some activities include community events, interactive workshops, invited seminars, leadership development courses, skills development, diversity and inclusion awards and recognition program, bias education, and museum of tolerance immersive programs. Mentors are an important facet of maintaining an inclusive professional environment, from new employee and onboarding programs to leadership mentors. An active omsbud program offers employees a resource for resolving differences and working through conflicts. Holding informational interviews between group leaders and new employees in other groups helps employees to network and more quickly learn about an organization, while also providing leadership with feedback that can be used to improve the environment. Investing in such programs helps employees feel more connected to and supported by the organization.

Building multi-disciplinary, cross-organizational teams

LLNL tackles challenging problems by using large, multi-disciplinary teams. Some of our most successful software projects include research, development, and operational team members, so that, in our unified success, we have production-quality products guided by both research and operational concerns. One key to success is including people whose communication and collaboration skills are as good as their technical depth because we need to partner with the users and vendors. For diversity, we need to consider not only cultural diversity but neurodiversity, and how to help everyone not only get along but to become strong teams and allies. We provide training for managers on neurodiverse employee engagement and support. We have developed career planning and succession planning activities to help employees advance in their careers. We recognize employees through awards programs as well as spotlight articles in newsletters. We can build on our success in this area and expand it beyond our own lab, which we have already done in some areas within the Exascale Computing Project and multi-lab projects.

Increase the role of Research Software Engineers in the HPC community

The Research Software Engineering (RSE) movement [51] that started a decade ago in Europe has, as of 2018, come to the United States, presenting us with an opportunity for better collaboration

across academia, national labs, and industry. This opportunity includes (but is not limited to) improved engagement to create shared best practices for software development, education and training resources, and community. With a better understanding of the desires for careers and learning in this growing community, the lab can better adapt traditional roles, training regimens, and developer environments to cater to these needs and further attract a cohort of new software developers.

Inclusive Language

Inclusive language plays an important role in making a welcoming environment. We suggest participation in the Inclusive Naming Initiative as well as attending training and presentations, such as the SC21 session on inclusive language. Inclusion should also extend to interfaces, such as using captions on videos for the hearing impaired and adhering to accessibility standards on web sites.

Key Takeaways: *Building an environment that demonstrates DE&I takes a long-term investment that may take years to show dividends. Individual institutions must be part of the solution by demonstrating their commitment through targeted action but cannot solve the problem themselves and must also engage in community efforts to build a pipeline workforce that represents our DE&I goals. DOE ASCR should facilitate communication and encourage laboratory participation and collaboration in community DE&I efforts.*

Requirements for and barriers to technology transfer and community building

LLNL has a long history of successfully developing and transferring scientific software using a variety of methods that have resulted in sustained ecosystems. For example, the SLURM Workload Manager, an open-source resource manager used on HPC systems at LLNL and around the world, was co-developed with industry partners and has been supported by a small business founded in 2010 by SLURM developers. Today, LLNL routinely openly licenses the source code for its HPC infrastructure codes and math libraries, including signature software such as SUNDIALS, hypre, and Spack, to attract and build open communities of like-minded developers and users to support and maintain a common base of foundational scientific software.

LLNL scientific software applications also show remarkable resilience in the ecosystem. Our Numerical Electromagnetic Code (NEC) [61] has enjoyed decades-long success in the antenna modeling community. The SLAB atmospheric model [62] created by LLNL in the 1980s is still available and supported by commercial licensees. Incorporated Research Institutions for Seismology, a consortium of over 125 US universities dedicated to the operation of science facilities for the acquisition, management, and distribution of seismological data, has maintained and distributed the Seismic Analysis Code since 2006, including its latest release of version 102. Finally, a special success is the commercial life of DYNA3D [63], released from LLNL as unrestricted software in 1978. DYNA3D, and "DYNA-like" products have been estimated to save U.S. automotive industry billions by significantly reducing the number of automobile crash tests needed for safety analysis. Livermore Software Technology Corporation (LSTC) was formed by LLNL's lead developer of DYNA3D in 1989 to support the user community and commercialize code improvements under the product name LS-DYNA [64]. LSTC was sold for $775 million in 2019 to ANSYS, who continues to support, maintain, and advance the LS-DYNA commercial product.

Achieving these high levels of sustained success has required innovation, collaboration, and persistence on the part of LLNL's scientists, developers, and technology transfer professionals as well as the collaborating organizations and funding sponsors. Continuing our software technology transfer successes requires engaging and cultivating collaborations that enable the healthy scientific software ecosystem to flourish. LLNL technology transfer success means that we further advance our mission, improve U.S economic competitiveness, and advance the state-of-the-art of scientific computing and the impact our research and industrial scientific computing community.

LLNL offers the following specific comments on the important characteristics and components of sustainable models:

- **Measurable, sustainable value:** For many engaged software community members, "what is success" is generally understood but may be unstated and differ from other individual members' definitions. For example, scientific software developers may measure citations, open-source engagement metrics or commercialization results and value the qualitative rewards of positive collaborative relationships, and the impact of their software. On a personal level, what is considered "success" to developers varies widely but is the inherent reason why each invests their time and expertise in their software. Engaged scientific software developers who care about their code and their community are the heart of any sustainable technology transfer/community engagement model.

  Other engaged community members measure success from their own viewpoints. Academic and research organizations engage in communities to enable and advance their academic and scientific missions and train students or junior employees. Industrial partners engage as vendors to advance their technology/product offerings, sustain their supply chains and workforce, or as users seeking solutions to their scientific challenges. Key to sustaining the engagement of these community members over time is to understand the value propositions and demonstrate ongoing, measurable value that each considers a success.

  It is also important to look outside the existing engaged community members for new stakeholders. Defining a value proposition for investors requires expressing success in terms that resonate with their investment criteria, not the technical success criteria commonly used between scientific collaborators. Successfully generating adoption of a software requires expressing the benefits and advantages of the software as a solution to a problem that is worth the investment.

- **Clarity:** Sustainable technology transfer models need to be understandable and clear to the members and to potential new stakeholders. The early years of open-source licensing saw a vast proliferation of licenses as developers sought to clarify understanding and language. Ultimately, the OSS licenses that are most used today are those that are commonly understood and well-suited to common use cases. In collaborations, hosting/cloud contracts, and licensing arrangements aimed at maturing or sustaining a scientific code, clarity lowers the risk of misunderstanding and enables commitments to the value propositions, resource commitments, and collaborative efforts.

- **Flexibility and Adaptability:** Software is an agile and rapidly evolving technology that operates in a competitive market for hardware, services, and developer support. Topic areas like artificial

intelligence and cybersecurity are particularly fast moving, and research areas like quantum science are pressed by publication drivers and national competitiveness concerns. Data is now recognized as a valuable asset with evolving business models and IP landscapes and is critical for driving scientific understanding forward. Sustainable technology transfer models must account for the fast pace of change and adapt as conditions warrant.

- **Timeliness:** To drive adoption, it is critical to move faster. Open-source licensing is preferred not only because it is open for scientific inquiry and for interoperability reasons, but also because it is quick and easy to transfer code to the community. However, building and sustaining a community requires more than just available code. Establishing collaborations, attracting investment (of time and money), and driving adoption require actions that now must happen on much faster time scales than ever before.

Obstacles, impediments, and bottlenecks to technology transfer

As complexity of the software ecosystem increases and more organizations collaborate in technology design, development, and deployment, non-technical concerns also increase in complexity. The ecosystem for software technology transfer now is highly interactive with academic researchers, lab scientists and developers, and private sector partners of all kinds co-creating complex technical solutions involving a wide variety of use cases, intellectual property assets, investment requirements, and time scales.

Consequently, legal and business terms that optimize value for one engagement may not benefit the next opportunity with different circumstances – or worse, may raise barriers for engagement. The U.S. technology transfer system has long understood that continuing to advance the tools of the trade is essential and that enabling those evolving tools to be adapted and adopted widely increases engagement, reduces transactional uncertainty and timing, and yields measurable value. Keeping technology transfer models static yields consistency and predictability but risks viability and effectiveness in today's rapidly changing environment.

Past successes in overcoming technology transfer obstacles can guide new approaches for scientific software successes. A few examples include:

- **Passing the Bayh-Dole Act:** This legislation unlocked the value of Federally funded investments in research and technology development by enabling greater technology transfer options. Applying similar concepts to software-related intellectual property are in consideration across Federal agencies and technology transfer communities. Fortunately, changes can be made through guidelines, policy changes, and contract updates, not requiring changing statutes.
- **Establishing DOE's Modular CRADA:** Balancing public and private stakeholders' interests can be expedited when standard models are available. DOE established the CRADA Order and the Standard Modular CRADA in the mid-1990s to expedite industrial collaborations and to drive mission success. Later updates included terms to address software. Similarly, establishing standard models for scientific software collaborations and community engagements beyond R&D – with modularity and flexibility to adapt to circumstances – can

reduce uncertainty, expedite timing, and minimize resources required to form new engagements. Related activities have begun in DOE's technology transfer community and with DOE's Office of Technology Transitions (OTT).

- **Issuing "Policy Guidance – OSS License Release of Software Developed with ASC and OASCR Funding" in 2003:** Clearly articulating the value of open-source licensing and granting blanket permissions for copyright assertion for open-source licensing spurred acceptance and adoption of open-source licensing models across the DOE community. Its core principles have been adopted into a wider policy for DOE's Office of Science and incorporated into Laboratories' management contracts. Licensing open-source software now has clear value as an essential technology transfer tool for open science, mission success, and broad community engagements. Similarly, policies and guidance that broadly encourage the use of the best-available technology transfer tools for enabling adoption, external investments, and sustainable availability could yield stronger scientific software communities and new models for U.S. economic competitiveness. Again, LLNL, DOE-OTT, and technology transfer leaders across the complex as well as with DOE's applied program offices are piloting new approaches that could provide insights and new models for ASCR.

Practices to establish successful technology transfer

From our experience at LLNL, we have identified several practices that favor – though do not guarantee – successful technology transfer. Our recommendations are:

- Recognize early that adoption and sustainable use requires more than just a strong technical solution. Supportive development practices that consider successful technology transfer act on non-technical requirements while development is ongoing and funded. Example activities include building documentation and user interfaces, establishing developer codes of conduct and guidelines for accepting dependencies and contributions, tracking contributions, disclosing innovations early and often, and working with the laboratory's technology transfer office to determine external interests and viable models for attracting additional investment and interest outside of the main developer community.
- Solicit customer and end user needs and desires to drive feature and usability decisions during development. Building valuable technology while development is funded is much easier than finding investment later to rebuild or retrofit technology that fails to excite customers or to provide advantages over existing solutions.
- Seek funding for the actions needed post-development to adapt software for changing user needs, build the community, and drive adoption. Some DOE offices and other federal agencies now offer calls for "technology commercialization funds" that can be used for these projects.
- Team, team, team. A core technical strength for LLNL is its ability to build and manage effective cross-functional teams. Teaming is now an expectation for building scientific software, and it is even more important for multi-laboratory, multi-organization co-development and sustainable support. Just like collaborating among a diverse technical team is a force-multiplier for solving sticky technical challenges, teaming among the business and legal professionals resolves the murky challenges of intellectual property ownership, administrative processes, and licensing and collaboration agreements. Again, engagement across the DOE tech transfer complex is ongoing to facilitate the business and legal frameworks that make multi-lab collaborations easier and faster.

- Be open to new innovations in business models. LLNL is actively engaged with DOE's OTT and other labs to innovate new technology transfer programs. A very promising project is the "Open Source Software: Seeds of Commercialization" project funded by OTT as part of its Practices to Accelerate the Commercialization of Technologies (PACT) program. In this ongoing project, LLNL's tech transfer lead is actively engaged with LLNL developers to identify best practices in and pilot new ideas for maturing and transferring sustainable software programs. As this program evolves, new best practices are being identified and documented for reproducibility across DOE.
- Finally, explore new mechanisms to partner with the fast-moving small business and start-up ecosystem. Cutting edge software technology has adapted a fast-fail, sweat-equity based model in which success or failure is decided in a matter of months and initially resources are severely limited. This model does not align with the current CRADA based, cost up-front model of technology transfer. However, engagements outside of CRADAs are hampered by stringent conflict of interest rules and the lack of incentives. Finding new frameworks that enable DOE staff to interact with the small business technology culture can drastically increase the number of opportunities for tech transfers and significantly accelerate their development.

Building communities around software projects

Having a community supporting and advancing a software project is a desirable outcome because it helps to ensure the longevity and support for a code base while distributing the responsibility and resources needed to support it. The many successes of open-source software, e.g., Linux, Python, MySQL, etc. all have dedicated and passionate communities. Specific DOE examples, such as Spack and VisIt, demonstrate that building such a community takes time and purposeful action beyond just having a desirable software tool. Software teams need to engage in outreach, workshops, tutorials, writing good documentation, "advertising", etc. Once software attracts users, additional time (and related resources) are needed to provide quality support to users, and more users means more work, which means a substantial cost in developer time. The time for community engagement and support needs to be explicitly recognized and funded. In addition, community code development introduces challenges in managing software design and development. For scientific software projects, where much of the code is specialized and interconnected, changes need to be carefully reviewed to maintain the software quality, but active community engagement requires efficient review and inclusion of community-provided capabilities.

Building communities through Foundations

For large enough code projects, it may make sense to join or start 501(c)(3) or 501(c)(6) projects (e.g., NumFOCUS [65] or the Linux Foundation [66]). These organizations can help popular projects build a larger community and raise funds. They have marketing and event teams and can organize large gatherings such as user group meetings. They also provide a neutral home and clear governance for projects that may be perceived to be dominated by a single organization.

Key takeaway: *Technology Transfer is one way that the National Laboratories can provide direct economic value to the U.S., but it has been proven to be a difficult process to navigate. Unlike many of the topics in this RFI, most of the advances in this area have been – and will need to be – through policy innovations.*

Overall scope of the stewardship effort

The proposed potential scope (Training, Workforce Support, Infrastructure, Curation, Maintaining Situational Awareness, Shared Engineering Resources, and Project Support) covers the majority concerns, which we have discussed throughout the previous sections. Of these activities, we would prioritize project support, workforce support, curation, and infrastructure as critical to the software stewardship enterprise. Training, Maintaining Situational Awareness, and Shared Engineering Resources, while important, are less central.

Furthermore, as defined in the RFI, the proposed scope fails to explicitly identify several other key activities: Application Engagement and Support, Incorporation of New Capabilities, and Community Outreach. A significant challenge for DOE ASCR, which is not the case in NNSA's ASC program, is that DOE ASCR does not – and should not – own the application codes; these more naturally should belong to and be funded by the science and engineering programs to which they are relevant. Thus, coordination and multi-project collaboration will be more difficult for a software stewardship effort, and careful consideration will need to be given to how such a software stewardship activity will fund application engagement and support, potentially participating in or superseding partnership programs such as SciDAC. The transition of DOE-ASCR-funded research results into sustainable production software as well as the implementation of new features in response to user input will be important for software to retain relevance, to increase ASCR research impact, and to engage the user community. Community Outreach deserves explicit consideration since software technologies are of little interest if they are not recognized and adopted by user communities. "Training", as defined in the RFI includes both software developer training as well as training in the use of core software and "Maintaining situational awareness", which incorporates documentation and communication about software, insufficiently capture the importance of Community Outreach.

A final concern about the proposed decomposition of scope is that obfuscates the key stakeholders and the channels of communication between them. Broadly, the scientific software roles can be thought of as researchers, developers, and users. Historically, these often have been the same person, with "development" being seen as a task that required no particular expertise other than the ability to program in a performant language. Developing software is resource intensive, however, so we should make the most of such investments by practicing good software engineering so that our scientific software has longevity. Prior to ECP, this activity has seldom been directly supported in DOE SC, so a software stewardship effort would be a new component of DOE ASCR activities that will need to interface with the research and user communities and facilities. The proposed scope mostly describes the responsibilities within the software stewardship component without much description of the interactions and information exchanges with researchers, users, and facilities. From our experience, it is best if these concerns are championed but not segregated. To be successful, we believe a software stewardship activity cannot be an activity insulated from users, research, facilities, or vendors, so it will be critical to identify and incentivize the ways research ideas will make their way into production software, how user needs will be addressed by sustainability efforts, etc.

Key takeaways: *Of the stewardship topics suggested in the RFI, LLNL would recommend prioritizing project support, workforce support, curation, and infrastructure as critical to the software stewardship enterprise and would recommend adding application engagement and support, incorporation of new capabilities, and community outreach as additional priorities. A challenge for ASCR will be to develop a model, perhaps inspired by our ASC experience coupled with SciDAC, that can bridge the software and application teams housed across Office of Science Programs.*

Management and oversight structure of the stewardship effort

A robust, coordinated software stewardship effort will require the engagement of personnel across the DOE laboratories, in industry, and in academia. DOE labs will likely need to have a leadership role, as many of the software technologies originate within the labs and connection to the DOE ASCR researchers will likely play an ongoing role in the support and further development of new capabilities. The labs are generally better positioned than academia to support long-term, production-quality software development, and technology transfer to the private sector is not always practical, particularly if the expertise behind software capabilities remains within the DOE labs. There are private-sector companies, such as KitWare, TechX, and IntelligentLight, who have much to offer as active team members for software products that have more commercial appeal, but DOE Laboratory computer scientists have more direct access to domain scientists and their scientific computing issues and needs. Labs need to be involved in project governance to ensure that software projects remain relevant. Software foundations with buy-in from the labs to support developers at companies such as Kitware and TechX is a model that would work well for public-private partnership, provided that barriers to efficient collaboration and technology transfer can be reduced.

Another concern that should inform management decisions is the need for stability. A current barrier to adoption is that application developers do not want dependencies on software that has questionable support or longevity. Software developed by consortia can overcome this hurdle, but it is difficult for most software projects to achieve this level of widespread use. Commitment to the long-term sustainment of software products is therefore imperative to their success, both in the eyes of the customers as well as to the developers supporting software.

One additional consideration is that, while top-level management is necessary to set goals, to define standards, and to measure progress, scientific software thrives best in an environment where the grass-roots defines directions and approaches. Those with hands-on-keyboards see the real challenges of scientific computing first-hand and are best positioned to both identify problems and to recommend solutions. Indeed, at LLNL, we have benefited from the close relationships between research, development, and production deployment in our mission, and a successful software stewardship effort will need to be able to adopt the results of research efforts, harden these for production use, address the new needs of users, and communicate new research questions back to ASCR researchers.

Given this context, we recommend that the management structure of a software stewardship program involve a high degree of local control and responsibility. At the top should be a small management team of 2-3 expert software development managers, drawn from the national laboratories, who work closely with the DOE Program Manager to set overall goals, monitor progress, and make high-level direction and funding decisions. Each laboratory would have a Program Lead who reports to the management team and negotiates the activities and deliverables associated with the scientific software portfolio of each laboratory. These local Program Leads, with the management team, would constitute a "DOE Software Council" that ensures the efforts are coordinated across the DOE labs. Given a proposed funding level from the management team, the efforts at each laboratory would be proposed as part of an annual work package with measurable deliverables and milestones, with the laboratory recommending how the resources would be allocated to each software effort. Local Program Leads would have latitude to adjust local resources as issues and needs arise, keeping the management team informed of these decisions. Laboratories would be responsible for managing subcontracts with academia as appropriate. New software tools could be proposed to be added to a

laboratory's software portfolio, with additional funding, to a review committee comprised of representatives from across the laboratories – perhaps the DOE Software Council itself. The overall stewardship effort should be reviewed periodically by an external panel of government and industry experts in scientific software and software development.

Key Takeaway: *For agility to respond to user needs, LLNL recommends a program with a lightweight hierarchy of oversight and a focus on local control of technical decisions at the laboratory project level.*

Assessment and criteria for success for the stewardship effort

Like verification and validation, there are two obvious metrics of success for the scientific software stewardship effort: completion of proposed work (did the project complete what is proposed to do?) and impact (are the products of the effort useful?). Of course, the former is easiest to measure, as it relies on milestone reports and deliverables. Other relevant "verification" metrics could include compliance with project standards, up-to-date documentation, and even number of feature requests or bug reports addressed.

More interesting, but more difficult to quantify, is impact, although a variety of surrogates can be identified. Number of downloads has long been a weak surrogate for community adoption, and such figures could be augmented with statistics about deployment to HPC sites worldwide. DOE facilities could help collect and report data on how frequently tools or libraries are used (number of applications using a library, number of unique runs accessing a library, etc.). Citations could be counted if each software package that is supported under the stewardship program maintained a standard document to cite when publishing results that depended on the software. None of these measures are precise, but, they can help to demonstrate adoption and impact of scientific software supported in this effort.

Key Takeaway: *No single metric or criteria can suffice to measure success of stewarding a single project or even the program as a whole, but a palette of oft-used metrics can be defined and refined.*

Summary

LLNL strongly supports the formation of a software stewardship component distinct from research, computational partnerships, and facilities within the DOE ASCR Program. A reliable software ecosystem is essential to access the capabilities of modern computational resources, and such an ecosystem needs to be healthy and sustainable. Significant investments in scientific and high performance software stewardship are a necessary complement to investments in new high performance computing hardware. The hardware is of little use to the science and national security missions of DOE without the software upon which application codes are built. As the originators of much of this software, the national laboratories are well positioned to lead software stewardship of scientific and HPC software, with support from the private sector and academia as appropriate, but this will require a new ASCR subprogram with a management structure and oversight different from ASCR's research program.

References

[1] OpenZFS on Linux and FreeBSD.

[2] SLURM: SLURM Workload Manager.

[3] Flux: Resource management framework.

[4] Gamblin, T., M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. "The Spack Package Manager: Bringing Order to HPC Software Chaos," In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. LLNL-CONF-669890.

[5] MFEM: Lightweight, general, scalable C++ library for finite element methods.

[6] *hypre*: Parallel solvers for sparse linear systems featuring multigrid methods.

[7] SUNDIALS: Suite of Nonlinear and Differential/Algebraic equation Solvers.

[8] ROSE: Open-source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C, C++, UPC, Fortran, OpenMP, Java, Python and PHP applications.

[9] BLT: Streamlined CMake-based foundation for Building, Linking and Testing large-scale high performance computing (HPC) applications.

[10] Black, A., R. Hornung, M. Kumbera, R. Neely, and Rieben R., "Computer science recommendations for LLNL ASC Next- Gen code." Technical Report LLNL-TR-658622, Lawrence Livermore National Laboratory, 2014." You can see it e.g., as [1] in https://doi.org/10.2172/1724326

[11] E4S: Extreme-Scale Scientific Software Stack.

[12] RADIUSS: Rapid Application Development via an Institutional Universal Software Stack.

[13] EESSI: European Environment for Scientific Software Installations

[14] Boissonneault, M., B. E. Oldeman, and R. P. Taylor. "Providing a Unified Software Environment for Canada's National Advanced Computing Centers," *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pp. 1-6, 2019.

[15] LBANN: Livermore Big Artificial Neural Network Toolkit.

[16] Maestro: Maestro Workflow Conductor: Developing Sustainable Computational Workflows.

[17] Merlin: Simplifying Machine Learning Based Workflows on High-Performance Computers.

[18] PARSL: Productive parallel programming in Python.

[19] Kubernetes: Production-Grade Container Orchestration.

[20] ESGF: Earth System Grid Federation.

[21] Abdulla, G. M. *Annual Earth System Grid Federation 2019 Progress Report.* United States: N. p., 2019. doi:10.2172/1530675.

[22] Newman, L. H., "Equifax Officially Has No Excuse," Wired, https://www.wired.com/story/equifax-breach-no-excuse/.

[23] https://esgf.llnl.gov/esgf-media/pdf/ESGF-Software-Security-Plan-V1.0.pdf

[24] https://gcn.com/articles/2014/05/14/fose-mosaic-effect.aspx

[25] https://www.archives.gov/cui/registry/category-list

[26] https://csrc.nist.gov/Topics/Laws-and-Regulations/laws/FISMA

[27] https://csrc.nist.gov/publications/detail/sp/800-171/rev-2/final

[28] git distributed version control system.

[29] GitHub: Where the world builds software

[30] GitLab: The DevOps Platform has arrived.

[31] Jupyter: Project Jupyter develops open-source software, open standards, and services for interactive computing across dozens of programming languages.

[32] Pandas: a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

[33] Kurtzer, G. M. et al. "Singularity: Scientific containers for mobility of compute." *PLoS ONE* 12, 2017.

[34] Priedhorsky, R. and T. Randles, "Charliecloud: unprivileged containers for user-defined software stacks in HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17).* Association for Computing Machinery, New York, NY, USA, Article 36, 1–10., 2017

[35] Al Saadi, A. et al. "ExaWorks: Workflows for Exascale." ArXiv abs/2108.13521, 2021.

[36] Labarta, J. et al. "Executive Summary" In Proceedings of the Dagstuhl Seminar on Program Development for Extreme-Scale Computing, Dagstuhl, Germany, May 3-7 2010.

[37] HPCToolkit: An integrated suite of tools for measurement and analysis of program performance.

[38] RAJA: Performance Portability Layer in C++.

[39] Edwards, H. C. and C. R. Trott, "Kokkos: Enabling Performance Portability Across Manycore Architectures," *2013 Extreme Scaling Workshop (xsw 2013),* 2013, pp. 18-24, doi: 10.1109/XSW.2013.7.

[40] Eichenberger A. E. et al., "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis," in: Rendell A.P., Chapman B.M., Müller M.S. (eds) *OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013. Lecture Notes in Computer Science*, vol 8122. Springer, Berlin, Heidelberg, 2013.

[41] Schulz, M. et al., "The MPI Tool Interfaces: Past, Present, and Future—Capabilities and Prospects," doi:10.1007/978-3-030-66057-4_3, 2021.

[42] Boehme, D. et al. “Caliper: Performance Introspection for HPC Software Stacks,” in *Supercomputing 2016 (SC16)*, Salt Lake City, Utah, November 13-18, 2016.

[43] timemory: A Modular C++ Toolkit for Performance Analysis and Logging.

[44] SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure - a scalable C++ framework for block-structured AMR application development.

[45] VisIt: Visualization and Data Analysis for Mesh-based Scientific Data.

[46] ZFP: Compressed numerical arrays that support high-speed random access.

[47] HiOp: HPC solver for nonlinear optimization problems.

[48] Conduit: Simplified Data Exchange for HPC Simulations.

[49] SCR: Caches checkpoint data in storage on the compute nodes of a Linux cluster to provide a fast,
scalable checkpoint/restart capability for MPI codes.

[50] CEED: ECP co-design Center for Efficient Exascale Discretizations.

[51] https://us-rse.org/about/dei-mission/

[52] https://www.americaspromise.org/

[53] https://www.blackgirlscode.com/

[54] https://www.mightycause.com/organization/California-Alliance-Of-African-American-Educators

[55] https://www.empoweringher.org/

[56] https://www.engineergirl.org/

[57] https://www.techbridgegirls.org/what-we-do/stem-events/

[58] https://girlswhocode.com/

[59] https://www.globaltechwomen.com/

[60] https://www.hiddengeniusproject.org/

[61] https://en.wikipedia.org/wiki/Numerical_Electromagnetics_Code

[62] Ermak, D L. User's manual for SLAB: An atmospheric dispersion model for denser-than-air-releases. United States: N. p., 1990. Web. https://www.osti.gov/biblio/6252170-user-manual-slab-atmospheric-dispersion-model-denser-than-air-releases

[63] Whirley, R. G. and B. E. Engelmann, DYNA3D: A Nonlinear, Explicit, Three-Dimensional Finite Element Code for Solid and Structural Mechanics – User Manual, Lawrence Livermore National Laboratory report UCRL-MA-107254-Rev.1, 1993. doi: 10.2172/10139227

[64] https://www.lstc.com/products/ls-dyna

[65] https://numfocus.org/

[66] https://www.linuxfoundation.org/