Position Papers for the ASCR Workshop on the Science of Scientific-Software Development and Use

https://www.orau.gov/SSSDU2021

David E. Bernholdt[1], John Cary[2], Michael Heroux[3], and Lois Curfman McInnes[4] (eds.)

December 2021

[1]Oak Ridge National Laboratory
[2]University of Colorado
[3]Sandia National Laboratories
[4]Argonne National Laboratory

Disclaimer

The position papers in this collection were submitted in preparation for an event sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

U.S. Department of Energy Point of Contact:
Hal Finkel, hal.finkel@science.doe.gov

https://doi.org/10.2172/1843575

Contents

Corresponding Author	Title
Organizing Committee	Call for Position Papers
Organizing Committee	Pre-Workshop Document
Ahn, Dong	A large multi-disciplinary drug design team's perspective on sustainable computing software for transformative science
Aiken, Alex	Specifying and Debugging Data Distributions at Scale
Anzt, Hartwig	Continued Investment to Secure Future-Readiness of Scientific Software
Arienti, Marco	An experience-based perspective on scientific software integration
Balaprakash, Prasanna	MLOps
Baldwin, Breck	Funding Strategies for Scientific Software
Bel, Oceane	Cybersecurity in High Performance Software Development: Challenges and Approaches
Bethel, Wes	Fostering Interoperability and Increasing Scientific Productivity in Environments of Heterogeneous
Bhatele, Abhinav	Machine Learning to Analyze and Improve the Development of Scientific Software
Brewster, Aaron	The challenges of scientific software sustainability
Candy, Jeff	Scientific-software Development and Lifetime
Carleton, Anita	Architecting the Future of Software Engineering
Carr, Leslie	Explaining the Value of Research Software
Carver, Jeffrey	Scientific Software Development Laboratory: A Proposal
Chen, Jin	On the need for efficient and scalable solvers for ill-conditioned sparse matrix equations
Chiang, Nai-Yuan	Maintaining Productivity and Increasing Community Outreach for Optimization Ecosystem
Cholia, Shreyas	The Last Mile in Software Engineering
Clark, Struan	Adopting Industry Software DevOps Practices to Improve Scientific Software
Crawford, Daniel	Engaging a Complex Domain Science in Software Engineering Best Practices: What Works and What Doesn't
Crozier, Paul	How to measure and improve the quality and quantity of software developed by agile teams of scientific software
Crystal-Ornelas, Robert	Best practices to enable a culture of effective cross-functional scientific software collaboration

Deelman, Ewa	The proof is in the Pudding: Detecting problems with software in production
Deslippe, Jack	Utilizing Facility Data and Best Practices to Guide Sustainable
Dong, Bin	High Productivity Data Analysis Infrastructure for Scientific Discoveries
Doucet, Mathieu	Understanding the patterns driving the development of software to support science
Dubey, Anshu	Sustainability of Software Meant for Exploration
Du, Caifan	Outlining the Complexities of Scientific Software Development And Use
Dudson, Ben	The plural of anecdote is data
Fagnan, Kjiersten	Put an end to just in time planning
Gallo, Andy	Coercive Software Process Improvement
Gamblin, Todd	Automating Software Integration for Sustainability
Gaston, Derek	Reliable, Efficient Workflows for Complex Scientific Software Development
Gesing, Sandra	Accelerating Changes in the Scientific-Computing Community to Improve Practices for Software Development and Use
Gesing, Sandra	SGCI - Advancing Access to Scientific Software via Science Gateways Improves the Trust within the Computational Sciences Landscape
Ghosh, Sayan	Performance Portable Data Movement Support for MPI using C++
Gopalakrishnan, Ganesh	The Unfledged State of Heterogeneous System Testing
Graziani, Carlo	Data Lifecycle Assurance in Scientific HPC
Gunter, Dan	Understanding Complex Software Dependencies for Better Scientific Reproducibility
Guo, Yanfei	Automated Validation and Verification for Scientific Software
Gupta, Rinku	The role of culture in scientific software teams
Harrington, Kyle	Quantitative methods for research software community management
Harrison, Cyrus	Adventures in Modularity - Why is Sharing Hard?
Haupt, Carina	Provenance based software dashboards
Heber, Gerd	Scientific Software as Emergent Behavior
Hickernell, Fred	Four Ways to Grow Scientific Software
Hoffman, Bill	The Problem with Modern CI
Jay, Caroline	Theory-Software Translation Challenges

Kale, Vivek	More Frequent and Regular Engagement for Success of Scientific-Software
Katz, Daniel S.	Software sustainability is people
Keahey, Kate	Software as a Scientific Instrument
Kowalkowski, Jim	High Energy Physics and Scientific Software Development
Kowalski, Karol	Sustainability of Scalable Computational Chemistry Software at PNNL
Krishnan, Harinarayan	Capturing the varied complex software lifecycle processes at a scientific user facility
Laguna, Ignacio	Compiler Analysis for Early Detection of Software Defects
Laguna, Ignacio	Reproducibility in the Era of Heterogeneous Computing
Landin, Kirk	Toward a Science of Abstraction Design in Software
Laney, Daniel	Community Building Blocks for Workflows
Lenhardt, W. Christopher	Using the Science of Team Science for Better Science Software and Better Science
Leung, Mary Ann	Transformative Profiles in Research Software Teams: DEI for Collaborative Innovation
Liao, Chunhua	Automated, Extensible Correctness Diagnostics for Scientific Computing
Liao, Chunhua	Reinventing the Ecosystem to Improve HPC Software
Liu, Xu	Bridging Software Developers with Dynamic Program Analysis in IDE
Li, Xiaoye	Enhancing Software Performance Portability and Reproducibility
Lofstead, Jay	The Opportunity and Challenge of Serverless Computing
Malviya, Addi Thakur	The New Science of Scientific Software Development
Mcdonnell, Marshall	Infrastructure-as-Code for Scientific Software and its Importance Towards Federated Infrastructure
Mclemore, Jacob	Microproductivity in the Lab
Meinecke, Annika	Considering Activity-Centered Design in Scientific Software Development
Mellor-Crummey, John	Developing a Sustainable Software Ecosystem for Tools, Runtime Libraries, and Applications
Meng, Xiaozhu	Deep Learning Assisted Performance Analysis and Diagnosis
Menzies, Tim	When Can We Trust Your Models
Milewicz, Reed	Building Bridges: Establishing a Dialogue Between Software Engineering Research and Computational Science

Miller, Ross	Emphasis on Software Maintenance for Scientific Software
Mundt, Miranda	Working in Harmony: Towards Integrating RSEs into Multi-Disciplinary CSE Teams
Nayak, Pratik	Introducing RSE as a practice in CS courses
Norris, Boyana	Joint Analysis of Development Processes and Code Quality
Paine, Drew	Improving Software Sustainability Through Adoption and Democratization of User
Pawlowski, Roger	Developing Objective and Quantifiable Processes for Assessing Software Component Usefulness
Pearce, Olga	Performance Diagnostics and Debugging at Scale
Peisert, Sean	Improving Assurance of Scientific Software
Peles, Slaven	Software for Simulations and Analysis of Complex Engineered Systems
Perciano, Talita	AI/ML and Scientific Software Development and Distribution, Maintenance, and Code Teams
Perr-Sauer, Jordan	Evidence-based interventions for research institutions
Peterson, Peter	The call to quantify the impact of scientific software
Pirkelbauer, Peter	Benchmark Suites
Pollard, Samuel	Real(istic) Specifications of Software
Poon, Sarah	Design Systems for Science
Rajamanickam, Sivasankaran	Can Scientific Software Development Use the Outsourcing Model Successfully?
Raybourn, Elaine M.	A Seat at The Table
Richard, Ryan	Is a Language Barrier Impeding Development of Better Scientific Software?
Ringler, Todd	Towards a More Sustainable Software Development Environment
Rogers, David	Investigating Software Developer Roles Supporting Reproducibility
Roth, Philip	Viewing Scientific Software Development Through The Lens of Economics
Rouson, Damian	Expanding the Scientific Software Development Community Necessitates Studying Cognitive and Social Issues in Programming-Language Learning and Bias
Ruehl, Kelley	Improving adoption of scientific software best practices through the development of a software Impact Factor
Sachdev, Jai	Balancing Innovation and Stability of Long Lived Scientific Software with Containers

Seal, Sudip	Revisiting Software Abstractions, Standards and Best Practices
Serra, Edoardo	Enhancing Scientific Software Development with AI-Capabilities
Sexton-Kennedy, Elizabeth	HEP Software Challenges
Shiraiwa, Syun'Ichi	Accelerate high fidelity physics simulations for fusion energy using machine learning technology
Shrager, Jeff	Cognitive Analytic Engines are Critical to Scientific User Facilities Reaching their Potential
Sims, Benjamin	Social science research is essential to the future of the scientific
Smith, Spencer	Long-Term Productivity Based on Science, not Preference
Sochat, Vanessa	Improving Development Practices through Culture and Collaboration
Sommer, Thorsten	Combining Repo and Process Mining for Better Sci Software
Sottile, Matthew	Formal Methods White Paper
Stoyanov, Miroslav	The Challenges in the Development of Math Libraries
Suetterlein, Joshua	Software Practices for Cutting-Edge Hardware Evaluation
Tallent, Nathan	Studying Data Patterns to Enable Scientific Exploration with Data-Intensive Distributed Workflows
Terry, Jeff	Literature Validation and Experimental Analysis Using Advanced Machine Learning Tools
Tharrington, Arnold	Software Engineering Mechanisms To Facilitate Best Practice Design Principles
Thayer, Jana	Challenges and Opportunities for Scientific-Software Development Research
Tripp, Charles	Fundamental Needs for Establishing Evidence-Based Scientific Software Engineering Practices
Vasquez, Valeri	Using big data to study and improve scientific software
Watson, Gregory	Accelerating Excellence in Scientific Software
Wiese, Jason	Building Human-Centered Computing Knowledge of Scientific Software Development
Willenbring, James	A Testing Strategy that Supports Scientific Software Sustainability
Wolf, Matthew	Engineering Scientific Workflows for the Scientists of the Future
Wozniak, Justin	Software Monsters: Quantifying, Reporting, and Controlling Composite Applications

Wu, John	Beyond Open Source: A Call to Rethink Policies and Incentives for Sustainable Scientific Software Development
Xu, Haowen	Metaverse for Urban Informatics
Zhang, Chen	AI-assisted Collaboration between Domain Scientists and Research Software Engineers
Zhou, Hui	Manage Complexity With Generic Meta Programming
Zisman, Sagi	Progress in the Study of Scientific Software Requires a Rigorous Validated Taxonomy

Call for Position Papers: Workshop on the Science of Scientific-Software Development and Use

Important Dates

- 19th November 2021: Deadline for position paper submission
- 3rd December 2021: Notification of position acceptance
- 13 - 15th December 2021: Workshop
- SUBMISSION URL: https://orausurvey.orau.org/n/SSSDU.aspx
- WORKSHOP URL: https://www.orau.gov/SSSDU2021

Motivation

Software is an increasingly important component in the pursuit of scientific discovery. Both its development and use are essential activities for many scientific teams. At the same time, very little scientific study has been conducted to understand, characterize, and improve the development and use of software for science.

Computational science teams have diversified over time to include contributions from domain scientists who provide expertise in scientific and engineering disciplines, applied mathematicians and computer scientists who provide optimal algorithms and data structures, and software and data engineers who provide methodologies and tools adapted and adopted from other software domains. These diverse contributions have enabled tremendous advances in the pursuit of scientific discovery, even as models, computer architectures, and software environments have become more complicated.

With this increasing diversity, we believe the next opportunity for qualitative improvement comes from applying the scientific method to understanding, characterizing, and improving how scientific software is developed and used. We believe that this pursuit requires expertise from computational scientists themselves, and from the cognitive and social sciences as well as the software engineering research community.

As we look to increase the productivity and sustainability of the scientific-software-development-and-use cycle, a more systematic application of the scientific method to understand processes for software development and use [1,2] will be a valuable tool to guide future work and result in more usable and sustainable software. This workshop will bring together computer scientists, software engineering researchers, computational scientists, applied mathematicians, social scientists, cognitive scientists, and others, to explore how we

can conduct such systematic investigations, what can be learned, and how doing so will benefit the scientific enterprise.

The workshop will be structured around a set of breakout sessions, with every attendee expected to participate actively in the discussions. Afterward, workshop attendees — from DOE, industry, and academia — will produce a report for ASCR that summarizes the findings of the workshop.

[1] https://bssw.io/blog_posts/research-software-science-a-scientific-approach-to-understanding-and-improving-how-we-develop-and-use-software-for-research

[2] https://www.exascaleproject.org/better-scientific-productivity-through-better-scientific-software-the-ideas-report/

Invitation

We invite community input in the form of two-page position papers that identify and discuss key challenges and opportunities in the science of the scientific-software development process and the study of the use of scientific software. In addition to providing an avenue for identifying workshop participants, these position papers will be used to shape the workshop agenda, identify panelists, and contribute to the workshop report. Position papers should not describe the authors' current or planned research, contain material that should not be disclosed to the public, recommend specific solutions, or discuss narrowly-focused research topics. Rather, position papers should aim to improve the community's shared understanding of the problem space, identify challenging research directions, and help to stimulate discussion.

One author of each selected submission will be invited to participate in the workshop.

By submitting a position paper, authors consent to have their position paper published publicly.

Authors are not required to have a history of funding by the ASCR Computer Science program.

Submission Guidelines

Position Paper Structure and Format

Position papers should follow the following format:

- Title
- Authors (with affiliations and email addresses)
- Challenge: Identify limitations of state-of-the-art practice with examples
- Opportunity: Describe how the identified challenges may be addressed, whether through new tools and techniques, new technologies, new methodologies, or new groups collaborating in the process

- Timeliness or maturity: Why now? What breakthrough or change makes progress possible now where it wasn't possible previously? What will be the impact of success?
- References

Each position paper must be no more than two pages including figures and references. The paper may include any number of authors but contact information for a single author who can represent the position paper at the workshop must be provided with the submission. There is no limit to the number of position papers that an individual or group can submit. Authors are strongly encouraged to follow the structure outlined above. Papers should be submitted in PDF format using the designated page on the workshop website.

Notional Questions

Position papers should present views on why and how scientific-software development and use can be studied as a scientific endeavour, perhaps taking inspiration from some of the following:

- What methods can be used to study the development and/or use of scientific software?
- What methods either uniquely apply or uniquely do not apply to the development of scientific software?
- How can we best integrate social and cognitive sciences into scientific software activities?
- How can we study changes in culture affecting scientific software development and use?
- What unique changes in scientific software development would improve productivity, accuracy, trust, and/or reliability?
- How can we motivate changes in the scientific-computing community to improve practices for software development and use?
- How will AI-driven development tools affect best practices for scientific software development and use?
- How will static and dynamic analysis methods affect the practice of scientific software development?
- How are scientific software developers and users atypical from other larger software communities?
- When and how can research and methods from other software communities be adapted and adopted for scientific software?
- How is the field of computational science changing, and how will it change?
- What are the roadblocks faced by computational scientists in computational application use?
- What are the challenges faced by computational scientists in their pursuit of discoveries?

Selection

Submissions will be reviewed by the workshop's organizing committee using criteria of overall quality, relevance, likelihood of stimulating constructive discussion, and ability to contribute to

an informative workshop report. Unique positions that are well presented and emphasize potentially-transformative research directions will be given preference.

The Science of Scientific-Software Development and Use

Workshop sponsored by the U.S. Department of Energy (DOE)
Office of Advanced Scientific Computing Research (ASCR)

Organizing Committee:
Michael A. Heroux, Sandia National Laboratories
David Bernholdt, Oak Ridge National Laboratory
Lois Curfman McInnes, Argonne National Laboratory
John Cary, University of Colorado

DOE Point of Contact: Hal Finkel, DOE/ASCR

Workshop website: https://www.orau.gov/SSSDU2021
Pre-Workshop Report
December 9, 2021

Introduction

Software development and use play a central and growing role in scientific discovery. In some communities, the primary tools are spreadsheets and similar mainstream software adopted for scientific research. In other communities, the specialized hardware, software, and algorithmic requirements call for community-developed software capabilities that are typically designed and written by scientists, who often use the software for their own research, while also providing it to others.

In recent years, the impact of scientific software has increased, with computational and data-enabled science and engineering pervading nearly all aspects of discovery, as complements to experimental and theoretical research. At the same time, scientific software has become more complex due to advances in both computer architectures and next-generation science challenges. As a result, software teams have grown more diverse, including people with computer science skills to assure appropriate algorithm and data structure choices, as well as software engineers to improve software tools, practices, and processes. A variety of community reports have expressed the importance of increasing direct investment in scientific software itself, not just as a byproduct of other research [1, 2, 3, 4, 5, 6, 7, 8]. Consequently, we now have the unique opportunity to fundamentally change how scientific software is designed, developed, and sustained, while addressing urgent challenges in workforce training and recruitment [9, 10, 11, 12, 13].

The art and craft of scientific software development and use have advanced over many decades, primarily benefiting from adapting and adopting advances from outside the scientific community. Advances from the broader software community in programming languages, computer system design and implementation, and software development tools and processes have strongly influenced (in both positive and negative ways) scientific software activities. Research focused on improving software development and use is certainly progressing. Numerous studies continually emerge from the software engineering literature, where authors study practices, methodologies, tools, and human factors with the goal of understanding and improving the impact of software for its intended use, at reduced cost and faster delivery schedule. Even so, the scientific software community is a small part of the overall software ecosystem and, from our observations, has received only modest attention from the professional software community in comparison with other target domains.

Presently, the scientific software community is exploring possibilities to include cognitive and social science, as well as advancement in artificial intelligence (AI), to further improve the effectiveness and efficiency of software in the pursuit of science. The 2021 Workshop on the Science of Scientific-Software Development and Use [14], sponsored by the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research (ASCR), focuses on these newer approaches, with the intent to increase and accelerate research on social, cognitive and AI themes as they can be applied to scientific software. Workshop participants and topics were chosen to create an opportunity for identifying and prioritizing research directions toward the goal of expanding the size and diversity of the scientific software community to effectively include these themes.

Scope: Science-based Methodologies for Scientific Software

The software engineering community has a robust R&D component. Conferences like the International Conference on Software Engineering (ICSE) provide numerous examples of this work. Software community leaders such as Steve McConnell provide even further value by synthesizing the literature into usable and impactful advice and practices. At the same time, little direct emphasis is placed on scientific research software relative to other much larger software domains. Our community certainly benefits tremendously from broader community R&D investments, but we believe there are unique opportunities for the *scientific* software community to contribute to and benefit from R&D efforts in the science of research software [15].

Because we are scientists ourselves, our hope is that we can appreciate, incorporate, and expand the role that science can play in improving the development and use of software for scientific research. This hope is at the core of this workshop. Areas of high priority for the scientific community include:

- Advanced design for software products, tools and applications
- Social and cognitive sciences applied to scientific software development and use
- Integration strategies for new ideas and culture change

Advanced design for software products, tools and applications

Various types of scientific software—including reusable libraries, development tools, and scientific applications—all would benefit from improvements in software architecture, flexibility/extensibility, and user experience. The traditional organic nature of scientific software design and development can often lead to impediments for further progress, as the effort required to further adapt the software for exploring the next scientific problem may become large or complicated, or may risk incorrect execution. Furthermore, user needs are often assumed to be known from previous informal experiences and are seldom considered from a formal and objective perspective.

We expect that advances in design strategies will have a disproportionate positive impact on scientific software because the broader software community has seen substantial gains from increased design emphasis, and the scientific software community has only started to consider advanced design approaches.

Social and cognitive sciences applied to scientific software development and use

One of the most promising approaches to understanding and improving software development and use is to incorporate the social and cognitive sciences into our scientific endeavors. The knowledge, tools, processes, and expertise of these communities have impacted many technology development efforts. Books like *Anthro-vision* by Gillian Tett [16] have popularized the important insight that paying attention to human factors is essential if we want to improve individual, team, and community activities in the pursuit of scientific discovery.

As stated above, there is already substantial R&D activity applying social and cognitive sciences to software activities, but very little of it focuses on scientific communities and the peculiar aspects of developing and using software for scientific research relative to other domains. We look forward to good ideas that can be realized in the future.

Integration strategies for new ideas and culture change

Change is hard—for individuals, teams, organizations, and communities. While we have already experienced a great deal of change in software development practices in recent years, there is still substantial room for further improvement. Can we take advantage of the theory of *Diffusion of Innovations* [17], change management processes, and other approaches for insights into how changes take place in our community, as well as guidance for how we might be more effective in facilitating the changes we seek (need) to create?

Similarly, at a more pragmatic level, are there strategies or tooling that would help lower barriers to the adoption of new processes and approaches in software development and stewardship? Are certain processes synergistic and more beneficial when adopted together? Are there ways to package processes together or layer them to facilitate adoption? What characteristics of a project set the stage for easier adoption of new strategies? What kinds of evidence are useful in helping to convince team members of the value of a new process?

Trends

Importance of high-quality scientific software

Scientific software is being employed by a wider community, to inform policy and decision making. At the same time, scientific software is increasing in complexity due to the demands of next-generation science and new computer architectures. This requires larger collaborative teams (and teams of teams), with a broader diversity of skills and perspectives.

Increased use of scientific software by a wider community. Software plays a large and increasing role in scientific discovery across a wide range of domains and computing platforms. These trends are probably most noticeable in relation to computational and data sciences and the application of data-driven methodologies across many scientific domains. In domains where data has always been a central source of insight, important trends are the emergence of large data sources, and computing tools that can take advantage of these sources. Scientists in these communities are increasingly seeking larger computing platforms, including the leadership systems where work has traditionally focused on modeling and simulation.

Data-driven approaches are also emerging as an alternative or complement to modeling and simulation in scientific domains that have traditionally been based on theoretical formulations. These hybrid theory and data approaches enable rich synergies where data-driven inference can learn from and replace slower simulation components in a multiphysics or multiscale computational environment, or can provide sophisticated interpolants trained from high-fidelity parameter space sampling using model-based simulations.

These trends point to the increased importance of scientific software across many scientific domains, implying even greater importance to improving how we develop and use software for scientific research.

Increased use of scientific computing to inform policy and decision making. As computational science and engineering have matured, predictive capabilities are emerging in many domains, moving beyond traditional interpretive simulations. As a result, numerical modeling is being used with increasing frequency to inform consequential decisions and policy making in government and industry. Examples range from ecological and climate-related policies to safety-critical decisions such as nuclear reactor licensing, aircraft certification, the design of bridges and buildings, and epidemiological modeling.

Increased reliance on computationally based results for important decisions justifies, even demands, increased scrutiny of the software and methods on which decisions are based. Further, practical experience suggests that the path from software intended solely as a research tool to its use in consequential settings is frequently both unanticipated and slippery. These concerns are, in many respects, just an extension of conventional concerns about transparency and reproducibility in science, which apply to computationally-based science as well as experimental and observational science. Consequently, we envision a growing need in the

computational science community to respond to increasing scrutiny and expectations for software quality, reliability, and credibility.

Increased model and software complexity. The difficulties described above are exacerbated by increasing complexity required to address next-generation computational science. Teams are working toward predictive science and engineering through multiphysics, multiscale simulations and analytics and are addressing requirements for greater scientific reproducibility [18,19]. For example, teams are grappling with the increased model and software complexity required to incorporate higher-fidelity models with more physical processes. It is difficult enough to know how to choose a particular solution component, such as an effective linear solver, for a particular problem. The difficulty becomes much greater in composite systems with multiple components, in trying to understand how to quickly reach a solution for a particular problem in various computing environments.

Increased complexity of computer architectures. Disruptive changes in advanced computer architectures also are causing unprecedented software challenges. The transition to hosted accelerated architectures, specifically nodes with multicore CPUs and multiple GPUs, requires developing new algorithmic approaches, porting code to new compiling and runtime environments, and realizing massive concurrency that is possible only by overcoming large latency bottlenecks. Furthermore, these new CPU/GPU platforms represent only the beginning of the system heterogeneity expected in the future [20].

Increased team sizes and teams of teams. The only way to tackle these increasingly difficult problems will be to bring together multiple teams (or 'teams of teams' [21]) in an efficient and scalable manner. There are, in many cases, natural ways to divide the problem space. For example, we have development teams for solvers, others for build and package management systems, others for visualization, and so forth. The approach of well-defined APIs, either callable or file based, has been successful. But we increasingly see very large projects with a number of components to be developed rapidly, and that includes setting the proper mechanisms of interaction (APIs, for example), which cannot be developed in isolation, as the considerations of multiple subteams need to be taken into account, along with the needs of the user community. Agile methodologies have become popular, but these can be problematic for large scientific computing projects, where at the outset one may not know what to code, as research is needed to determine appropriate algorithms, and then further research may be needed to determine the most performant data structures and methods. An outstanding problem is whether there are methodologies for scientific software development that permit rapid progress in research along with early and incremental delivery.

Increased diversity in needed skills. Software is broadly recognized as a primary means of collaboration across disciplines in computational and data-enabled science and engineering, encapsulating expertise in mathematics, statistics, computer science, and core disciplines of scientific and engineering. Indeed, reusable software libraries and tools have a long history of broad impact, and application-specific community codes are becoming widespread as a means of disciplinary collaboration [22]. Given the continually increasing scope and complexity of

collaboration, *software ecosystems* [23] are proving effective, where communities explicitly consider interrelationships among interdependent software products whose development teams have incentives to collaborate to provide aggregate value, where the whole is greater than the sum of its parts. Central to this work are the contributions of *research software engineers* (RSEs) [24], whose expertise in both software engineering and research helps to bridge across disciplines though high-quality software. This broad scope of collaboration also benefits from *project coordinators*, who help software teams to plan work and handle the logistics of coordination. Throughout all of these interactions, experts in social and cognitive science can help us navigate collaborations across aggregate teams, or teams of teams.

History: Progress through newly emerging communities of practice

Addressing these scientific software challenges requires broad community collaboration to change the culture of computational science, increasing the emphasis on high-quality software itself and the people who create it. Responding to these challenges, various grass-roots community groups have arisen in recent years to nurture "communities of practice" [25] in their respective spheres of influence, where like-minded people share information and experiences on effective approaches for creating, sustaining, and collaborating via scientific research software. These groups articulate key issues and needs to stakeholders, agencies, and the broader research community to effect changes in policies, funding, and reward structure, while advancing understanding of the importance of high-quality software in multidisciplinary CSE collaboration and the integrity of computational research [26, 27].

Software Sustainability Institute. An international leader in this topical space is the U.K.'s Software Sustainability Institute (SSI, https://www.software.ac.uk), which has existed for more than a decade for the express purpose of advancing software development and use for scientific research. SSI develops and promotes methodologies for understanding software requirements, building community awareness of the importance of better software, and much more.

NSF SI2, URSSI. The U.S. National Science Foundation has sponsored several programs focused on direct funding for software teams to further develop and support products that have proven broad usability in the scientific computing community, for example Software Infrastructure for Sustained Innovation (SI2) and the more recent Cyberinfrastructure for Sustained Innovation (CSSI). In addition, NSF has sponsored the U.S. Research Software Sustainability Institute (URSSI, https://urssi.us), focused on topics related to the scope of this workshop.

Research Software Engineering Movement. Research Software Engineering (RSE, https://society-rse.org and https://us-rse.org) has emerged as an increasingly recognizable career track, with a growing number of people who consider themselves part of the RSE community. While the definition of RSE varies and other terms have been used to describe this kind of position (e.g., software scientist, scientific programmer), many people in this role increasingly identify themselves as RSEs. Also, the number of workshops and organizations expressly focused on RSE topics is increasing. The growth and institutionalizing of RSE roles as

sustainable career paths are in part driven by the goals of improving developer productivity and software sustainability, as RSE skills are essential to achieving these goals.

IDEAS Productivity Projects. In 2014 the U.S. DOE Office of Science, as a partnership between the Offices of Advanced Scientific Computing Research (ASCR) and Biological and Environmental Research (BER), sponsored the creation of the IDEAS Productivity Project. The project expanded in 2017 in the DOE's Exascale Computing Project (ECP, https://www.exascaleproject.org), which requires intensive development of applications and software technologies while anticipating and adapting to continuous advances in computing architectures. Likewise, in 2019 the BER-funded IDEAS-Watersheds project grew out of the original IDEAS project, with emphasis on accelerating watershed science through a community-driven software ecosystem.

IDEAS (https://ideas-productivity.org) [28], as a collection of projects, represents the first direct DOE funding for developer productivity and software sustainability. The funding and support of program managers Thomas Ndousse-Fetter, David Lesmes and Paul Bayer, as well as ECP leaders Paul Messina and Doug Kothe, opened the door to multi-institutional collaboration to foster improving software quality as a key aspect of advancing scientific productivity, while also building communities of people strongly interested in these topics.

IDEAS has benefited from collaboration with SSI, URSSI and similar efforts. The IDEAS initiative has provided the launching pad for various software quality improvement efforts, including the xSDK math libraries (https://xsdk.info) work to advance collaboration and community policies. The xSDK experience has in turn prompted complementary software development kit (SDK) efforts and broader work on the Extreme-scale Scientific Software Stack (E4S, https://e4s.io). The IDEAS initiative spearheaded the Better Scientific Software site (https://bssw.io), a community-based hub for sharing information on practices, techniques and tools to improve developer productivity and software sustainability. Likewise, IDEAS launched the BSSw Fellowship Program (https://bssw.io/fellowship), to provide recognition and funding to leaders and advocates of high-quality scientific software, beginning with DOE support in 2018 and incorporating NSF sponsorship in 2021. IDEAS also sponsors numerous outreach activities (see https://ideas-productivity.org/events), including the webinar series *Best Practices for HPC Software Developers* [29], the panel series *Strategies for Working Remotely* [30], the *Collegeville Workshop Series on Scientific Software* [31], and other tutorials, BOFs, minisymposia, and events.

Emerging transformative technologies

Ubiquitous collaborative software platforms. The rise of software-as-a-service (SaaS) platforms has included a wide range of platforms supporting software development. Although GitHub and GitLab may be among the most recognized such platforms, numerous others offer a wide range of tools related to software development, the management of development projects and teams, communications tools, etc. Many provide a substantial array of services at little or no direct cost.

The widespread availability of such platforms has lowered the barriers to both collaborative and open software development, and through their adoption by many scientific software teams, greatly enhanced the availability and opportunities for contribution from the community. To the extent that projects on these platforms are publicly accessible, they are more easily found by prospective users and contributors. And whether they are public or private, the barrier to collaboration is lowered simply by the reduction in the number of distinct login credentials users need to maintain. But in the context of this workshop, these factors also result in a great deal more (scientific) software being publicly accessible, which facilitates the study of more scientific software, by mining information from the development artifacts thus exposed, in order to better understand the software and the teams who produce it.

Ubiquitous virtual communication platforms. The number of virtual communication platforms continues to explode. For just messaging, it is now common to move from text messaging to email to Slack to the various texting systems provided by social media. Beyond text communication, we broadly use a variety of collaboration platforms, such as Zoom, Microsoft Teams, Google Hangouts, and more. This plethora of mechanisms means that each member of the community must learn the idiosyncrasies of these many tools. At present these tools are generic, with the ability to share screens, view cameras, and have audio communication. Are there ways to make these tools have extensions specific to (scientific) software development?

AI-assisted tools. AI-assisted tools are now prevalent in our content-creation systems. Most of our word-processing environments now include real-time predictive and corrective spelling and grammar support. Similar tools are emerging for programmers such that a programmer can increasingly focus on intent and get support from a pattern-aware AI system to suggest how to specifically implement that intent.

Advances in social and cognitive sciences. In the software engineering community, the field of user experience (UX) is a growing component of software organizations. While UX has always been a part of the software product development cycle, the degree of sophistication has increased, taking into account a growing knowledge base from the social and cognitive sciences. Furthermore, the size and complexity of our scientific software environments continue to grow, making it more important to consider the usability of software within particular environments.

Opportunities

Enabling the continued advancement of scientific discovery

Accelerating scientific discovery. Computation has long been a critical part of scientific discovery, along with theoretical and experimental investigation. Moreover, computation accelerates those methodologies, allowing the design of complex experiments and providing guidance to theorists in parameter regimes not yet reachable. But scientific discovery occurs only when appropriate software exists. This circumstance implies the need for a wide range of

computational software that is well designed and well documented, considering the broad context of its use.

Improving developer and user efforts. Developer efforts must first start from domain knowledge. For example, in object-oriented computing, we construct software objects, each of which represents a physical object, such as the electromagnetic field or a charged particle. This very construction requires deep domain knowledge. On the other hand, developers often make use of mathematical libraries, for which they may well understand basic concepts, but efficient use is either not well known or not well explained. A simple example is the use of iterative linear algebra libraries. Small changes in preconditioners can make a huge difference in the number of iterations required for convergence. Yet, there are few guidelines for choosing preconditioners for different types of problems.

It is important for developers to realize that to a software user, the time to solution is the total time, including startup time to learn the software, time to set up the problem within the software, time to debug the setup, time to run, and time to analyze the data and/or visualize it. On the other hand, it does not make sense to create an advanced and sophisticated graphical user interface, which might require a team of multiple developers, to set up a run for some aspect of computational software if that software has only a few users.

Enabling more sophisticated models and advanced use cases. The HPC community continues to move towards greater realism, more accurate computing, the inclusion of more physics, and better coupling between physics modules. For example, an accelerator cavity designer needs to transfer data between electromagnetic modeling software and thermal modeling software. This requires knowledge of different data formats, neither of which the designer controls, and an understanding of the data to be transferred. Work of this sort often ends up being a number of one-offs, which is time consuming. Tools to reduce this work are desirable, with said tools being easy to learn so that the work of learning is more than amortized over the number of transformations to encode.

Expanding the scope of scientific skills and disciplines

Leveraging scientific advances in social and cognitive sciences. Focusing on a scientific approach to understanding and improving scientific software development and use creates a natural impetus to engage social and cognitive scientific experts as part of our software community, leveraging their experience and working with them to understand our software requirements. We anticipate that the integration of these new perspectives, tools, and skills will also benefit the larger software community.

Leveraging advances in AI for advanced tools and methodologies. As software developers and users, we regularly observe, analyze, and execute many similar patterns. The more we can capture these patterns as data sets, the more we can apply machine learning algorithms to expose these patterns for future automated and machine-assisted programming and software use. This approach is already used in many settings but has generally not been extended to

important scientific software needs such as Fortran programming, floating-point data, and optimization for performance.

We are already starting to see some of these tools, e.g., GitHub AI programming assistant. We need to understand how these tools will impact scientific software activities. This is both a technical and human factors topic. Based on what we discover, the broader software community can also benefit from what we learn.

Expanding the scientific developer and user communities

Designing higher-level interfaces. Computational science pervades virtually all aspects of our world, including the physical sciences, life sciences, social sciences, and more; many industries already heavily rely on computational technologies to assist with design and manufacturing. However, the effective use of HPC software technologies, even for people whose primary focus is domain-specific science and engineering, tends to require substantial understanding of computer architectures, programming models, and lower-level algorithms and data structures. Enormous potential exists to expand the use of HPC software technologies in industry and decision making. However, a prerequisite is broadening the scope of people who can effectively use these sophisticated tools – requiring the development of higher levels of abstraction and interfaces for non-expert users, while concurrently enabling specialists to customize lower-level choices.

Creating opportunities for new and more diverse scientific community members. Numerous studies have shown that diverse organizations, teams, and communities perform more creatively and effectively—and thus are more productive. While some efforts are already under way to broaden participation in computational science and engineering, our communities could benefit by increasing emphasis on sustainable strategies to advance diversity and inclusion. Work is needed to understand how to improve teaming skills and culture, including leadership of remote, distributed, and hybrid teams, while fully leveraging tools to facilitate distributed work. Equally important is research on building a growth culture in scientific communities, with recognition of the benefits of rich engagement across diverse demographics and areas of expertise.

Understanding how to leverage and adapt knowledge, tools and processes from the broader software development and user communities

Overall, software development and use are fairly well studied by the software engineering research community, and today, perhaps to a lesser extent, by the social and cognitive science research communities. Members of the scientific software community have successfully both adopted and adapted a great deal of knowledge, processes, and tools from the broader software community, but frequently have found ideas that do not translate well, for a wide range of reasons. As a simple example, consider the challenges in applying the Scrum methodology to settings where developers are often neither co-located nor dedicated to the project. But what distinguishes the scientific software development context from more general development is multi-faceted, and itself not well understood.

There are numerous opportunities arising from this situation. First, to understand what distinguishes scientific software development from the more general activity. Second, to better understand (and predict) what knowledge, processes, and tools will or will not translate well into scientific settings. And third, to facilitate the adaptation of general approaches to the scientific software context.

Expanding the impact of scientific approaches

Scientific approaches to learning how to develop scientific software ultimately are of no use without being taken up by the community. It is expected that there will be resistance to changing methods, as there always is. So it is important that from the outset, those studying how to improve methods have thoughts early on about how they would communicate their effectiveness to a wider audience. Naturally, there will be a need for workshops and tutorials, but these are greatly strengthened by showing the effectiveness of new methods in developing non-trivial scientific applications.

Next Steps

Scientific software development and use are typically the means to an end in the pursuit of scientific discovery and advancement. We hope that by emphasizing the potential for impact of scientific approaches to improve scientific software, our community can justify increasing the kinds of skills of scientific software teams (developers, users and other contributors) and the amount of time spent on improving software practices as part of their overall work.

Potential approaches

Approaches that seem attractive include:

- **Partnering with the social and cognitive science communities on methodologies for data gathering and analysis.** The social and cognitive science communities are formally trained in producing scientific understanding by observing and interacting with human systems. While scientific software teams are such systems, and we typically work to make our teams high functioning and improving, our approaches are usually ad hoc and of an engineering nature. If we perceive a need to improve a practice, adopt a new tool, or institute a new workflow, we typically do a quick search for new possibilities, perform an informal evaluation, adopt a new approach, and then move forward. This approach risks missing out on fundamental principles for improvement, and we seldom record why we make particular choices, thereby eliminating the possibility for others to learn from what we have discovered. Partnering with scientific communities who have these skills and experience will help us make better choices and communicate what we learn via publications and other venues.
- **Adapting and adopting knowledge, skills, and tools from the broader software community.** Software engineering R&D investments are substantial in the broader software community. However, some of the assumptions made when gathering requirements or distilling findings to actionable advice are not well suited to scientific software development and use. For example, the members of scientific software teams typically have highly specialized skills, so some assumptions about developer

interchangeability presumed in the broader software community are not feasible for scientific teams. Scientific software teams need to leverage the R&D results from the broader software community but with awareness of what is most promising for our community and how it might need adaptation to be most effective.

- **Iterative and incremental exploration of new ideas.** Another attribute of many scientific software teams is that producing leading-edge science is the team's primary goal. If we are going to be successful in improving the practices of scientific software developers and users, we need to carefully plan how to introduce new practices, processes, and tools. Few scientific software teams can afford to suspend efforts to generate new scientific results, as their funding requires results, and competitiveness with other scientific teams would suffer. Iterative and incremental change enables tuning to achieve a balance of delivering today's results while also improving how tomorrow's results will be produced.

Sample priority research directions

Considering these trends and opportunities, here we introduce a few promising directions of research. A goal of workshop discussions is to consider broad community input to determine priority research directions for the science of research software.

New AI-based tools for improved productivity and sustainability.
AI tools are already improving the quality and reducing the cost and time required to produce effective prose and other written content. Promising research directions for scientific software include exploring which developer tasks are most time consuming and which could be improved by AI tools. While such tools are already under development in the broader software community, creation and integration of these tools for scientific software development and use can be accelerated and adapted to our specific priorities.

Science-based software design methodologies for improved usability and sustainability.
The software development community in general is moving to methods of rapid and incremental delivery. As noted above, there are many unique aspects of scientific software development, including pre-research to understand promising algorithms, data structures, and so forth, given the premium placed on performance in scientific computing. First, what are the usage patterns? Is it the case that scientific computing always follows the steps of computational experiment conception, creating inputs, running, analyzing data, and visualization? A variation on this process is to introduce an optimization loop that wraps the above set of steps with software that selects new simulations. What other patterns are common? Would using AI in such setups improve outcomes? Have such approaches been tried?

Studies should also extend to the users. Looking over a broad range of projects, what are the types of interactions between users and the software? At what point does it make sense for a project to put more effort into usability? Is there a threshold number of users at which such a transition occurs? And is the transition sharp? Or do successful projects ramp up usability over the lifetime of the project? What aspects of usability are addressed first? Is there an endpoint?

Characterizing the attributes of sustainable scientific software. A key concern about software is its *sustainability*. Despite the importance of this concept, it is a rather nebulous term, lacking a clear definition in the context of scientific software. Like in an eye exam, we can often compare and say we think that one software package is more sustainable than another. But in this case, there's no optometrist who can turn those comparisons into a straightforward prescription for sustainable software. Can we develop a working definition of what constitutes sustainable scientific software, without being overly prescriptive? Can we identify the attributes of the software, the development processes, the team dynamics, and other factors that influence the sustainability of a software product? If we want to do a better job of producing sustainable software, it would help to have a clearer idea of the target and how we might get there.

References

[1] M. R. Benioff and E. D. Lazowski, Pitac Co-Chairs, Computational science: Ensuring America's competitiveness: President's Information Technology Advisory Committee. http://vis.cs.brown.edu/docs/pdf/Pitac-2005-CSE.pdf, 2005.

[2] W. Gropp, R. Harrison et al., Future directions for NSF advanced computing infrastructure to support U.S. science and engineering in 2017-2020. National Academies Press, 2016. http://www.nap.edu/catalog/21886/.

[3] M. A. Heroux, G. Allen, et al., Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop Report, Sept 2016. https://www.nitrd.gov/PUBS/CSESSPWorkshopReport.pdf.

[4] S. Hettrick, Research software sustainability. Report on a Knowledge Exchange Workshop, 2016. http://repository.jisc.ac.uk/6332/1/Research_Software_Sustainability_Report_on_KE_Workshop_Feb_2016_FINAL.pdf

[5] H. Johansen, L. C. McInnes, D. Bernholdt, J. Carver, M. Heroux, R. Hornung, P. Jones, B. Lucas, A. Siegel, and T. Ndousse-Fetter, Software productivity for extreme-scale science, 2014. Report on DOE Workshop, Jan 13-14, 2014. https://science.osti.gov/-/media/ascr/pdf/research/cs/Exascale-Workshop/SoftwareProductivityWorkshopReport2014.pdf.

[6] D. Keyes, V. Taylor, et al., National Science Foundation Advisory Committee on CyberInfrastructure, Task Force on Software for Science and Engineering, final report, 2011. http://www.nsf.gov/cise/aci/taskforces/TaskForceReport_Software.pdf.

[7] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, and S. Wild, Applied Mathematics Research for Exascale Computing, LLNL-TR-651000, March 2014. https://pdfs.semanticscholar.org/08c6/485730d302944967cc6c44762deb2c71c080.pdf?_ga=2.89380702.451571458.1583807553-541362651.1583807553.

[8] J. Hack, K. Riley, R. Gerber, K. Antypas, R. Coffey, E. Dart, T. Straatsma, J. Wells, D. Bard, S. Dosanj, I. Monga, M. Papka, L. Rotman., Crosscut Report: Exascale Requirements Reviews, 2017. https://www.osti.gov/servlets/purl/1417653.

[9] B. Chapman, H. Calandra, S. Crivelli, J. Dongarra, J. Hittinger, S. Lathrop, V. Sarkar, E. Stahlberg, J. Vetter, D. Williams, DOE Advanced Scientific Advisory Committee (ASCAC): Workforce Subcommittee Letter, 2014, https://dx.doi.org/10.2172/1222711.

[10] National Science & Technology Council, National Strategic Computing Initiative Update: Pioneering the Future of Computing, 2019, https://www.nitrd.gov/pubs/National-Strategic-Computing-Initiative-Update-2019.pdf

[11] L. Arafune et al., CI Workforce Development Workshop 2020, https://www.rcac.purdue.edu/ciworkforce2020,

[12] National Science & Technology Council, Pioneering the Future Advanced Computing Ecosystem: A Strategic Plan, 2020, https://www.nitrd.gov/pubs/Future-Advanced-Computing-Ecosystem-Strategic-Plan-Nov-2020.pdf

[13] Office of Advanced Cyberinfrastructure, CISE, NSF, Transforming Science Through Cyberinfrastructure: NSF's Blueprint for a National Cyberinfrastructure Ecosystem for Science and Engineering in the 21st Century: Blueprint for Cyberinfrastructure Learning and Workforce Development, Aug 2021, https://www.nsf.gov/cise/oac/vision/blueprint-2019/CI-LWD.pdf

[14] Workshop on the Science of Scientific-Software Development and Use, sponsored by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, Dec 2021, https://www.orau.gov/SSSDU2021.

[15] M. Heroux, Research Software Science: A scientific approach to understanding and improving how we develop and use software for research, Sept 2019, https://bssw.io/blog_posts/research-software-science-a-scientific-approach-to-understanding-and-improving-how-we-develop-and-use-software-for-research.

[16] G. Tett, *Anthro-vision*, Avid Reader Press/Simon & Schuster, 2021, https://www.simonandschuster.com/books/Anthro-Vision/Gillian-Tett/9781982140960.

[17] E. M. Rogers, Diffusion of Innovations, Simon & Schuster, Free Press, New York, 5th Edition, 2003, https://www.simonandschuster.com/books/Diffusion-of-Innovations-5th-Edition/Everett-M-Rogers/9780743222099.

[18] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl, R. Pawlowski, A. P. Randles, D. Reynolds, B. Riviere, U. Rude, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth, Multiphysics simulations: Challenges and opportunities, International Journal of High Performance Computing Applications, 27 (2013), pp. 4–83. Special issue, https://dx.doi.org/10.1177/1094342012468181.

[19] U. Ruede, K. Willcox, L. McInnes, and H. D. Sterck, Research and education in computational science and engineering, SIAM Review, 60 (2018), pp. 707–754. https://dx.doi.org/10.1137/16M1096840.

[20] J. Vetter et al., Extreme heterogeneity: Productive computational science in the era of extreme heterogeneity, 2018, https://doi.org/10.2172/1473756.

[21] B. Hendrickson et al., ASCR@40: Highlights and Impacts of ASCR's Programs, U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, 2020, https://doi.org/10.2172/1631812.

[22] L. C. McInnes, M. A. Heroux, E. W. Draeger, A. Siegel, S. Coghlan, and K. Antypas, How community software ecosystems can unlock the potential of exascale computing. Nature Computational Science, 1 (2021), pp. 92–94, https://doi.org/10.1038/s43588-021-00033-y.

[23] S. Hettrick, A not-so-brief history of Research Software Engineers, SSI blog, 2016, https://www.software.ac.uk/blog/2016-08-17-not-so-brief-history-research-software-engineers-0.

[24] E. M. Raybourn, J. D. Moulton J.D., and A. Hungerford, Scaling Productivity and Innovation on the Path to Exascale with a “Team of Teams” Approach. In: Nah FH., Siau K. (eds) HCI in Business, Government and Organizations. Information Systems and Analytics. HCII 2019. Lecture Notes in Computer Science, vol 11589. Springer, Cham., 2019, https://doi.org/10.1007/978-3-030-22338-0_33.

[25] E. Wenger, Communities of Practice: Learning, Meaning, and Identity, Cambridge University Press, 1998. ISBN 978-0-521-66363-2.

[26] D. S. Katz, Software in Research: Underappreciated and Underrewarded, 2017 eResearch Australasia conference, Oct 2017, https://doi.org/10.6084/m9.figshare.5518933.

[27] D. S. Katz, L. C. McInnes, D. E. Bernholdt, A. C. Mayers, N. P. Chue Hong, J. Duckles, S. Gesing, M. A. Heroux, S. Hettrick, R. C. Jimenez, M. Pierce, B. Weaver, and N. Wilkins-Diehr, Community organizations: Changing the culture in which research software is developed and sustained, special issue of IEEE Computing in Science and Engineering (CiSE) on Accelerating Scientific Discovery with Reusable Software, 21 (2019), pp. 8–24. https://dx.doi.org/10.1109/MCSE.2018.2883051.

[28] M. A. Heroux, L. C. McInnes, D. E. Bernholdt, A. Dubey, E. Gonsiorowski, O. Marques, J. D. Moulton, B. Norris, E. M. Raybourn, S. Balay, R. Bartlett, L. Childers, T. Gamblin, P. Grubel, R. Gupta, R. Hartman-Baker, J. Hill, S. Hudson, C. Junghans, A. Klinvex R. Milewicz, M. C. Miller, H. Nam, J. O’Neal, K. Riley, B. Sims, J. Shuler, B. Smith, L. Vernon, G. Watson, J. Willenbring, and P. Wolfenbarger, Advancing Scientific Productivity through Better Scientific Software: Developer Productivity and Software Sustainability Report, ECP Technical Report ECP-U-RPT-2020-0001, Jan 2020, https://exascaleproject.org/better-scientific-productivity-through-better-scientific-software-the-ideas-report.

[29] O. Marques, D. E. Bernholdt, Best Practices for HPC Software Developers: The first five years of the webinar series, Oct 2021, https://bssw.io/blog_posts/best-practices-for-hpc-software-developers-the-first-five-years-of-the-webinar-series.

[30] E. Raybourn, R. Milewicz, D. Rogers, E. Gonsiorowski, B. Sims, G. Watson, Working Remotely: The Exascale Computing Project (ECP) Panel Series, July 2020. https://bssw.io/blog_posts/working-remotely-the-exascale-computing-project-ecp-panel-series.

[31] K. Beattie, G. Chourdakis, H. Cohoon, V. Dyadechko, N. Fadel, C. Ferenbaugh, R. Jacob, J. Lofstead, R. Milewicz, D. Moulton, J. Moxley, T. Munson, S. Osborn, W. S. Pereira, S. Shende, B. Smith, J. Willenbring, U. M. Yang, S. Yates, S. Knepper, L. C. McInnes and M. Heroux, Software Team Experiences and Challenges: A Report from Day 2 of the 2021 Collegeville Workshop on Scientific Software, Oct 2021, https://bssw.io/blog_posts/software-team-experiences-and-challenges-a-report-from-day-2-of-the-2021-collegeville-workshop-on-scientific-software.

A large multi-disciplinary drug design team's perspective on sustainable computing software for transformative science

Dong H. Ahn, Xiaohua Zhang, Jeffrey Mast, Francesco Di Natale, Dan Kirshner,
Sam Ade Jacobs, Bronis de Supinski, Brian Van Essen, Jonathan E. Allen, Felice C. Lightstone
Lawrence Livermore National Laboratory, Livermore, CA, USA
ahn1,zhang30,mast2,dinatale3,kirshner1,jacobs32, desupinski1,vanessen1,allen99,lightstone1@llnl.gov

In May 2020, we formed a large multidisciplinary team to develop a new highly scalable, end-to-end drug design and screening workflow that could expediently produce or screen potential drug molecules for further clinical testing. Our team brought together multiple subject matter experts at Lawrence Livermore National Laboratory (LLNL): Livermore Big Artificial Neural Network Toolkit (LBANN) researchers to construct high-quality Machine Learning (ML) models using over one billion chemical compounds; ATOM (Accelerating Therapeutics for Opportunities in Medicine) researchers to create new drug molecules with desired properties (e.g., diversity, validity, efficacy, and safety) using LBANN ML models; ConveyorLC researchers to couple the above with their High Performance Computing (HPC) simulations that evaluate the docking properties of the newly generated drug candidate molecules; and researchers from workload/workflow management software such as Flux and Maestro to couple all of these components into a scalable scientific workflow. In this paper, we present our perspective on the science of scientific-software development based on our team's experience in making the most of compute resources at LLNL, one of the world's largest supercomputing centers. We argue for a *composite science workflow* model as a sustainable approach to enable transformative science on next-generation HPC centers that are rapidly becoming much more heterogeneous and dynamic. It will enable multi-disciplinary science teams like ours to create and to run their simulation campaigns in a much more automatic, scalable, portable, expedient and sustainable fashion. Clearly, this approach allowed our team to conduct new and exciting drug design and screening science on HPC with unprecedented high science throughput by reducing unnecessary human intervention in its simulation campaigns, while leveraging compute resources from multiple supercomputers and new types of software resources including on-premises Kubernetes. Our workflow allows a single subject matter expert to handle the simulation campaigns much more efficiently. The simulation campaign at the center of this effort had previously been divided into four modules: protein and ligand preparation; docking simulations; ML inference; and data collection, overseen by four different subject matter experts, respectively. Furthermore, the four experts would work with their various users to coordinate their applications and convert the Input/Output files for them.

1 Challenge

In a composite science workflow architecture, researchers select a set of preexisting workflow-management software components and flexibly blend them with a disparate set of domain-specific application and management software tools so as to create an end-to-end workflow. While this approach is getting increasingly popular for HPC, it is a burgeoning field, and there are key challenges to overcome.

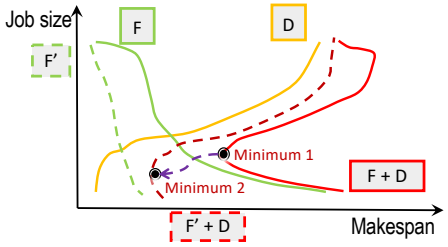


Figure 1: Performance Space

First, while creating a scientific workflow by combining preexisting, production-quality software components can significantly save software development time, we find that achieving high performance from the resulting workflow is particularly challenging. What distinguishes a composite workflow from the conventional approach is that computational tasks are generated, scheduled, and ultimately executed by a set of many domain-specific and general-purpose components in concert. More specifically, the domain-specific workflow managers are closely interacting with a set of general-purpose workflow and HPC workload managers to push the increasingly finer-grained units of work through their work queues and resource allocators at multiple levels. Moving tasks via a pipeline of two different components often involves well-known operations (e.g., generate, submit, schedule, etc) as well as adaptation procedures between these components. A task in a composite workflow must go through compositions of key functions across large numbers of components before it can be allocated to a compute resource and executed at increasingly finer granularity. Thus, the relative performance characteristics of the components over which these key operations are defined are critical for any composition. These unique performance characteristics of a composite science workflow present unprecedented challenges to performance optimization and analysis. High performance can only be achieved by understating complex performance composite functions. For

example, Figure 1 shows how non-linear performance composite functions determined the overall optimality of our team's workflow: where F is the performance curve of Flux-level scheduling, F' is that of its optimized scheduling, D is MPI-based docking domain-specific task scheduler with the coupled ConveyorLC application. This approach demands new performance analysis techniques that can analyze the performance across *all* of the components. Currently, there is a paucity of tools that researchers can use to analyze workflow-level performance.

Second, we find that composite workflows demand new software design and engineering principles which are significantly different from the traditional application design, and two of our specific examples follow. We find that a highly disciplined technique to monitor the performance of the overall workflow is required. A composite workflow almost always uses a large set of *preexisting* components, each of which would generally use different logging and built-in performance monitoring. Yet, we need a consistent way to gather performance data for a particular composition consisting of a disparate set of components to capture the critical-path events. Furthermore, the architecture of a composite workflow must be flexible and highly re-configurable so that researchers can iterate over multi-dimensional optimization across many components for higher performance.

Finally, we find a significant deficiency in how a large HPC center is managed and scheduled. For example, to run our heterogeneous workflow, we must use Dedicated Application Time (DAT) to guarantee the allocation of the compute resources on two different machines at the same time. The molecule batch creation, ConveyorLC Docking jobs, and batch completion are run on a CPU-only supercomputer while ML application jobs are run on a GPU-capable supercomputer, specialized hardware better for ML. The fact that we also couple this with container services run on an on-premises OpenShift Kubernetes machine and plan to couple more components each with preferred specialized hardware (e.g., disaggregated AI accelerators, I/O storage of certain tiers in a multi-tiered storage subsystem etc) will only continue to make this problem increasingly complex.

2 Opportunity

We present opportunities at multiple levels of the key HPC software stack to create and to optimize a composite science workflow like ours more effectively on next-generation HPC centers.

Advancing this approach requires highly disciplined techniques to analyze the performance of the end-to-end workflow. *Workflow-level* tooling techniques to collect, analyze and visualize performance data from many existing components are required. Aspect Oriented Programming (AOP)-based techniques would be a fruitful direction as it can cast a cross-cutting performance-analysis concern or aspect across a heterogeneous set of components in a highly manageable fashion. Second, this approach requires new software engineering principles and techniques to facilitate easy re-configuration and/or re-targetable workflows. Unlike the traditional monolithic workflow approach with a single scheduler of known performance characteristics, a composite workflow's performance optimization is far more complex as it entails multi-dimensional optimization across many components. The resulting architecture must be highly re-configurable to streamline many trial-and-error-based optimization iterations. Furthermore, a manual scheme where researchers go through a workflow optimization iteration will quickly become unwieldy as the number of performance variables are expected to sharply increase. We must explore smarter techniques such as reinforcement learning to aid researchers in determining the optimal choices of performance variables.

Last but not least, this approach increasingly demands system management software-level innovation to enable workflows to break down the barriers between different supercomputers at a HPC center. Next-generation workload management software systems must be co-designed with science teams like ours to schedule heterogeneous sets of tasks on different specialized hardware, as well as services on an on-premises Kubernetes system. The current individual-supercomputer-domain workload management solutions will increasingly impede forward progress to broaden the use of scientific software. These solutions must be capable of scalably managing and scheduling the entire center as a pool of heterogeneous resources and create and assign a dynamically resizable instance of heterogeneous resource set to a composite science workflow. Next-generation workload managers must be able to allow for multi-choice requests (e.g., a job can either be scheduled to 1 GPU-capable node or 10 CPU-only nodes) and use that to find best matching resources per the current resource availability. Further, the workload managers must increase accessibility to end users to enable a specialized workload manager instance to easily blend with domain-specific and other workflow manager components.

3 Timeliness or Maturity

In the past few years, sustainable building blocks for composite workflows have begun to emerge and to be adopted. While component-wise interoperability still remains to be at the native Application Programmming Interface (API) level, common component-wise interface design will be accelerated for better sustainability. Flexible and cloud-computing friendly next-generation workload managers such as Flux are maturing and gaining traction at high-end computing sites. Our described opportunities will only be realized through broad multi-disciplinary efforts. Thankfully, major science workflow initiatives across the Department of Energy (DOE) complex are becoming increasingly multi-disciplinary and existing relationship and collaboration models are placing vertical co-design in our community's reach as described in the Opportunity section above.

Specifying and Debugging Data Distributions at Scale

Elliott Slaughter, Alex Aiken (SLAC) and Pat McCormick (LANL)

The Challenge Distributed data is a central concern in modern scientific software development, and one of the distinguishing features of such software from other kinds of software. Simulations, and also increasingly data analyses, typically require more data than fits in a single machine's memory. Due to the increasing gap between computational performance and memory/communication bandwidth, data movement costs often dominate computation costs for many such applications. As a result, the data distributions chosen, and the ability to tune those distributions to the application's needs, are essential.

Traditional programming models for distributed and high-performance computing do not provide a global view of data. As a result, any data distributions must be manually specified and debugged, and if it is necessary to alter the data distribution for performance or correctness reasons significant amounts of code may need to be rewritten. There are no widespread, general-purpose tools for either specifying data distributions, or for debugging issues that inevitably occur when things go wrong. New research in these directions would dramatically improve the programmability and debuggability of high-performance software systems, and thus contribute to an increase in productivity.

An example of some problems with data distributions that arise in practice occurs in the combustion chemistry code S3D. For most chemical mechanisms the highest performance distribution strategy is to simply place all simulation data in the GPU framebuffers available on a node, but if the mechanism involves a large number of chemical species this approach will often exceed the framebuffers' capacity. For such reactions it is necessary to have a different distribution in which some data is either kept entirely in system memory (and some computations are carried out on the CPUs) or data is moved as needed to the GPU to stay within memory limits. Without a global view of the data, it is very difficult to support both use cases without considerable extra code complexity.

The Opportunity The ability to specify and debug data distributions go hand in hand. In the current shared-nothing approach to distributed high-performance computing, data is always local to a rank; there is nothing intrinsically tying a local data structure in one rank to one in another. Thus, the first step in enabling higher-level specification and debugging of data distributions is to recognize that the issue is a programming models question as much as it is one of tooling.

A number of programming systems provide the ability to specify data distributions, though none of these techniques have yet entered the mainstream. Examples include High Performance Fortran [1] and Chapel [2], as well as task-based systems PaRSEC [3] and Legion [4]. In other cases, libraries may offer abstractions that operate over MPI, but these are typically domain-specific; e.g., AMReX [5]. Additional research is needed to explore what approaches, if any, can be generalized and made productively available to a broader audience.

The lack of generally available data distribution abstractions has also resulted in a limited ability to debug data distributions. In MPI, there are no data distributions, there are only local data structures involved in communication. However, if data abstraction layers can be developed, these also provide additional opportunities: most notably, the ability to visualize and debug data distributions and their performance implications. Like distributed debugging capabilities currently offered for MPI programs, this would enable a global view of the data in a program and would permit programmers to optimize data and its distribution for optimal performance.

Timeliness Recent developments, particularly in global view [2] and task-based [3,4] programming models, are enabling programmers to reason about data distributions, but are not yet widespread. Such efforts have enabled software technologies like NVIDIA cuNumeric [6], which provides seamless distributed execution of Python NumPy programs to programmers with no experience in high-performance computing. This is possible in large part due to abstractions that enable programmatic discovery and exploitation of data distributions.

While promising, these techniques only scratch the surface of the possibilities that can be achieved when data distributions are fully programmable, debuggable and visualizable. In practice, many HPC codes are optimized by tuning their data distributions. If these techniques can be broadened, generalized, and brought to a wider audience, they will have a dramatic impact on the ability of performance engineers and domain scientists to effectively tune the next generation of scientific software.

References

[1] Loveman, David B. "High performance fortran." IEEE Parallel & Distributed Technology: Systems & Applications 1, no. 1 (1993): 25-42.

[2] Chamberlain, Bradford L., Steven J. Deitz, David Iten, and Sung-Eun Choi. "User-defined distributions and layouts in Chapel: Philosophy and framework." In Proceedings of the USENIX conference on Hot Topics in Parallelism, pp. 12-12. 2010.

[3] Bosilca, George, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. "Parsec: Exploiting heterogeneity to enhance scalability." Computing in Science & Engineering 15, no. 6 (2013): 36-45.

[4] Bauer, Michael, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing locality and independence with logical regions." In SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1-11. IEEE, 2012.

[5] Zhang, Weiqun, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day et al. "AMReX: a framework for block-structured adaptive mesh refinement." Journal of Open Source Software 4, no. 37 (2019): 1370-1370.

[6] Bauer, Michael, and Michael Garland. "Legate NumPy: Accelerated and distributed array computing." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-23. 2019.

Continued Investment to Secure Future-Readiness of Scientific Software

TERRY COJEAN, ADITYA KASHI, and TOBIAS RIBIZEL, Karlsruhe Institute of Technology, Germany
HARTWIG ANZT, Karlsruhe Institute of Technology, Germany and University of Tennessee, Knoxville, USA

Delivering production-level research software requires continued funding to keep track of technological changes from the hardware side. The US Exascale Computing Project required leadership software libraries to redesign and embrace platform portability, either by using a portability layer or a hardware backend model. This brings the ECP software stack into a pole position when moving to new and upcoming hardware, but harvesting the fruits of platform portable software design also requires continued investment.

1 CHALLENGE: THE DEVELOPMENT OF SCIENTIFIC SOFTWARE IS NEVER COMPLETED.

A common misconception in research software engineering is to believe that there is a point when the software is complete and no further investment is needed. This belief completely ignores the rapidly changing hardware landscape in terms of processor architecture, parallelism levels, communication strategies, and programming languages. In fact, almost any software that stops adjusting to a changing environment, new challenges and opportunities starts its decline. This is acceptable only for one-time use software. For software meant to serve the community, the challenge is to allocate resources to continuously keep up with changes in the hardware landscape and programming models, and adapt the software stack to these changes. There are a few exceptions to this rule in terms of software packages that continue to be actively used by the community despite their functional and technological stagnation. However, it is such examples that make the argument for continuous investment in software even stronger: These software packages are so important to the community that they are used despite being technologically and functionally outdated. At the same time, the gap between the aged software and the state-of-the-art software design has grown so large that the community hesitates to modernize the software as this would typically require a complete rewrite. As a result, these software packages continue to live as "red giants," often prone to hard-to-resolve technical problems and usually unable to efficiently leverage the compute power of modern architectures. Though not part of the science community, a perfect example of outdated software that is difficult to maintain, virtually impossible to modernize due to the technological gap, yet indispensable and heavily used are the COBOL-based legacy systems used for unemployment benefit management [Lee 2020]. Similarly, many weather and climate models have been developed more than two decades ago, without undergoing a continuous modernization process, and are now failing to port to modern hardware architectures.

2 WHY THE CURRENT SITUATION IS AN OPPORTUNITY FOR ECP'S SOFTWARE STACK.

Within the US Exascale Computing Project (ECP), several applications and software libraries were identified to play an essential role in society, from improving decision-making to advancing humanity's knowledge of the world and the universe. The US ECP invested a significant amount of money to compensate for technological gaps in the software ecosystem that grew over the last years due to a lack of funding for scientific software development, and to prepare the strategically-relevant software stacks so that they can run on the ECP pre-exascale and exascale systems. At the same time, ECP made the preparation for the upcoming supercomputers a challenge as the target systems are all heterogeneous, different in terms of vendor and processor, and unavailable at the start of the project. The software developers, however, turned this challenge into an advantage: instead of preparing or porting the software to one specific architecture, software developers embraced the concept of platform portability. The idea is to abstract the platform-specific details away from the domain scientist and user of the software and to enable the execution of the software on all hardware platforms included in ECP. Within the ECP effort, one can identify two different approaches to enable cross-platform portability and readiness for future hardware architectures. One approach is the adoption of a hardware portability layer that is devoted to supporting hardware through an abstraction. Popular examples are the Kokkos ([Carter Edwards et al. 2014]) abstraction layer and the Raja ([Beckingsale et al. 2019]) abstraction layer; both are successful in supporting new hardware technologies and providing the users with a unique interface that allows applications to run the same parallel code on very different hardware architectures. The software products that rely on these abstractions have little porting effort and are guaranteed to run on current and future hardware supported by the abstraction.

The second approach is to decouple the library-core functionality from hardware-specific kernels and support the backends for different hardware (e.g., the Ginkgo [Anzt et al. 2020] or SLATE [Gates et al. 2019] software package). From a high-level, one could argue that the second approach takes the first approach and combines the high level algorithms, the hardware abstraction layer, and the hardware-specific kernel support into a single software product. However, unlike the first approach, this supports hand-written optimized kernels for each hardware architecture. In addition, the abstraction and kernel development is focused on a single product allowing for a more hardware-specific kernel design and performance optimization.

Relying on a Portability Layer. Relying on a portability layer removes the burden of platform portability from the library or application developers and allows them to focus exclusively on the development of sophisticated algorithms. This convenience comes at the price of a strong dependency on the portability layer, and moving to another programming model or portability layer is usually extremely difficult. Furthermore, relying on a portability layer naturally implies that the performance of algorithms and applications is determined by the quality and hardware-specific optimization of the portability layer. This performance penalty may not always be insignificant, as portability layers usually have a wide user base,

Authors' addresses: Terry Cojean, terry.cojean@kit.edu; Aditya Kashi, aditya.kashi@kit.edu; Tobias Ribizel, ribizel@kit.edu, Karlsruhe Institute of Technology, Germany; Hartwig Anzt, Karlsruhe Institute of Technology, Germany and University of Tennessee, Knoxville, Knoxville, USA, hanzt@icl.utk.edu.

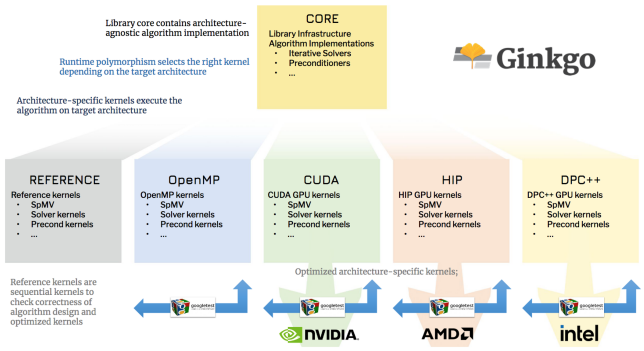


Fig. 1. Backend model used in the Ginkgo numerical linear algebra library.

and dramatic changes to the interface, logic, or kernel design of the portability layer would likely result in the failure of some applications that rely on the portability layer. Hence, performance portability layers should avoid modifying the design or hardware coverage, which can limit the opportunities to heavily optimize kernels for a new hardware architecture. In practice, applications relying on such a portability layer require continuous funding for adopting newer architectures and moving to new technologies. As the portability layer adopts new architectures, the applications need to test this new functionality, parameterize or specialize their kernels and algorithms as needed, and suggest new features to the portability layers to enable better performance. As the ecosystem evolves, so must the applications.

Natively Supporting Various Hardware Backends. Libraries that decouple the core methods from the hardware-specific kernels and support various hardware backends can apply much more aggressive hardware-specific optimization and often achieve higher performance. One reason is that the set of kernels is usually much smaller than what portability layers provide as the hardware-specific backends because only the kernels required by the library's core methods are included. A second reason is that a library has more freedom to phase out support for a specific hardware architecture. This can usually be justified because the dependency on a library is generally much looser than the dependency on a portability layer. To use this model, a library must follow the separation of concerns between the higher-level method and the different hardware-specific backends, see Figure 1 for a visualization of the backend design used in the Ginkgo library. The different backends need to be managed and interacted with thanks to a specific interface layer between numerical methods and kernels. However, the price for the higher performance potential is high: the library developers have to synchronize several hardware backends, monitor and react to changes in compilers, tools, and build systems, and adopt new hardware backends and programming models. The effort of maintaining multiple hardware backends and keeping them synchronized usually results in a significant workload that can easily exceed the developers' resources. On the other end, the freedom and performance it gains can be more cost efficient. Applications and libraries relying on a hardware backend model require continuous funding as a key to their success.

We do not aim to provide general advice on whether relying on a portability layer or natively supporting various backends is the better choice. This decision depends on the scope of the library or application, the tradeoff between performance and platform coverage, and the available resources. In any case, we are convinced that only software packages that provide platform portability and are able to jump to new hardware architectures are sustainable and can play a role in the very diverse world of HPC. Conversely, software packages that do embrace platform portability as a central design principle are not limited to the hardware currently supported. But with continued investment, they are able to extend the scope to emerging hardware and hardware outside the scope of ECP.

3 WHY NOW?

The US Exascale Computing Project has brought the scientific software ecosystem into a pole position for future hardware technologies: The software technology supports platform portability across ECP processor architectures and embraces platform portability as a central design principle, and the software developers are experienced and skilled in enabling portability for new and even yet-unknown architectures. This situation is an asset moving forward, but also a liability as we see a continued proliferation in hardware technology that is waiting for software support: ARM technology is on the rise, a number of machine-learning hardware architectures like Cerebras, SambaNova, Graphcore, Habana, etc. are knocking on the door, neuromorphic and quantum computing are yet to come. The developers of the scientific software stack developed in ECP are eager to explore the potential of these platforms and enable portability. But this requires continued investment. If we stop the strategic funding of scientific software at the end of ECP, we freeze the status quo of software technology, being able to run on current hardware architectures but allowing the technology gap to grow in the next years. The consequences are likely the need for a much larger investment at a later point in time. Providing sustainable funding to scientific software development ensures that the ECP one-time investment turns into a long-lasting strategic advantage in terms of software technology that keeps pace with the hardware developments and secures a leadership position in the scientific software community.

REFERENCES

Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. 2020. Ginkgo: A High Performance Numerical Linear Algebra Library. *Journal of Open Source Software* 5, 52 (Aug. 2020), 2260. https://doi.org/10.21105/joss.02260

David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. 2019. Performance Portable C++ Programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 455–456. https://doi.org/10.1145/3293883.3302577

H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos. *J. Parallel Distrib. Comput.* 74, 12 (Dec. 2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

Alicia Lee. 2020. Wanted urgently: People who know a half century-old computer language so states can process unemployment claims. (2020). https://edition.cnn.com/2020/04/08/business/coronavirus-cobol-programmers-new-jersey-trnd/index.html

Title: An experience-based perspective on scientific software integration.

The authors are currently developing a workflow for tensor-based in situ anomaly detection based on computational fluid dynamics DNS. The task touches three ECP projects, regarding exascale-capable combustion simulation (Pele), in situ visualization (Alpine), and analysis (ExaLearn). Its success depends on the effective communication between end-users (what visual or diagnostic features are useful for that specific scientific community), applied mathematicians (what algorithms can be developed without a human in the loop), computer scientists, and hardware experts.
For two of the authors (Hemanth Kolla and Marco Arienti), the scientific software experience follows a rather common path, starting with the PhD development of a single-user simulation code, and then moving to a more mature, petascale-capable, software in a collaborative setting involving exclusively scientists in the same knowledge domain; gradually, our scientific computing interests have been situated into larger and more diverse groups. The third author (Matt Larsen) comes more directly from a computer science background and has been a key contributor to the development of visualization tools for scientific applications.

- How are scientific software developers and users atypical from other larger software communities?
  The creation of scientific code still tends to be a one-off experience, where programming best practices may not be known at the start of the project. The educational nature of the institutions where scientific code is developed (often by students or postdocs) also contributes to make the establishment of common best practices more difficult. The Labs afford more continuity in setting common programming norms, but only to a degree. Most, if not all, scientific software developers have a primary training in scientific/engineering disciplines, with very few having any formal computer science curriculum. As software have grown large and collaborative, the best practices for software design seem to come from primarily the computer science community. Scientific software developers learn concepts of software design and development mostly on the job.
- What unique changes in scientific software development would improve productivity, accuracy, trust, and/or reliability? Perhaps, it is useful to contrast the development of scientific code with the work of scientists building a new lab. In that situation, even the most advanced instrument comes in a box – complete with instructions, recommended range of use, etc. – and it works (mostly) as advertised. In the scientific computing world, there are sometimes libraries suited to the task, but the approach of writing software is not nearly as modular, with most of the “equipment” to be built from scratch.
- How can we study changes in culture affecting scientific software development and use? In most commercial enterprises, a software architect will be sufficiently knowledgeable to guide the project even without deep knowledge of the company products. Conversely, developing scientific code requires a much deeper domain knowledge. The consequence is that the experience from one scientific code development project cannot easily be distilled into a set of rules that works for a different one. Moreover, scientific software is more continuously evolving than, say, commercial enterprise software (scientific methods and concepts are constantly evolving). There rarely is an end point in the design of a certain

scientific software. Commercial software goes through cycles of development, maturity, stability, and redevelopment. For scientific software these stages are less distinct and more overlapping. Embracing the culture as is from enterprise software development communities is probably not going to work. Finally, in scientific establishments (universities, research labs), there is little to no reward for time/effort spent on writing software (few awarded degrees, or tenure/promotion decisions, give credit for time spent in writing software), and hence there is no incentive. Most software is designed to solve an R&D problem. With ECP that has changed a bit, since ECP was specifically a software development project, not an R&D program. But even here, while ECP acknowledges completed milestones, the parent institutions do not always consider them as genuine accomplishments. Moreover, it is a risk that once ECP ends scientific software will go back to the old culture of serving an R&D need, rather than a mature capability.

- Inter-team communication and team composition. Our experience is that the more homogenous the team, the more difficult it is to maintain an effective exchange of information with the other teams. An example is a group of developers whose members are all based in the same institution, or share the same manager, or share the same education (all physicists, all software engineers, etc.). While this arrangement might look convenient from an organizational point of view, in our view it can become a risk if the project is strongly interconnected. We contrast here two situations from our experience. In the Pele project, the engineers developing the physics package of the code at times find compilation breaks and unreliable behavior due to unexpected changes in the core architecture of the code. In the meanwhile, the architecture developers, who made those changes (in a different Lab, mostly Applied Math majors), complain that they miss actionable information about the application space. Information mostly flows from top (the lead PIs) to bottom (the developers). A different situation is found in the Pele/Ascent/ExaLearn integration: the team is more heterogenous in the sense that it includes an application specialist, the exascale algorithm developer, and one of the main Ascent developers. These individuals report to different managers in different Labs, but they are committed to the same development path in the planning phase. Information is exchanged in a situation where it is necessary to find a common language between different disciplines. This is challenging and possibly time consuming at first, but the day-to-day collaboration that it involved avoids blind spots. Communication from the lead PIs is filtered by the team members and made into actionable items.

In summary, we find that scientific code is developed in conditions that make the establishment and enforcement of best practices tools more difficult than in code development for commercial purposes. The R&D nature of the projects and the discipline-specific knowledge that is necessary to guide a successful development make it more difficult to generalize good practices. With exceptions (such as ECP), the reward system of research institutions does not necessarily enforce the learning and enforcement of software design concepts. With increasing project complexity, the effective communication between development teams is essential, as one would expect, however it is crucial to eliminate language barriers between institutions and disciplines as much as possible. More heterogenous groups, formed across departments and disciplines, have the potential of increasing awareness and peer-to-peer communication.

MLOps: The science of AI/ML software development

Prasanna Balaprakash[1*] (pbalapra@anl.gov, corresponding author), Romain Egele[1], Ewa Deelman[2], Mariam Kiran[3], and Anirban Mandal[4]
[1]Argonne National Laboratory, [2]University of Southern California,
[3]Lawrence Berkeley National Laboratory, and [4]Renaissance Computing Institute (RENCI)

Challenge

Artificial intelligence (AI) and machine learning (ML) are rapidly becoming successful technologies in the modeling of complex phenomena in concert with traditional computational, experimental, and observational approaches. Scientific machine learning (SciML) methods seek to overcome the limitations of the existing heuristics-based, manually intensive, time-consuming scientific discovery processes through large-scale data analytics, pattern matching, surrogate modeling, optimization, and automation. Despite the promising future roadmap, the design and development of SciML software suffer from the very same pitfalls that they are expected to overcome, owing to the challenges posed by the data characteristics and application requirements. In traditional scientific software development, the underlying physical phenomena that need to be modeled and implemented do not change rapidly; therefore the software development, test, validation, and verification, and scaling are often stable and static. This is often not the case in SciML because of the iterative nature of data-driven science, wherein the ML models drive the data collection, which in turn changes the models. Consequently, the ML models need to be constantly retrained, validated, verified, and deployed. This iterative cycle, when performed efficiently, can tremendously increase the effectiveness of the SciML software and can significantly accelerate AI-enabled scientific discovery. MLOps (akin to DevOps in traditional software development) seek to automate, orchestrate, and manage various stages of ML model development and deployment. MLOps apply to the entire SciML software development lifecycle: data collection, data processing, feature engineering, data labelling, model design, model training, optimization, deployment, and monitoring. While the industry has realized the importance of MLOps and started developing the required tools and technologies, the DOE software development community is still in the early stages of understanding the complexity of SciML software lifecycle. Moreover, MLOps software stack in the industry does not reflect scale, diversity, and unique characteristics of the DOE SciML applications. The worrisome issue is the significant knowledge and technological gap with respect to SciML software development lifecycle because no related prior research exists within DOE or elsewhere.

Opportunity

We envision MLOps for SciML software development to enable the rapid design, development, training, and deployment of SciML models. The MLOps will require all best practices in traditional software development such as continuous integration, deployment, testing, validation, and verification. However, SciML demands a new set of components that differ drastically from the traditional scientific software development setting. We believe that there are three scientific advancements required for the MLOps to be effective in the context of DOE applications.

MLOps for Data We envision tools and methods to automate data collection, data processing, feature engineering, and data labelling. The data acquisition schemes for AI/ML models such as design of experiments and active learning are required for systematic data collection. Moreover,

data quality assessment and data augmentation methods should be integrated within MLOps. In several fields, the data is high dimensional and high frequency; therefore, representation learning approaches can be leveraged to reduce the SciML software development complexity. Often, when the model is deployed in production, the distribution of test data becomes different than the training data used for model development. Methods that monitor and detect such distribution changes and shifts are critical to manage the data.

MLOps for Model While data volume and computing power are helping to advance SciML model development, the current educated trial-and-error approaches toward the development of SciML models are rapidly becoming a bottleneck. We envision automated ML methods providing solutions to overcome the pitfalls of expert-driven manual SciML model development and enable the research community to develop models in a systematic and automated way. For example, neural architecture and hyperparameter search methods can be used to find best performing models for the given data set with minimal human intervention. Moreover, automated error analysis can provide insights into the hardness of the data in light of the trained model. Uncertainty quantification methods with the ability to separate the model and data uncertainty can allow software developers to assess the noise in the data and training data coverage, and generalization (out of distribution) errors.

Workflows for MLOps MLOps comprises tasks that are quite complex; automating and orchestrating these tasks using workflows is the only way to solve the complexity of the SciML software lifecycle. These workflows need to execute both data and model MLOps pipelines iteratively in an efficient manner. The workflow for MLOps should cover the entire AI/ML software lifecycle from data collection to model deployment. A challenge in MLOps for DOE applications is the distributed data and computing infrastructure. Specifically, data generation and model training can happen in different locations. Therefore, the workflows need to support MLOps that span the edge to DOE supercomputing facilities. We envision intelligent workflow systems that predict the performance of the MLOps, scale, optimize, and tune the tasks based on the resource availability. We expect that these types of workflows will be able to aid in the test, validation, and verification of tasks in MLOps, which often require scale and high throughput computing.

Timeliness or maturity

Developing MLOps for SciML software development and deployment presents a number of unique research challenges. A key challenge stems from the iterative nature of data-driven science, which leads to data sets that are constantly augmented with data generated from new experimental hypotheses. The DOE ASCR research community is in a unique position to develop MLOps for SciML because it can leverage past successes in developing scalable and efficient data and workflow management software tools for handling massive amounts of data from simulations, experimental facilities, and observational instruments. Nevertheless, there is no science-centric MLOps framework that adopts a systematic approach to relate data, models, and tasks in a scientific domain.

A New Funding Strategy for Scientific Software and an Experiment

Breck Baldwin, Safety 3rd, breckbaldwin@gmail.com

Challenge
The depth, breadth, and impact of open source scientific software exceeded any expectations that one might have had at the start of the millennium. We enjoy high-quality, largely free, state-of-the-art technology all of which came to be due to a nearly magical confluence of cultural changes in software developers, backed by huge technology companies and the development of open source behavioral conventions that coordinate disparate developers towards a common goal. Even Microsoft supports and releases open source software nowadays, inconceivable in 1999 but it started in 2004 (https://en.wikipedia.org/wiki/Microsoft_and_open_source)

Nobody planned this, it just came to be and we are very fortunate to have it. However the heady days of software creation are meeting the realities of middle age which are far less 'fun' and the work has become more like actual work involving maintenance, support, and shoring up all the fast fun work with re-engineered, re-conceptualized versions of what already exists. This is work-work that is best handled by people paid, not volunteers.

A very welcome development is that scientific funding agencies are recognizing that all this software could use some resources to help maintain it, e.g., NASA's recent call for infrastructure funding with the TOPS program. I have been executive director of a major open source software organization, Stan, and if you have not heard of Stan then you hopefully have heard of NumFOCUS of which Stan is a member along with NumPy, SciPy, Julia, and others. This position paper comes from what I thought was a more ideal way to allocate limited resources for scientific software.

So I will argue that funding processes need to evolve as scientific software has evolved. Currently, broadly speaking, reviewers expect novelty in proposals, lean hard on the reputations of the principle investigator/team and then, if awarded, awards less funds than requested. This has problems. Addressing each in turn:

Innovation: Funders/reviewers, even for infrastructure proposals, want something new. In my experience, 'infrastructure' in an RFP (request for proposals) means ‘building new infrastructure’, not 'stay the course'. Bigger, better, faster even if it is not a new idea. This is not what maturing open source projects need. As time advances software needs to be redesigned, perhaps made smaller, documentation needs to be rewritten--all of this is frankly boring but vital non-innovative work.

Reputation: The creators/maintainers of much open source scientific software are relatively junior, not famous, and often don't have strong publication records. Teams of them work from soft money jobs with tremendous responsibility and impact but that is not reflected in easily assessed citations in Nature or Science.

Volatile award amounts: The traditional 10-30% haircut upon award hurts small organizations and any efforts at diversity expressed in the funding application. A 90% paid position is not a position anymore if that is the only source of funding, a diversity initiative will be the first thing cut. There is no 'other source' of funding like teaching fellowships at a University.

Opportunity
The opportunity here revolves around fundamentally changing how funding decisions get made. The 'Intellectual Merit' and corresponding translation into whatever DOE/DARPA/NASA/NIH etc... calls it has to go. In place of innovation, I suggest focusing on anti-innovation by evaluating past and current metrics with the future being "more of the same". In more detail, I suggest metrics like:

1. How much is the software used as quantified by citation count, downloads, forum traffic or breathless letters of support from Nobel laureates?

2. Evidence of an active development community--pull request processing, governance, forums.
3. Evidence that the project knows how to spend money in support of project goals.

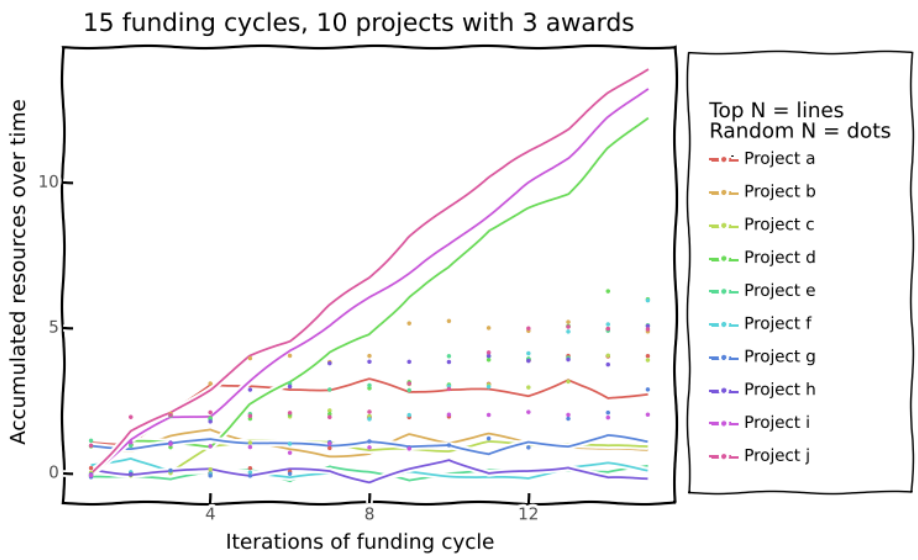
Also, define the merit of the proposal to be well defined/easy to compute hurdles that an organization can aspire to achieve in checklist like fashion-- "Pull request aging less than 10 days on average, check, the average response to forum post 36 hours, check, diversity program in place, check..." Defining such hurdles will be a positive influence on scientific software development if done well, but they must be backed by actual funding decisions.

Stop funding 'Top N' selection: Algorithmic policies impose numerical scores whose goal is to establish the relative rank of all proposals with some wiggle room for program manager judgment. If reviewers/program managers were infallible this would be a great strategy but I question the oracular performance of this arrangement. Any total order selection process reinforces biases, easy quid-pro-quo understandings, and pure reputation-based funded-for-being-funded feedback loops.

Start funding 'Random N' above a threshold: We could send all reviewers off to Baldwin's re-education camp for unbiased evaluation, or... just have reviewers eliminate proposals that seem very unlikely to succeed and fund those that remain. I suggest the awardees be chosen by random draw if there are not sufficient funds. Also, no funding reductions to 'stretch' the budget.

Maturity

Why now? Obviously, federal funders have figured out that infrastructure funding for all this free software might be a good idea. The graph below is a very simple simulation demonstrating the consequences of selection of the top N proposals based on reviewer scores compared to randomly selecting N proposals that are above a threshold. The simulation can be altered and experimented with at: https://blooming-lake-98194.herokuapp.com/.



Resources are reported on the y axis with 1 unit per award, an award increases the reputation of a project by .1 and the score is reputation + constant merit (.2) + random noise. This applies to either selection strategy. The 'Top N' strategy concentrates resources on past awardees given their reputation increment from past funding creating a funding-for-being_funded loop. 'Random N'' strategy, dots, awardees are selected that are above a threshold of .3. There is no advantage being way above .3 or barely above .3. The latter spreads resources much more evenly.

An Experiment

These position papers are supposed to be about the science of scientific software, so I'll suggest an experiment. DOE's next call for proposals should award 50% based on the standard funding process and 50% with a random draw from a pool of competent submissions--scientific software or not. Establish metrics criteria ahead of time and, as much as it pains me as a Bayesian to say, we have a perfect null hypothesis test of the oracle of current practices against the power of randomness.

Cybersecurity in High Performance Software Development: Challenges and Approaches

Oceane Bel*, Cimone Wright-Hamor*, Joseph Manzano*, and Kevin Barker*
Emails: obel@pnnl.gov, cimone.wright-hamor@pnnl.gov, Joseph.Manzano@pnnl.gov and Kevin.Barker@pnnl.gov
*Pacific Northwest National Laboratories

As new computational substrates become more common (e.g., [1]), the development of new software components, ranging from system software to application workflows, is accelerating as well [2]. The community is seeing new efforts in porting previous scientific workflows to the newer programming standards plus introducing new, more "productive", computational environments to help scientists in their search for answers. Alas, the process of software development is riddled with opportunities to introduce performance bugs and security vulnerabilities (e.g., [3]). These issues can trigger cascading effects, such as system usage degradation [4] or leaking sensitive information to possible attackers [5]. Since software development is an unending process of iterative building, it quickly becomes difficult to consistently test a piece of software against amorphous security or performance issues during the development phase. Therefore, whatever techniques that are proposed to aid in the testing of software must be adaptive, accurate, time-efficient, and clear to understand (e.g., as processes detailed under Agile methodologies.)

Washizaki *et al.* [6] describes a taxonomy of over 200 security papers containing security patterns. They define security patterns as "security-related problems and solutions that recur in certain contexts for secure software system development and operations." Several security patterns that develop in the maintenance and evaluation phases of software developments are not well researched, and as such, there is a need for more exploration in pattern classification of the system lifecycle, in defining dynamic behaviors of security patterns, and finding out if these patterns exist in current systems. This is further exacerbated by having different requirements in different domains. For example, the willingness to adapt and to test new technologies under high performance scientific workflows might curtail the use of good software development practices and tools designed to address common security patterns.

Although there exists tools that help with both performance degradation and security vulnerabilities [7], [8], [9], new advances in Machine Learning (ML) plus Descriptive Cognitive Models (DCM) [10] can be used to understand the subtleties of user / code interactions in terms of performance and security. However, the concerted interactions between all these methodologies / technologies have not seen much exposure because of a lack of synergy between performance and security concerns prevalent in scientific circles in the past.

High Performance Software and the Wireless Revolution: One of the most promising novel computing substrates for science workflows is in the area of wireless communications. The introduction of 5G and WifiMax technologies with their promises of low latency, high bandwidth and always-on nature allow the execution, collection, and analysis of scientific workflows across many intermediate components. These components are manifested at the sensor, the cloud or the supercomputer levels. This mode of operation has been dubbed the "digital continuum" [11]. Thus, large set of small devices connected in an elastic and dynamic mode might be used as a proxy to parts of this digital continuum paradigm. Nowadays, these types of networks are common in the domain of the Internet of Things (IoT). IoT plus advancements in ML have greatly contributed to advances in technologies such as self-driving cars, smart electrical grids, and smart cities [12].

However, the expansion of these IoT paradigms to other domains also brings to light new security patterns that have not been previously identified due to the scale of such deployments. Additionally, each IoT device is potentially accessible by any independent software developer, and the execution of unverified software introduces the risk of cascading security vulnerabilities. One IoT vulnerability may escalate into an unmitigated gateway for more insidious exploits.

Researchers [13] have developed systems using ML to counter some of the security threats present in IoT devices. ML enables systems to dynamically change their response when facing new threats caused by the execution of unfamiliar code on the system. Using Neural Networks alongside reconfigurable hardware, systems can adapt and change in response to detected attack patterns. It can do so by changing system parameters or rerouting data to block attackers from accessing sensitive data.

Additionally, ML techniques can be used to stress test a code base against multiple attack patterns that have been previously seen on a system. Researchers [13], [14] have summarized several common attack patterns. These patterns can be used to develop and train models, such as Generative Adversarial Neural Networks (GANs) [15], that can generate similar, but synthetic, attack patterns for the purpose of testing a code base's security. Simulators such as NS3 can be leveraged as a test bed mimicking the network topology that the developer is working on. The current limitation on these approaches is the unknown limits of the GANs. The idea of applying GANs to data generation stems from their already established ability to generate new images from existing image data. However, GANs are computationally expensive and require vast amount of representative data which might not be available currently.

Identifying performance issues: Forecasting demand on a system can prevent issues coming from sub-optimal parameter

settings. Usually, programs have parameters that can be set or toggled depending on the workload the system is experiencing, and these parameters affect the latency/throughput of executing work. The drawback of not knowing what parameters to set for certain situations is that shifts in demand can cause a mismatch between currently set parameters and what should be set. This may cause bottlenecks to occur in the system. It is difficult to determine which values the parameters should be set during development but, using a combination of ML approaches and simulation tools, a developer can stress test their code against a wide variety of demand waves, and prepare sets of parameters to counter bottlenecks.

Another performance issue can arise during code development, where performance degradation is introduced during development. Researchers [16] have identified several performance issues that may arise during code development, and how it impacts the performance of the system in the long run. One of the major issues that can occur is performance regression, which causes response time degradation and increased resource utilization. The authors demonstrate a need for tools to identify these performance regressions during the development phase.

An unoptimized code base may lead to performance repercussion felt across the system. These repercussions could be a lack of available resources since the slow code may hold more resources for a longer period of time, thus depriving other programs of resources until released. DCMs have been used to understand how the users of a system or program will interact with the finished product. Using such a model, a user can determine what sort of interaction with they program can create performance deficits, faulty interactions, or use-errors. These models are especially useful when trying to identify which part of the code could become a bottleneck in the near future. The drawback of such approaches is that they are unable to stress test on a wide variety of workloads. They are limited by whatever the user can simulate which is usually whatever behaviors that are observable on the system they want to run their code base on.

Opportunity: There is a need to stress test code before it is deployed on systems, especially against potential attack patterns. A vulnerable code base could lead to security issues for the system as a whole. Neural networks can be used to find ways to dynamically classify these patterns in the code when integrated with system software toolchains such as LLVM. A driven example can be constructed with Generative adversarial networks (GANs) that generate potential solutions to the patterns that has been identified. Models can then simulate different security threats that the code may face which can be used to identify any security vulnerabilities with the code. Moreover, tools such as Descriptive Cognitive Models can determine how a user may interact with the code base thus allowing us to determine if the code may face performance regression or security vulnerabilities once it is deployed on the system. On the other hand, a model can be tuned to focus on specific vulnerabilities and threats to allow researchers to stress test their code when faced with the adversary they are trying to defend against.

Using all these techniques in a synergistic fashion can help develop more robust software designed to run in the highly dynamic and elastic environments that will become more common in the future scientific landscape.

Timeliness or maturity: The availability of accelerator hardware, such as GPUs and FPGAs, allows access to a vast reservoirs of computational power [17] that can be used to run complex learning models along with other workloads. Moreover, the availability of low latency / high bandwidth wireless networks allows the exploitation of untapped potential residing at the edge. However, this impetus to go farther and farther might prove counterproductive if there is not a concerted effort to identify and to resolve performance degradation bugs and security vulnerabilities. Such systems should encompass both static analysis and dynamic behaviors by stretching the analysis to all parts of the software stack. An advantage of this approach is that the reduction of security issues with deployed code will enable to focus on larger security threats that can happen during live execution of the code, without worrying about any security flaws present in the newly deployed code.

REFERENCES

[1] "Wafer-scale deep learning," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–31, 2019.

[2] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, 2014.

[3] N. Pundir *et al.*, "Secure High-Level Synthesis: Challenges and Solutions," in *2021 22nd Intl. Symposium on Quality Electronic Design (ISQED)*, pp. 164–171, 2021.

[4] D. G. Reichelt and S. Kühne, "Empirical analysis of performance problems on code level," ICPE '16, (New York, NY, USA), p. 117–120, ACM, 2016.

[5] M. A. Obaida *et al.*, "Interactive sensitive data exposure detection through static analysis," in *2017 IEEE International Conference on Electro Information Technology (EIT)*, pp. 270–275, IEEE, 2017.

[6] H. Washizaki *et al.*, "Taxonomy and literature survey of security pattern research," in *2018 IEEE Conference on Application, Information and Network Security (AINS)*, pp. 87–92, IEEE, 2018.

[7] J. Viega *et al.*, "Statically scanning java code: Finding security vulnerabilities," *IEEE software*, vol. 17, no. 5, pp. 68–74, 2000.

[8] J.-E. J. Tevis and J. A. Hamilton, "Methods for the prevention, detection and removal of software security vulnerabilities," in *Proceedings of the 42nd annual Southeast regional conference*, pp. 197–202, 2004.

[9] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.," in *USENIX security symposium*, vol. 14, pp. 18–18, 2005.

[10] C. Silva *et al.*, "Development and validation of a descriptive cognitive model for predicting usability issues in a low-code development platform," *Human Factors*, vol. 63, no. 6, pp. 1012–1032, 2021.

[11] P. Beckman *et al.*, "5G Enabled Energy Innovation: Advanced Wireless Networks for Science, Workshop Report," 3 2020.

[12] T. Rajmohan *et al.*, "Research landscape of patterns and architectures for iot security: A systematic review," in *2020 46th Euromicro Conf. on SW Eng. and Advanced Applications (SEAA)*, pp. 463–470, IEEE, 2020.

[13] I. Idrissi *et al.*, "Iot security with deep learning-based intrusion detection systems: A systematic literature review," in *2020 4th Intl. Conf. On Intelligent Computing in Data Sciences (ICDS)*, pp. 1–10, IEEE, 2020.

[14] P. Ruf, J. Stodt, and C. Reich, "Security threats of a blockchain-based platform for industry ecosystems in the cloud," in *2021 Fifth World Conference on Smart Trends in Systems Security and Sustainability (WorldS4)*, pp. 192–199, IEEE, 2021.

[15] I. Goodfellow *et al.*, "Generative adversarial networks," *Commun. ACM*, vol. 63, p. 139–144, oct 2020.

[16] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *2017 IEEE Intl. Conf. on Software Maintenance and Evolution (ICSME)*, pp. 341–352, IEEE, 2017.

[17] D. Choudhury *et al.*, "High-performance and energy-efficient 3d many-core gpu architecture for accelerating graph analytics," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 1, pp. 1–19, 2021.

Fostering Interoperability and Increasing Scientific Productivity in Environments of Heterogeneous Software and Computational Platforms

E. W. Bethel*, B. Loring, O. Rübel, G. Weber
LBNL
N. Ferrier, J. Insley, V. Mateevitsi, S. Rizzi
ANL
P. O'Leary, U. Ayachit, C. Wetterer-Nelson
Kitware, Inc.
E. Duque, B. Whitlock
Intelligent Light

1 Context and Focus

As has been observed [6], the scientific software development community *"face[s] an urgent need to improve developer productivity, positively impacting product quality, development time, and staffing resources, and software sustainability, reducing the cost of maintaining, sustaining, and evolving software capabilities."* When combined with the increasingly multidisciplinary nature of modern scientific software development [5], we observe that there are numerous challenges and opportunities in the areas of heterogeneous software tools in our ecosystem (§2), the inescapable reality of heterogeneous computational platforms (§3), and the increasingly diverse nature of scientific software communities (§4).

2 Heterogeneity in Scientific Software Tools, Methods, and Use Cases

Challenge. Growing complexity resulting from increasing heterogeneity in software tools and methods. The past decade has witnessed a rapid growth in the diversity of software tools that play important roles in DOE mission science. These include tools for AI/ML, for data analytics, performance measurement and analysis, code coupling, workflow environments, I/O, platform portable programming and more. The rapid growth in tools often results in increased complexity for code developers who must contend with and rectify differences in APIs, data models, execution models, and parallelization strategies. Given that we expect the first exascale systems to be put into service during 2022, we can expect additional unknown challenges will arise due to the nature of those systems.

Opportunity. Lowering the barriers to using heterogeneous tools for data intensive workloads will have broad benefit. The complexity of adopting and using a growing and changing ecosystem of scientific software tools is multi-faceted. Focusing on data-intensive scientific workloads—which in particular can benefit from third-party methods from research programs and industry—, complexity reducing efforts will likely be directed towards simplifying and streamlining the use of diverse software for data-intensive scientific analysis/learning/understanding pipelines where multiple codes and tools are coupled, and run in parallel on DOE HPC platforms.

Timeliness, Maturity, Impact. The SENSEI project [1] is an example of early success where a SENSEI-instrumented code may leverage any number of different parallel endpoints, including user-written Python code [3, 8]. While SENSEI's strengths lie in its embrace of diversity in data-centric tools, the broader community will benefit from a compendium of knowledge that provides exemplars and curated, pre-configured examples of use cases and tool combinations important to DOE science. A potential impact of this type of breakthrough would be an acceleration in development and deployment time, where scientific software developers and users would have a ready-made set of "recipes" or "motifs" from which to quickly implement common processing patterns that involve use of 3rd party tools for AI/ML, for iterative data-intensive pipelines, and bidirectional data movement and execution control. Because all sciences are increasingly data-driven, lowering complexity of using diverse toolsets will have broad impact across DOE mission science. These would represent a set of *"curat[ed] methodologies to improve software practices of individuals and teams"* [6].

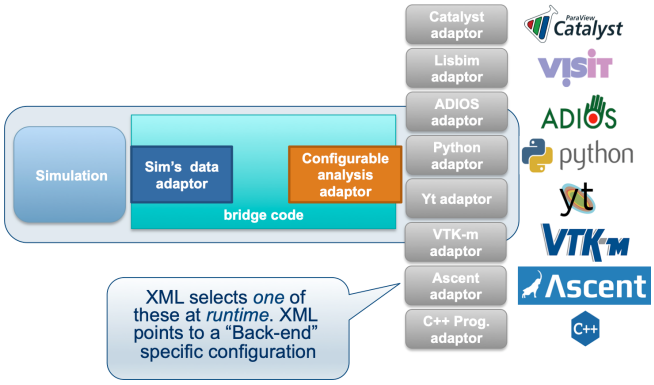


Figure 1: SENSEI's design promotes diversity of computational endpoints, where a user may switch between different endpoints with a runtime configuration file. The endpoints may include existing parallel applications, like VisIt, as well as user-written parallel Python that implements iterative scientific analysis pipelines. Image courtesy B. Loring.

3 Heterogeneous Computational Platforms

Challenge: Use of heterogeneous computational platforms for multistage analytics pipelines is not a straightforward endeavor. While many code teams are leveraging GPU platforms for work in computational and data sciences, numerous issues emerge when considering the more complex task of using heterogeneous computational platforms for multi-stage scientific data-centric pipelines. These include the fact that different stages of the pipeline may not use the same approach for parallelization (domain decomposition), may need to run at different levels of concurrency and/or thread blocking factor due to underlying algorithmic characteristics, and data may need to be repartitioned and moved from producer to consumer as in the case with M-to-N processing pipelines [7]. Furthermore, different stages of these pipelines may be implemented using completely different technologies: some portions may be distributed-memory parallel with MPI, other portions may be targeted at a specific device, e.g, NVIDIA GPUs using CUDA.

Opportunity. Simplify and streamline the use of heterogeneous computational resources for multi-stage scientific data-centric pipelines. Given that the exascale-class platforms will all consist of heterogeneous platforms, efforts that target simplifying the use of

*placeholder corresponding author: ewbethel@lbl.gov

these resources will be of broad benefit across many DOE science programs. Here, "simplifying use of" should be interpreted broadly, but with an emphasis on encouraging the diversity of tools and approaches that are in use now and that will emerge in the future.

Timeliness, Maturity, Impact. Significant effort has been directed towards the problem of code platform portability, with language extension models like SYCL [2] and Kokkos [4]. While these help with individual codes, they don't address the challenges that arise when combining multiple heterogeneous processing stages into a pipeline, where the pipeline stages are diverse and consisting of many potentially different types of code implementations. Advances in awareness and management of data residency and placement is a critical part of the computational landscape needed to support DOE mission science. Other efforts have shown the benefit of careful data partitioning and proximity portability in an M-to-N context [7], ideas that are likely to have applicability and benefit in heterogeneous computational environments. Together, these concepts of platform portability and proximity portability hold promise to be of benefit for data-centric multi-stage scientific pipelines on heterogeneous computational platforms.

4 Unique Characteristics and Challenges of Scientific Software Communities

Challenge. Scientific software development communities have unique characteristics and challenges that require unique approaches. On the one hand, scientific users typically expect that (virtually all) software tools should be open source and accessible free of charge. On the other hand, research software projects are often faced with the reality that identifying and maintaining long-term, stable funding is a significant factor in attracting and maintaining high-quality developers to the project, and to provide them with career development pathways. Related, science code teams may be reluctant to put effort into adopting and using a technology that appears to have doubt about long term funding and stability. Together, these challenges form a "chicken-and-egg" problem for code teams that can be difficult to solve.

Opportunity. Rethink scientific software development and user communities in light of increasing diversity and change. By rethinking how we approach scientific software development and community building, the opportunity is to promote the heterogeneity of ideas, users, tools, developers, and potential funding sources to support and sustain long-term scientific software development projects and scientific software communities.

Timeliness, Maturity, Impact. Building upon ideas from the dawn of the Open Source movement [9], the idea is to foster the growth of a vibrant, open, nimble community of users and developers where users have a vested stake in the long-term health, growth, and evolution of key software infrastructure. While many things have changed since the early days of the Open Source software movement – Spack recipes, HPC/heterogeneous computing platforms, continuous integration, etc. – many similarities exist, most notably, sociological aspects of scientific community development and growth. The hypothesis is that by focusing some attention on the formation and growth of a community of users and contributors, that there will be a deeper sense of ownership and commitment to long-term stability that can provide some ballast against an otherwise state of perpetual change in the technological landscape.

References

[1] SENSEI – Scalable in situ analysis and visualization. `http://www.sensei-insitu.org`, last accessed Nov. 2021.

[2] SYCL 2020 Specification revision 4. `https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf`, last accessed Nov. 2021.

[3] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel. The SENSEI Generic In Situ Interface. In *Proceedings of In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV 2016)*, Nov. 2016. LBNL-1007263. doi: 10.1109/ISAV.2016.13

[4] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12), 7 2014. doi: 10.1016/j.jpdc.2014.07.003

[5] M. Heroux. Research Software Science: A Scientific Approach to Understanding and Improving How We Develop and Use Software for Research, 2019. Online at `https://bssw.io/blog_posts/research-software-science-a-scientific-approach-to-understanding-and-improving-how-we-develop-and-use-software-for-research`, last accessed Nov. 20221.

[6] M. A. Heroux, L. McInnes, D. E. Bernholdt, A. Dubey, E. Gonsiorowski, O. Marques, J. D. Moulton, B. Norris, E. Raybourn, S. Balay, R. A. Bartlett, L. Childers, T. Gamblin, P. Grubel, R. Gupta, R. Hartman-Baker, J. C. Hill, S. Hudson, C. Junghans, A. Klinvex, R. Milewicz, M. Miller, H. Ah Nam, J. O'Neal, K. Riley, B. Sims, J. Schuler, B. F. Smith, L. Vernon, G. R. Watson, J. Willenbring, and P. Wolfenbarger. Advancing scientific productivity through better scientific software: Developer productivity and software sustainability report. Technical report, Oak Ridge National Laboratory, 1 2020. `https://www.osti.gov/biblio/1606662`. doi: 10.2172/1606662

[7] B. Loring, J. Gu, N. Ferrier, S. Rizzi, S. Shudler, J. Kress, J. Logan, M. Wolf, and E. W. Bethel. Improving performance of m-to-n processing and data redistribution in in transit analysis and visualization. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*. Norrköping, Sweden, May 2020.

[8] B. Loring, A. Myers, D. Camp, and E. W. Bethel. Python-based in situ analysis and visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV '18*. ACM Press, 2018. doi: 10.1145/3281464.3281465

[9] E. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Inc., 2001.

Machine Learning to Analyze and Improve the Development of Scientific Software

Abhinav Bhatele[†], Daniel Nichols[†], Aniruddha Marathe[⋆]

[†]Department of Computer Science, University of Maryland
[⋆]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
E-mail: [†]bhatele@cs.umd.edu, [†]dnicho@umd.edu, [⋆]marathe1@llnl.gov

Scientific software in high performance computing (HPC) is often large and complex, comprising of hundreds of thousands of lines of code and utilizing many libraries for different functionality. As a consequence, programmers spend an enormous amount of time and effort in developing, debugging, optimizing and maintaining code. Unfortunately, in spite of the programmer effort involved, there has not been much emphasis on analyzing the development process of scientific software in a scientific manner to learn from and improve processes in the future. The billions of lines of source code that have been written across the HPC community contain implicit knowledge about how to write good code—code that is easy to read and to debug, performant code and portable code. We just need to apply the scientific process to extract this knowledge from it. Recently, there have been efforts in the programming languages/software engineering (PL/SE) and machine learning (ML) communities to find statistical patterns in large corpora of code to drive new software development tools and program analyses [1]. We believe that we can apply these methods to scientific software development and benefit from such work immensely by improving the efficiency of the development and maintenance effort.

Challenge Despite following best practices such as version control, continuous integration and testing, most scientific software development teams do not analyze the development history in a scientific way to improve development practices. Such analysis requires different expertise in programming languages/software engineering that a typical HPC developer may not have. In addition, for software with a long history of development, the data to be analyzed may be large and noisy.

Techniques that can help with ensuring better performance, portability, code quality, and correctness not only help improve the software over time but also help boost programmer productivity both in terms of contributing to and maintaining existing code bases. However, achieving each of these goals is challenging, and requires bringing different communities together to develop a shared understanding the needs and goals of scientific software in HPC. Research that has been demonstrated on code in the industry may not directly translate to traditional HPC software written in C or Fortran + MPI or CUDA.

Opportunity Automated tools for analyzing code have begun to emerge from the PL/SE and ML literature. Novel techniques in code representation learning, which preserve syntactic structural information in the feature space, have been the driving force behind these advancements. These tools and techniques have been successfully used to identify bugs [5], make performance optimizations [4], improve decompilation [8], detect malware [10], offer code suggestions [3], and perform many other tasks in the software development cycle [1]. However, the domain applications of these studies are largely industry-based, focusing on languages, software stacks, and development paradigms which are not necessarily representative of scientific computing. A collaborative effort between the HPC, PL/SE, and ML communities will help build targeted tools for scientific software development, which will improve the correctness, performance, and quality of scientific code, and productivity of the programmer.

Many existing methods can be adapted to scientific code. There is already a significant amount of open-source scientific software online, which includes software development meta-data, such as git commits, testing, and documentation. Some of these techniques may simply need a more focused data set, while others will need to be re-adapted and engineered to fit the paradigms of HPC. Among these methods are a set of ML based tools for improving **programmer productivity.** Certain works [6, 12] define tools for helping developers query for code snippets with natural language. Other solutions include code auto-completion [3] and helping developers overcome language specific development difficulties [7]. Developing code snippet querying and auto-completion tools focused on typical scientific computing code (such as numerical kernels, MPI, GPU kernels, etc...) will greatly improve the productivity of developers as it will reduce the time spent searching for previously written code snippets.

Performance is another critical component of scientific software and is often the focal point. It is desirable to preserve performance as we improve programmer productivity and code quality. ML is already being used at finer

granularities to do code optimizations [4], however, the HPC community needs to begin developing techniques to study how course-grained code changes impact performance. Such work would allow developers to understand the impact of certain lines of code on performance before they even run or profile the software. A recent work which predicts the relative performance of two codes using neural networks [9] shows the promise of this line of work.

There has also been recent work surrounding **code quality** and assessments using ML. Numerous automated code quality tools already exist in the form of linters/formatters. However, [11] takes a novel approach by using code quality assessments from course assignments to predict feedback given by professors. Similar code feedback exists for scientific software within GitHub Issues and Pull Requests. This data contains information about code quality issues, which are too abstract for a linter to detect. Using this data in a similar manner to [11] will allow developers to utilize the vast amount of existing code feedback.

Finally, paramount to scientific applications is **correctness.** There have been many recent studies on using ML to detect possibly incorrect code. Zhu et al [13] use encoder architectures to automatically perform code repair and fix potential bugs. Another work [2] proposes a neural network based method for correcting syntax errors. Other works have also looked at identifying incorrect variable usage and memory access. The HPC community can adopt these to ensure correctness is maintained during the software development process.

Altogether, many techniques have been proposed to address programmer productivity, and software performance, quality, and correctness using ML. The scientific software community needs to adapt and extend these methods to improve HPC software moving forward. This also presents the opportunity for the community to define desirable metrics and objectives in each of these categories: productivity, performance, quality, and correctness.

Timeliness Best software practices have been the source of continuous study, but up until recently, have largely been human driven. The explosion of machine learning and, in particular, recent advancements in language modeling have opened up the possibility to automate the study of software engineering and develop automated development tools [1].

Using state-of-the-art language modeling tools there have been numerous advancements in code representation learning in the last five years. Several dense embedding algorithms for source code, intermediate representations, and abstract syntax trees (ASTs) have increased the feasibility of using advanced machine learning techniques on source code data. Additionally, the innovation in graph neural networks in the past four years enables the direct study of syntactic structure, such as ASTs. These recent advancements open up a wide range of research which can be conducted directly on source code data. Additionally, the ever-growing amount of open source scientific code online with detailed version control history further contributes to the amount of data which can be studied. Hence, it is timely that the HPC community is actively thinking about improving the development process of scientific software and we believe that machine learning can be a powerful tool to help achieve our goals.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[2] S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70, 2018.

[3] M. Chen and et al. Evaluating large language models trained on code, 2021.

[4] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, 18–24 Jul 2021.

[5] R. Gupta, A. Kanade, and S. Shevade. Neural attribution for semantic bug-localization in student programs. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[6] J. Huang, D. Tang, L. Shou, M. Gong (YIMING), K. Xu, D. Jiang, M. Zhou, and N. Duan. Cosqa: 20,000+ web queries for code search and question answering. In *ACL-IJCNLP 2021*, May 2021.

[7] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios. Type4py: Deep similarity learning-based type inference for python. *ArXiv*, abs/2101.04470, 2021.

[8] V. Nitin, A. Saieva, B. Ray, and G. E. Kaiser. Direct : A transformer-based model for decompiled identifier renaming. In *NLP4PROG*, 2021.

[9] N. Pinnow, T. Ramadan, T. Z. Islam, C. Phelps, and J. J. Thiagarajan. Comparative code structure analysis using deep learning for performance prediction, 2021.

[10] J. Senanayake, H. Kalutarage, and M. O. Al-Kadri. Android mobile malware detection using machine learning: A systematic review. *Electronics*, 10(13), 2021.

[11] M. Wu, N. D. Goodman, C. Piech, and C. Finn. Prototransformer: A meta-learning approach to providing student feedback. *ArXiv*, abs/2107.14035, 2021.

[12] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 344–354, 2020.

[13] Q. Zhu, Z. Sun, Y. an Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

The challenges of scientific software sustainability

Aaron S. Brewster, Dorothee Liebschner, Billy K. Poon, Nicholas K. Sauter, and Paul D. Adams
asbrewster@lbl.gov, Molecular Biophysics and Integrated Bioimaging Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA

Challenge

Well sustained scientific software is reliable throughout its useful lifespan, which can extend far beyond its initial funding period. Creating sustainable software requires effort in the following areas:

- *Usability*: software must be easily usable with clear command line and/or graphical user interfaces. Clear design may require surveys, focusing group testing, and interface design expertise.
- *Maintenance*: code needs continual maintenance to optimally function: refactoring, removal of unused code, fixing bugs, creation of clear interfaces between components and encapsulation. Developers need to stay up-to-date with modern development methods and practices.
- *Regression testing*: Tests exercise individual code components and entire applications. Writing tests should be part of the normal development process.
- *Continuous integration*: use of automated tools to test code on regular intervals and whenever new code updates are made. Reports should be reviewed regularly by administrators.
- FAIR (Findability, Accessibility, Interoperability, and Reusability)[1] data management: adherence to international standards should be built into software management plans and be adhered. This may require participation in international committees, studying file formats, or implementing current practices in data indexing and searching.
- *Documentation*: For users, the software should be clearly explained (online documentation, help messages, tooltips). For developers, code should be well commented and readable.
- *Training*: teaching scientists how to use the software via workshops, tutorials, videos, lectures, or printed materials.
- *Outreach*: dissemination of the software through publications, mailing lists, and forums. Particular attention should be devoted to equity and diversity.

Sustainable scientific software can be produced through government grants that include such efforts in their scope (which is rare[2]), through corporate sponsors, or most commonly, through volunteer efforts.[3*] We note that volunteer efforts may lead to burnout and can hinder academic progress, since they are undervalued by funding agencies. Indeed, scientists have immense time pressures from their research activities, which include designing and executing experiments, securing funding, and publishing findings. Many scientists will find themselves without sufficient expertise in modern software engineering practices to sustain the functionality of their software. Therefore, the common challenge for scientists striving for software sustainability is receiving training in these areas. For example, modern continuous integration systems have emerged quite recently and are typically unused by scientists because it is time-consuming to learn these systems, to set them up and to monitor their operation.

Therefore, proper incentives (both academic and professional) are vital to motivate scientists to pursue training to create sustainable software. Furthermore, metrics are needed to track success of training.

Opportunity

Two major opportunities arise from these challenges. 1) social sciences could be used to investigate why scientists struggle developing software in a sustainable manner. For example, it would be interesting to find out why scientists neglect usability principles, what challenges exist that prevent usage or adoption of the software in a way that promotes equity and diversity in their field, or why FAIR data management principles are often ignored (while a seemingly trivial problem, this may lead to devastating loss of scientific data). Social scientists could address these questions with both quantitative (surveys, questionnaires) and qualitative methods (interviews, case studies and focus groups). 2) Investigate what kind of incentives can be used to motivate scientists to apply sustainability principles. Applying these incentives, the DOE could provide the training that scientific developers need to create sustainable software. Finally, metrics for long-term software sustainability can be built into funding mechanisms to ensure that the software is reliable throughout its useful lifespan.

How to find incentives, how to organize training, and how to formulate appropriate metrics for software sustainability could be the focus of several workshop sessions.

Timeliness

Addressing sustainability of scientific software cannot be more timely. As government and academic salaries continue to be outpaced by industry standards, volunteer maintenance of vital open source projects cannot be justified. Further, industry veterans with the requisite software training lack justification to switch into academic research positions and bring that experience into research groups. Retaining scientists in the field of software development requires adoption of sustainable software development practices, through training, incentives, and metrics.

References

1. Wilkinson, M., Dumontier, M., Aalbersberg, I. *et al.* The FAIR Guiding Principles for scientific data management and stewardship. *Sci Data* **3**, 160018 (2016).
2. As an example, R24 Grant Number: R24GM141254, "Phenix: providing high quality software to the research community for crystallography and cryo-EM"
3. *(this reference is of especial significance) Why Is NumPy Only Now Getting Funded? blog post Jul 2017, https://numfocus.org/blog/why-is-numpy-only-now-getting-funded

SCIENTIFIC-SOFTWARE DEVELOPMENT AND LIFETIME

J. Candy (primary point of contact, candy@fusion.gat.com), F. Halpern (federico.halpern@ga.com), M. Kostuk (kostukm@fusion.gat.com), and D. Schissel (schissel@fusion.gat.com)
General Atomics, San Diego, CA

INTRODUCTION

Fusion Theory and Simulation — General Atomics (GA) has been an international leader in magnetic fusion research since the late 1950s and operates DIII-D, a Department of Energy (DOE), Office of Science user facility. GA conducts plasma theory and simulation research in support of DOE's goal of advancing fundamental understanding of plasmas. A tradition of analytic theory is combined with large-scale numerical simulations to understand key physical effects and to validate simulations against experimental data. GA has extensive in-house high-performance computing (HPC) experience and collaborates with the wider HPC community through the SciDAC centers.

CGYRO and ALMA — Two new application codes, CGYRO and ALMA, were developed at GA specifically for modern peta and exascale systems. CGYRO is a rewrite of the legacy GYRO code (in use from 2002 to 2016) using a revised data distribution scheme together with numerical algorithms suitable for acceleration on both multicore NUMA nodes and hybrid central processing unit (CPU)-graphic processing unit (GPU) nodes. Since the older GYRO code was designed in the era of message passing interface (MPI)-only systems, it was deemed more time-effective to rewrite rather than adapt for operation with GPUs. ALMA is a solver for 3D hyperbolic systems, such as the Navier Stokes and magentohydrodynamics models. It was developed from scratch to achieve extreme scalability and performance on leadership architectures. Both CGYRO and ALMA are highly scalable on Summit and Cori KNL, and are expected to perform well on Frontier and Perlmutter. Compared to past experience, GA found that a significantly increased development effort was needed to take full advantage of new hardware generations. The development strategy for both codes involved early integration between the analytical formulation and computational implementation, such that a deep understanding of the hardware was used to motivate code design.

CHALLENGE

Golden Age of CPU Computing — HPC enjoyed a *golden age* of homogeneous CPU architectures with pure MPI for workload distribution. Because the hardware and software ecosystem changed slowly, application developers could concentrate on production runs supporting scientific research, while enjoying incrementally increased application performance through straightforward improvements in CPUs, memory, and networks. The situation changed dramatically with the introduction of GPU and multi-core systems, which favor different mathematical algorithms, memory management, and programming paradigms such as off-loading. Developers have been forced to cater to a fragmented ecosystem because none of the existing platforms have become dominant.

Compute-ecosystem Turnover Time — Remarkably, even with a thoughtful strategy and a blank slate, the application development time has become comparable to the compute-ecosystem turnover time. For example, since ALMA and CGYRO development started, several major versions of the CUDA toolkit were released, and the GPUDirect MPI paradigm for GPU-GPU communication emerged. OpenACC was originally widespread, but is now supported by only one vendor. OpenMP extensions for GPUs became commonplace, but the optimal compute strategy in GPUs and multi-core are different. The older Titan system used low-memory K20 GPUs which did not allow large data objects to remain resident in GPU memory. Around the same time, Cori Haswell and KNL partitions were operational at the National

energy Research Scientific Computing Center and required different optimization strategies. Even as Cori faces retirement, selected mission-critical science application codes were never able to achieve good performance on KNL. With the introduction of Summit in 2019, the availability of large-memory V100 GPUs forced a new cycle of code optimization that allowed large objects (i.e., static arrays) to remain resident on GPUs. Access to Perlmutter has just begun, and Frontier will come online soon. Perlmutter will have minor changes, whereas Frontier will have major hardware changes and different programming requirements than Summit.

Standard Libraries — A widespread strategy to facilitate porting to new architectures is the use of prepackaged, standard mathematical libraries specifically optimized for each hardware. This is most advantageous when dealing with FFTs and BLAS, which have well-established, stable interfaces. However, the developers of these packages, and particularly derivative packages (i.e., sparse LA) suffer from the same obstacles as the application developers. Without a predictable direction for the HPC ecosystem, optimized math libraries become available near the end of each hardware cycle. This has two effects: first, a missing library constitutes a roadblock for application deployment. Second, once available, that library becomes useful for a short time. The overall effect is to decrease the return on investment for both library and application developers.

Commercial versus HPC Software — GA scientists interact with commercial software on a daily basis, for which there is an expectation of functionality, stability, and portability. From this have emerged various schools of practice, such as agile and test-driven development, which extend the philosophy of object-oriented programming to include the code writing process itself for the benefit of maintenance and enhancement. Many of these best practices in modern software design are slow to permeate HPC computing. For example, test-driven development involves defining interfaces and desired functionality though the writing of tests before or in parallel with the actual code, and by placing stability and human-readability as a priority above execution performance. Contrast this commercial approach with HPC software development, which does not typically incorporate these approaches and is more fragile than commercial end user-based applications.

OPPORTUNITY AND TIMELINESS

Addressing Challenges — A more serious effort to stabilize the appearance of the hardware with respect to software and code design would reduce the burden on scientists and developers. The Exascale Computing Project constituted a step forward towards addressing these issues, in particular those faced by industry HPC developers. More broadly, innovative techniques to lower the human cost required to achieve good performance on modern systems would be welcome. Is it feasible to ensure the longevity of persistent software standards (MPI, OpenMP, OpenAcc, OpenCL, CUDA), or is it preferrable to instead move to a new programming model (Kokkos) where distribution and accelerator directives are replaced by novel solver libraries? GA believes the answer to this question should be viewed holistically, which weighs not only the cost of new hardware, but the time-cost of the scientific developer who must adapt to the rapidly changing computing landscape.

REFERENCES

CGYRO: J. Candy et al., "Multiscale-optimized plasma turbulence simulation on petascale architectures", Computers and Fluids 188 (2019) 125–135 [doi:10.1016/j.compfluid.2019.04.016]

ALMA: F.D. Halpern et al., "Simulations of plasmas and fluids using anti-symmetric models", J. Comput. Phys. 445 (2021) 110631 [doi:10.1016/j.jcp.2021.110631]

Anita Carleton, CMU SEI Software Solutions Division Director

Excerpted from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=741193

Architecting the Future of Software Engineering: A National Agenda for Software Engineering Research and Development

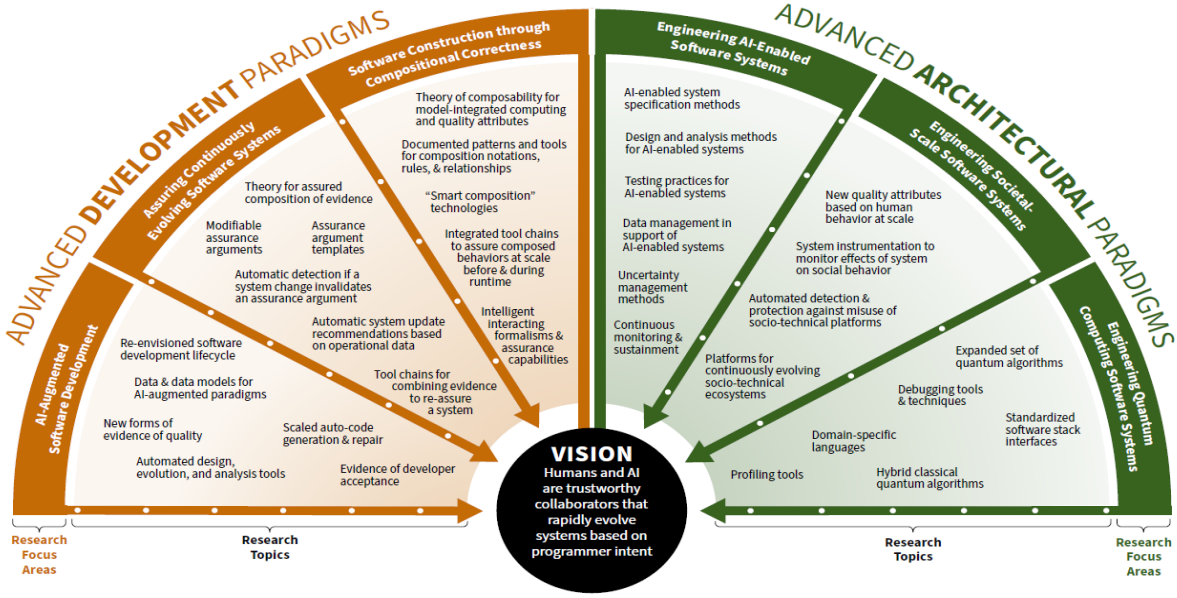
Software Engineering as a Strategic Advantage

We live in an age of software-enabled transformation. Software, and all the software engineering processes, practices, technologies, and the scientific domains that support it, makes our world-class healthcare, defense, commerce, communication, education, and energy systems possible. It is also a key enabling component in nearly every area of research, such as smart infrastructure (nanotech), human augmentation (biotech), and autonomous transportation. Our dependence on software, however, makes us vulnerable to its weaknesses. Software weaknesses are a direct reflection of inadequacies in the state of the art and practice of software engineering, and they can affect millions of people without warning. Without a catalyst for investing in software engineering, the situation will worsen as we increasingly depend on ever larger and more complex software- reliant systems. This report is intended to be such a catalyst. Identifying the critical technologies and areas of research that will enable future systems and laying out a roadmap to guide research efforts is a crucial step toward making software a competitive advantage. This study outlines efforts intended to make future software systems safe, predictable, and evolvable. The Carnegie Mellon University Software Engineering Institute (CMU SEI) engaged the software engineering community and assembled an advisory board of visionaries and senior thought leaders to ensure that the views of the broad software engineering ecosystem were represented in this multi-year research and development vision and roadmap.

A Guiding Vision and Roadmap for the Future of Software Engineering

Our guiding vision, as described in our findings, is one in which the current notion of the software development pipeline is replaced by one where humans and software are trustworthy collaborators that rapidly evolve systems based on programmer intent. To achieve this vision, we anticipate the need for new development and architectural paradigms for engineering future systems.

Our study helped to inform new areas of research that must be met to advance software engineering for future systems. In close collaboration with our advisory board and other leaders in the software engineering research community, we developed a research roadmap with six research focus areas. The following figure shows those areas along with a list of research topics to undertake, and then short descriptions of each of the research focus areas follow.



[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

AI-Augmented Software Development. At almost every stage of the software development process, AI holds the promise of assisting humans. By relieving humans of tedious tasks, they will be better able to focus on tasks that require the creativity and innovation that only humans can provide. To reach this important goal, we need to re-envision the entire software development process with increased AI and automation tool support for developers. A key challenge will be taking advantage of the data generated throughout the lifecycle. The focus of this research area is on what AI-augmented software development will look like at each stage of the development process and during continuous evolution, where AI will be particularly useful in taking on routine tasks.

Assuring Continuously Evolving Software Systems. When we consider the software-reliant systems of today, we see that they are not static (or even infrequently updated) engineering artifacts. Instead, they are fluid—meaning that they are expected to undergo almost continuous updates and improvements and be shown to still work. The goal of this research area is, therefore, to develop a theory and practice of rapid and assured software evolution that enables efficient and bounded re-assurance of continuously evolving systems.

Software Construction through Compositional Correctness. As the scope and scale of software-reliant systems continues to grow and change continuously, the complexity of these systems makes it unrealistic for any one person or group to understand the entire system. It is therefore necessary to integrate (and continually re-integrate) software-reliant systems using technologies and platforms that support the composition of modular components. This is particularly difficult since many of such components are reused from existing elements that were not designed to be integrated or evolved together. The goal of this research area is to create methods and tools that enable the specification and enforcement of composition rules that allow (1) the creation of required behaviors (both functionality and quality attributes) and (2) the assurance of these behaviors.

Engineering AI-Enabled Software Systems. AI-enabled systems, which are software- reliant systems that include AI and non-AI components, have some inherently different characteristics than those without AI. However, AI-enabled systems are, above all, a type of software system. These systems share many parallels with the development and sustainment of more conventional software-reliant systems. This research area focuses on exploring which existing software engineering practices can reliably support the development of AI systems, as well as identifying and augmenting software engineering techniques for the specification, design, architecture, analysis, deployment, and sustainment of systems with AI components.

Engineering Socio-Technical Systems. Societal-scale software systems, such as today's commercial social media systems, are designed to keep users engaged and often to influence them. A key challenge in engineering societal-scale systems is predicting outcomes of the socially inspired quality attributes that arise when humans are integral components of the system. The goal is to leverage insights from the social sciences to build and evolve societal-scale software systems that consider these attributes.

Engineering Quantum Computing Software Systems. Advances in software engineering for quantum are as important as the hardware advances. The goals of this research area are to first enable current quantum computers to be programmed more easily and reliably, and then enable increasing abstraction as larger, fully fault-tolerant quantum computing systems become available. A key challenge is to, eventually, fully integrate these types of systems into a unified classical and quantum software development lifecycle.

Architecting Future Systems Requires Software Engineering Advances

Due to the conceptual nature of software, it continues to grow, without bounds, in capability, complexity, and interconnection. There seems to be no plateau in the advancement of software. To make future software systems safe, predictable, and evolvable, the software engineering community—with sufficient investment from private and public sources—must work together to advance the theory and practice of software engineering strategically to enable the next generation of software- reliant systems.

[DISTRIBUTION STATEMENT A] This material has been approved for public release
and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Explaining the Value of Research Software

Leslie Carr, University of Southampton, UK. lac@soton.ac.uk [communicating author]
Simon Hettrick, University of Southampton, UK. s.hettrick@soton.ac.uk
Heather Packer, University of Southampton, UK. hp3@ecs.soton.ac.uk

1 Challenge

The academic CV or research track record is a well-developed research narrative drawing on supporting evidence from analytical services that allows researchers to explain their work in terms of research publications, citations, funding obtained, impact delivered and reputational achievements.

Research software lacks the data, analytical services and associated narrative templates that describe the contributions that a research software author has made to science, and the leadership that they show within the scientific community. This has been particularly noticeable in a software-sceptic academy that has traditionally treated "software development" as a hidden or commodity activity whose worth does not compare with other research outputs: books, journal articles or conference papers.

The United Kingdom Research Assessment Framework (REF), a periodic assessment of all University research by the UK higher education funding bodies, explicitly allows software submissions to evidence the quality of an institution's research. However, software accounted for only 38 out of the total 190962 submissions (0.02%) to REF 2014 [4]. Part of the reason for this "software hesitancy" may include the delay of developing guidance for adequately describing (and assessing) software submissions [5], compared with the decades of experience in making arguments about the importance of publications.

2 Opportunity

Considering the question "How can we **motivate changes** in the scientific-computing community to improve practices for software development and use?", we believe an important issue is improving the scientific community's understanding of, and acceptance of, the value of software. In order to improve research software, we need to support researchers' efforts to describe the value of their software contributions and their value as practitioners and software contributors to other research stakeholders (promotion panels, research managers, funders, evaluators). We refer to in-depth, evidence-backed narrative descriptions as "research software biographies".

As open science agendas have increasingly aligned with research software (*e.g.* [UNESCO 3]) and research software practice has increasingly adopted open source community platforms (*e.g.* GitHub), research software has become more open to examination, description and analysis. An increasing number of studies of research software have been conducted [1, 2, 3, 6] in an attempt to understand some basic principles of the field.

We propose that the community develops (a) an evidence base of research software ecology enabling (b) analytical methods supporting (c) research software biographies. An evidence base would join existing information sources – software in GitHub, publications in Crossref, researchers in ORCID, projects from funder databases. Built on this data layer, search services, simple statistical comparisons and various analytical methods of software construction or network centrality could provide evidence of research significance, rigour, originality and innovation that can feed into a "data storytelling" process for research software biographies.



Figure 1: Research Software Biographies

Such an activity would necessarily be collaborative and distributed across numerous organisations, as research software is diverse and difficult to identify without national, discipline, and community-specific knowledge and insight. (Although very large numbers of research software repositories can be identified in GitHub, the practice of attributing research software to its related research or *vice versa* is still by no means universal.) Similarly, the kinds of analyses, narratives and values that are relevant to each discipline or area of science are not standardised.

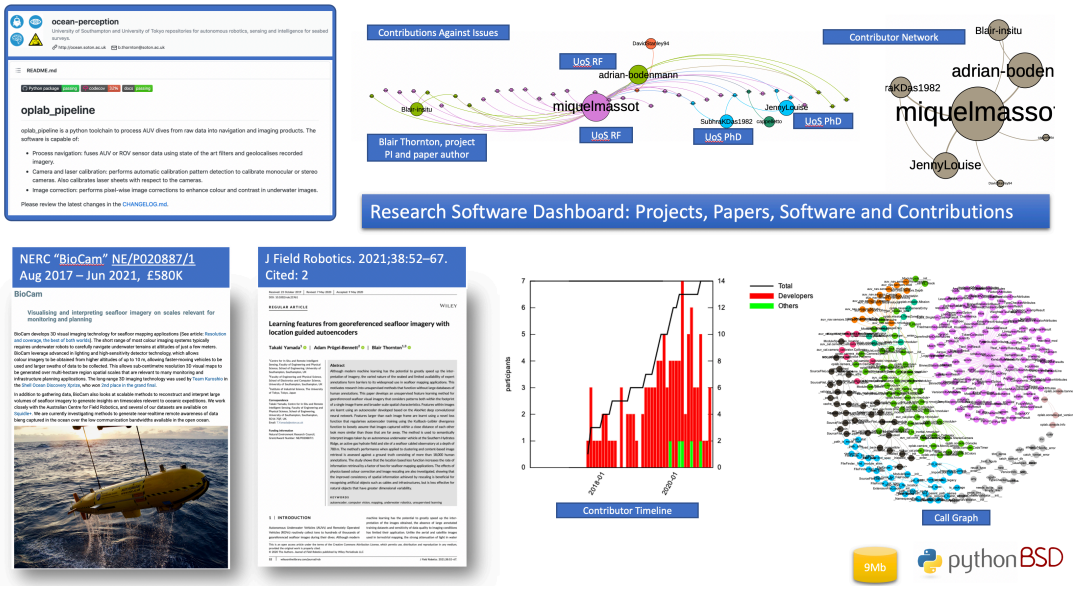


Figure 2: Example of a Simple Research Software Dashboard combining a variety of evidence about a GitHub repository

3 Timeliness or maturity

In recent years, software development practice has evolved such that open source platforms are incredibly popular for any substantive software development activity. In addition, open science is more actively engaging with software. The alignment of these two phenomena make it feasible to develop and leverage an open knowledge base of research software to the community's benefit.

4 References

[1] Färber, M. (2020). Analyzing the GitHub Repositories of Research Papers. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020 (JCDL '20)*. ACM, New York, NY, USA, 491–492. DOI:https://doi.org/10.1145/3383583.3398578

[2] Gonzalez, D., Zimmermann, T. and Nagappan, N. (2020) The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. ACM, New York, NY, USA, 431–442. DOI: https://doi.org/10.1145/3379597.3387473

[3] Hasselbring, W., Carr, L., Hettrick, S., Packer H. and T. Tiropanis, Open Source Research Software, in *Computer*, vol. 53, no. 8, pp. 84-88, Aug. 2020, doi: 10.1109/MC.2020.2998235.

[4] Software Sustainability Institute (2021) Submissions to REF 2014. *The Hidden REF: Celebrating all Research Outputs*. Retrieved from https://hidden-ref.org/submissions-to-ref2014/ on 2021-11-22

[5] Software Sustainability Institute (2020) Guidance for the submission of software outputs to REF2021 [Blog post]. Retrieved from https://www.software.ac.uk/news/guidance-submission-software-outputs-ref2021 on 2021-11-22

[6] Wattanakriengkrai, S., Chinthanet, B., Hata, H., Kula, R.G., Treude, C., Guo and Kenichi Matsumoto,K. (2021) GitHub repositories with links to academic papers: Public access, traceability, and evolution. In *Journal of Systems and Software*, https://doi.org/10.1016/j.jss.2021.111117.

Scientific Software Development Laboratory: A Proposal

Jeffrey Carver, University of Alabama* Nasir Eisty, Boise State University

1 Challenge

Software development is a complex socio-technical activity that incorporates tools, languages, development environments, processes, measurement, and people, all of which must be studied and understood. When considering scientific software development, one could add aspects that may be less relevant to traditional software development, including complex problem domains, advanced computing platforms, and unique developer cultures. Studying the development of software in a traditional setting is challenging enough, however with the addition of these factors related to scientific software development, the challenge increases.

One of the key challenges to scientifically studying scientific software development (or software development of any kind) is a collaborative environment in which software engineering researchers can connect with software development teams to identify and study real problems. Scientific software development teams, like any software development team, have aspects of their software development process, tools, or teams that are sub-optimal. These deficiencies, if left unaddressed, could result in low-quality software and ultimately incorrect scientific results. However, in many cases, these teams do not have the time, resources, or knowledge to address these deficiencies. Conversely, many software engineering researchers do not have the knowledge of these real problems or access to the developers to study those problems.

Therefore we can describe the two-fold challenge as:

- **Scientific software teams, who have needs, do not have an easy way to connect with software engineering researchers who would like to study those needs.**
- **Software engineering researchers who would like to support scientific software developers do not have an easy way to identify the real problems or study those problems in context.**

2 Opportunity

A *virtual laboratory* that connects scientific software development teams with software engineering researchers for mutual benefit would help address the challenges above. Such a laboratory would provide a venue where scientific software development teams can describe their software development challenges and their willingness to participate in research to develop and evaluate solutions to those challenges. Examples of the types of challenges include: understanding code, finding reviewers, using proper metrics and benchmarks, using proper testing techniques and frameworks, CI issues, and long-term sustainability management. In addition, software engineering researchers interested in scientific software development can monitor these challenges to identify those they are interested in addressing. For example, software engineering researchers might produce training materials, develop new tools and techniques specifically for scientific software, formalize processes, or create guidelines. After this "matchmaking", the researcher and the scientific software development team can develop a plan. The formalization of this type of laboratory will allow for the following benefits:

- Multiple software engineering researchers can propose solutions to the same challenge and use research studies to identify the strengths and weaknesses of each approach.
- Different scientific software teams who face the same, or similar, challenges will be able to identify each other and work together with researchers on solutions.
- The laboratory will provide the opportunity to systematize the challenges, the solutions, and the results from research studies in a manner that can scientifically advance our understanding of scientific software development to develop new metrics, tools, and development processes, and training materials.

*carver@cs.ua.edu

We model this proposal on the successful *Software Engineering Laboratory*, a collaboration between NASA, Computer Sciences Corporation, and the University of Maryland (in which Carver participated as a Ph.D. student). This 25-year collaboration focused on understanding software development principles in context, the factors that affect the development of software, and the types of problems encountered all within the context of scientific research software focused on space flight. In the DOE software context, our proposed laboratory build on the experiences of this laboratory, as described in a paper reflecting on its history [2].

This type of portal will also provide the infrastructure necessary to help answer many of the questions from the solicitation, including:

- What methods can be used to study the development and/or use of scientific software?
- What methods either uniquely apply or uniquely do not apply to the development of scientific software?
- How can we best integrate social and cognitive sciences into scientific software activities?
- How can we study changes in culture affecting scientific software development and use?
- What unique changes in scientific software development would improve productivity, accuracy, trust, and/or reliability?
- How can we motivate changes in the scientific-computing community to improve practices for software development and use?
- How will static and dynamic analysis methods affect the practice of scientific software development?
- How are scientific software developers and users atypical from other larger software communities?
- When and how can research and methods from other software communities be adapted and adopted for scientific software?

3 Timeliness or Maturity

With the growing interest in the appropriate use of software engineering practices for the development of scientific software, the time is ripe for the creation of this laboratory. As more scientific software developers realize the benefits of software engineering and seek out the most appropriate methods, the benefits provided by this proposed laboratory will be readily recognized by many. The authors' prior work has begun to address a number of questions posed by the solicitation [1, 3, 4, 5, 6, 7], but there is a need for additional work.

References

[1] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Jeffrey C. Carver. Software engineering practices for scientific software development: A systematic mapping study. *Journal of Systems and Software*, 172:110848, 2021.

[2] Victor R. Basili, Frank E. Mcgarry, Rose Pajerski, and Marvin V. Zelkowitz. Lessons learned from 25 years of process improvement: The rise and fall of the nasa software engineering laboratory. In *of the NASA Software Engineering Laboratory, IEEE Computer Society and ACM International Conf. on Soft. Eng., Orlando FL*, pages 69–79. ACM Press, 2002.

[3] Nasir .U. Eisty and Jeffrey C. Carver. Developers perception of peer code review in research software development. *Empirical Software Engineering*, 27, 2022.

[4] Nasir U. Eisty, George K. Thiruvathukal, and Jeffrey C. Carver. A survey of software metric use in research software development. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 212–222, 2018.

[5] Nasir U. Eisty, George K. Thiruvathukal, and Jeffrey C. Carver. Use of software process in research software development: A survey. In *Proceedings of the Evaluation and Assessment on Software Engineering*, EASE '19, page 276–282, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207–219, 2015.

[7] A. Nanthaamornphong and J. C. Carver. Test-driven development in hpc science: A case study. *Computing in Science & Engineering*, 20(5):98–113, Sep./Oct. 2018.

On the need for efficient and scalable solvers for ill-conditioned sparse matrix equations

J. Chen (jchen@pppl.gov), N. Ferraro (nferraro@pppl.gov), S. Jardin (jardin@pppl.gov)
Princeton Plasma Physics Laboratory

Challenge: Many problems of interest to research groups in the DOE Office of Science, and to a wider group of scientists and engineers as well, involve large sparse matrix equations, $\mathbf{A}\bullet\mathbf{X}=\mathbf{B}$, where the matrices $\mathbf{A}$ are poorly conditioned. Our experience is in the implicit MHD equations for fusion applications, but these types of equations also occur in other disciplines such as power grid simulations, subsurface flow, circuit simulation, and computational cardiology. In each instance, the condition number is very high, but for one or more different underlying reasons. This leads to difficulties in iterative solutions as the number of iterations required for solution to a given tolorance normally increases with the condition number.

Opportunity: Domain scientists or engineers, applied mathematicians, and computer scientists all look at these problems differently. The domain scientist might understand the reason for the large condition number, for example in terms of the wide range of time and spatial scales present in the underlying physics equations. The domain scientist and applied mathematician may discuss different preconditioning techniques based on addressing these underlying reasons. The computer scientist may be the best one to implement the preconditioner and iterative solver on modern CPU/GPU hardware in a scalable way. Perhaps the best way to approach this problem is to form an inter-disciplinary team around each domain -specific sparse matrix $\mathbf{A}$ possessing its own unique properties, and to work towards a solution for that specialized problem.

As a case in point, we refer to the sparse matrix used in the implicit MHD code M3D-C^1 [1] to advance the velocity one time step using the semi-implicit algorithm. This matrix has a wide range of eigenvalues as it was formed by discritizing equations describing all MHD waves with oscillations ranging from zone-to-zone to the size of the entire device. The code can also be run in 2 "reduced MHD" regimes where only the shear-Alfven wave is present (NV=1), where that and the slow wave are present (NV=2), or where all three waves are present, (NV=3 or full MHD). The geometry is a torus, which is represented by high order C^1 continuous finite elements in 3 dimensions. The elements are structured in the direction going the long way around the torus, but unstructured in the toroidal planes at a given toroidal angle. It was recognized that the underlying wave equations are stiffest within a given toroidal plane where the zone dimensions are also smallest. This led to the development of a block-Jacobi preconditioner that uses the sparse direct solver SuperLU_dist to concurrently factor each diagonal block describing wave propagation within a given plane and to use those to form a preconditioner. The efficacy of this approach can be seen in Figure 1 where we plot the magnitude of each of the eigenvalues of the velocity matrix $\mathbf{A}$ before and after being preconditioned for NV=1, 2, and 3. It is seen that the condition number for the full MHD case has been reduced from about 10^{15} to 30 by this preconditioning technique.

While this approach with the block-Jacobi preconditioner led to a workable solution method for the M3D-C^1 velocity matrix, it is not an optimal solution for the other matrices in M3D-C^1 describing the magnetic field and pressure evolution. The pressure equation is often dominated by a large anisotropic thermal conductivity that is many orders of magnitude larger in the direction parallel to the magnetic field than in the other directions. The ill-conditioning that results from this is not well addressed by the block-Jacobi preconditioner. Also, we have yet to see much speedup when the block-Jacobi preconditioned GMRES iterative solver is implemented on GPUs as compared to CPUs as it is communication dominated.

Timeliness: There is a need for this now because matrices are getting larger and more poorly conditioned, for example those in the implicit MHD applications describing ITER. Because of the large physical size of ITER and because of its large magnetic field strength and projected temperatures and densities, the condition number of the MHD matrices will be orders of magnitude larger than those now used in modeling existing tokamaks. Also, the HPC hardware is getting more difficult to program efficiently due to the CPU/GPU processors, communication bottlenecks, and memory hierarchies.

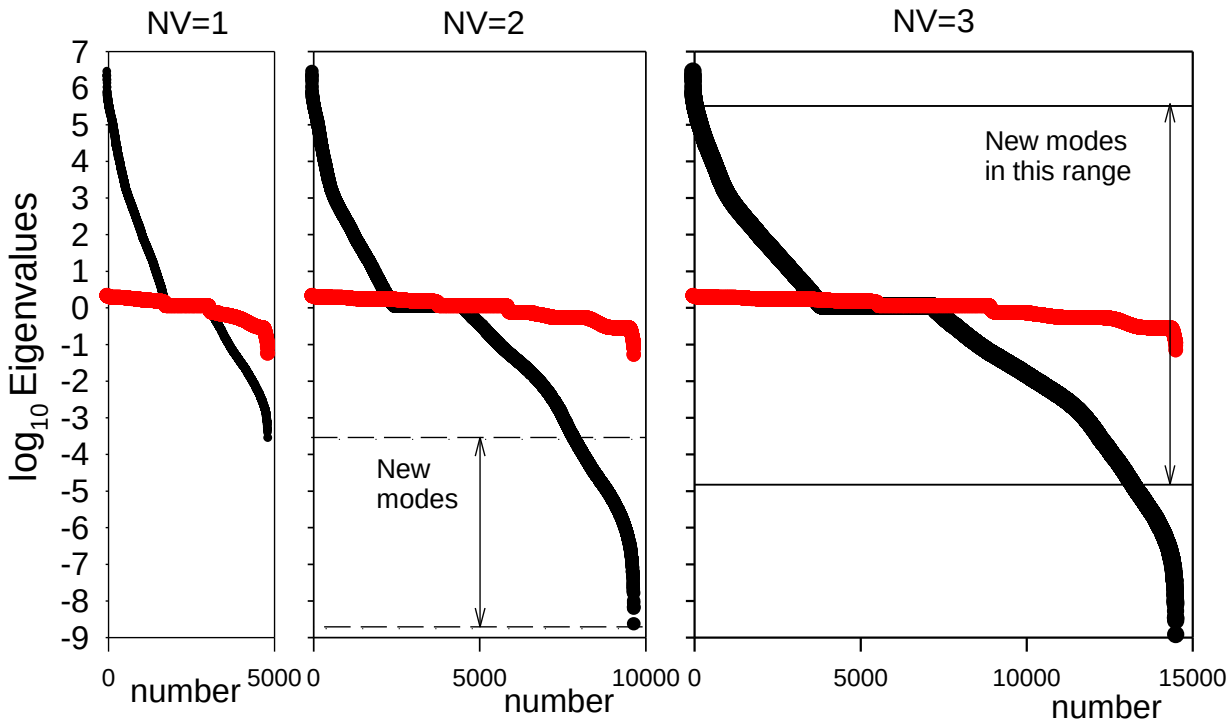


Figure 1 Plotted are all the eigenvalues of a full 3D M3D-C1 velocity matrix before (in black) and after (in red) the block-Jacobi preconditioner has been applied for 1,2 and 3 velocity variables. The ratio of the largest to the smallest eigenvalue has been reduced from about 10^{15} to approximately 30 for the 3 velocity variable case.

References:

[1] S. C. Jardin, N. Ferraro, J. Breslau, and J. Chen, "Multiple timescale calculations of sawteeth and other global macroscopic dynamics of tokamak plasmas", Comput. Science and Discovery **5** 014002 (2012)

Lawrence Livermore National Laboratory

TITLE: MAINTAINING PRODUCTIVITY AND INCREASING COMMUNITY OUTREACH FOR OPTIMIZATION ECOSYSTEM

AUTHORS:
Aravena Solis, Ignacio Andres, Lawrence Livermore National Laboratory, aravenasolis1@llnl.gov
Chiang, Nai-Yuan, Lawrence Livermore National Laboratory, chiang7@llnl.gov
Laird, Carl D., Carnegie Mellon University, claird@andrew.cmu.edu
Petra, Cosmin G., Lawrence Livermore National Laboratory, petra1@llnl.gov
Watson, Jean-Paul, Lawrence Livermore National Laboratory, watson61@llnl.gov

Challenges

As the ECP community continuously works toward an extreme-scale scientific software ecosystem, many high-quality and reusable packages have been introduced within the last decade. The goal of those tools is to reduce barriers to collaborative development, improve developer productivity, and improve software reproducibility and sustainability, while ensuring continued scientific success. However, the increasing number of packages also makes the entry barrier higher for the research communities who are not familiar with computer architectures because deploying packages to specific computing platforms requires a steep learning curve.

For example, algebraic modeling languages (AMLs) are scientific programming tools that allow users to describe the mathematical formulation of optimization problems in a concise and human-readable way. Researchers and engineers from different areas, *e.g.,* optimization and operations research, use modern AMLs such as Pyomo [1], JuMP [2] and AMPL [3] to perform their daily core activities. These scientific packages play a critical role as they provide a platform for researchers to design, solve, analyze, and modify the optimization model. It also allows the users to test and tune different optimization solvers, *e.g.,* IPOPT [4] and HiOp [5], to create a robust pair of problem formulation and optimization solver for their applications. Extensive work has already been done to develop efficient and user-friendly AMLs, both in commercial software and research code, as this is a proven approach that can dramatically improve the programming and deployment productivity. Figure 1 shows the state-of-the-art life cycle to develop and deploy an optimization solution in practice. Currently, a big gap between the offline development and online deployment exists, especially in extreme-scale applications but more broadly at all scales of computing.

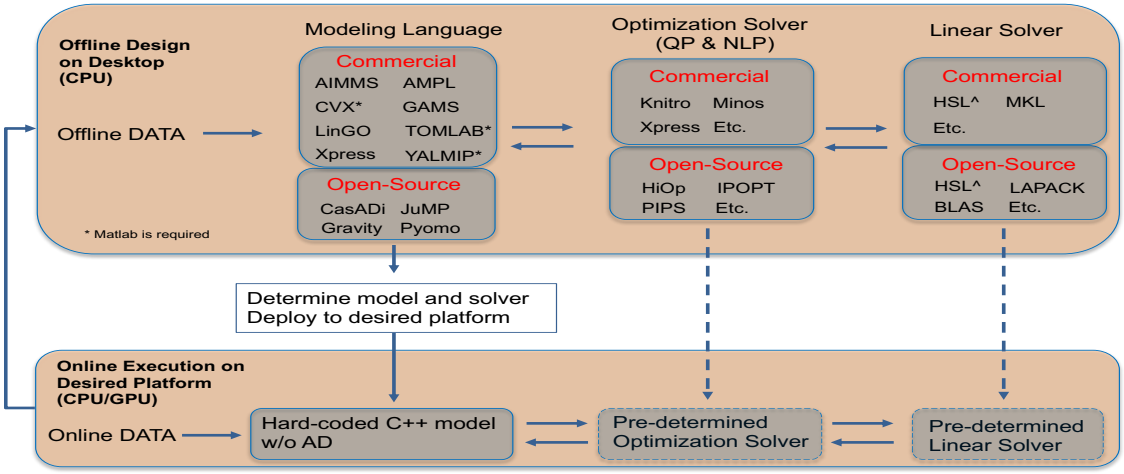


Figure 1: the state-of-the-art scientific ecosystem of optimization in practice. (*) requires commercial package MATLAB. (^) free access for teaching and research.

For industrial embedded computing, even though the model is determined via offline experiments, researchers still need to hard code the entire model (usually in C++ for efficiency) before deploying the model and solver onto the desired embedded platform. If the optimization model and solver need to be tuned again due to the unexpected behavior of online data/simulation, researchers need go back to the offline development and start the process again. Example of the practical applications on embedded platform includes, but not limited to, real-time additive manufacturing, and advanced autonomy in robots and vehicles.

On the other end of the spectrum, for extreme-scale applications that requires HPC, the lack of GPU support in current AMLs is a key limitation preventing high productivity. While development time may be reduced by using a high-level abstraction package, such as Kokkos/RAJA[6],[6] there is still a manual translation step from the model in the AMLs. Furthermore, to achieve portable performance on different platforms, additional work is required to implement a GPU variant. For researchers who are not familiar with C++ or GPU coding, this step can dramatically reduce their productivity.

Opportunities

To improve developer productivity and software sustainability, a more automated and seamless transition from CPU software development to its GPU variant is needed. Particularly, a more automated user-friendly optimization ecosystem should be provided. One general solution to achieve this is the code generation technique. For example, after offline development, CVX [7] and CasADi [8] can generate C code for users to use on their desired embedded system. Both packages claim this is important to achieve the best performance on embedded systems for small-scale problems, since the generated C code has no dynamic memory allocation. However, CVX only supports quadratic programming, where the 1st and 2nd order derivatives do not change; and CasAdi only generates code for the model functions, without the core of optimization solver/algorithm. As a result, there is no existing ecosystem that can provide optimization community an automated way to develop and deploy optimization solutions to the target platform. Having such capability bridges the gap between off-line research and on-line practical use, while achieving the optimal performance on the platform. It addresses some priority research directions for national interests [9][10], by providing a modular and automated solution to improve software productivity and sustainability.

Timeliness or maturity

While computational scientists in the ECP primarily focus on HPC (with GPU capabilities), researchers and engineers from academia and industry overwhelmingly employ CPU computing. Many ECP packages tried to address this issue by making the platform-dependent programming easier to develop, e.g., adding a portable abstraction layer [6]. Due to the maturity of these tools in the recent years, it becomes possible to automate a customized optimization solution for a particular application, on a particular platform. This automated approach can bridge the gaps between different research communities, while maintaining the programmers' productivity and achieving better performance.

References

[1] Hart, William E., Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. "*Pyomo-optimization modeling in python*." Vol. 67. Berlin: Springer, 2017.

[2] Dunning, Iain, Joey Huchette, and Miles Lubin. "*JuMP: A modeling language for mathematical optimization.*" SIAM review 59, no. 2, 2017.

[3] Fourer, Robert, David M. Gay, and Brian W. Kernighan. *Ampl*. Boyd & Fraser, 1995.

[4] Wächter, Andreas, and Lorenz T. Biegler. "*On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*." Mathematical programming 106, no. 1 (2006): 25-57.

[5] Petra, Cosmin and Nai-Yuan Chiang, *"HiOp – User Guide (version 0.5)"*, *LLNL-SM-743591,* Lawrence Livermore National Lab, United States, 2021.

[6] https://www.exascaleproject.org/wp-content/uploads/2020/02/ECP_ST_Kokkos_RAJA.pdf

[7] Mattingley, Jacob, and Stephen Boyd. "CVXGEN: A code generator for embedded convex optimization." *Optimization and Engineering* 13, no. 1 (2012): 1-27.

[8] Andersson, Joel AE, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. "CasADi: a software framework for nonlinear optimization and optimal control." *Mathematical Programming Computation* 11, no. 1 (2019): 1-36.

[9] Vetter, Jeffrey S., Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas et al. "*Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity*.", 2019.

[10] Heroux, Michael A., Lois McInnes, David Bernholdt, Anshu Dubey, Elsa Gonsiorowski, Osni Marques, J. David Moulton et al. *Advancing Scientific Productivity through Better Scientific Software: Developer Productivity and Software Sustainability Report*. No. ORNL/TM-2020/1459. Oak Ridge National Lab., Oak Ridge, TN (United States), 2020.

Title: The Last Mile in Software Engineering: In Support of Comprehensive Scientific Research Software Testing.

Authors: Shreyas Cholia, Valerie C. Hendrix, Keith Beattie (Lawrence Berkeley National Laboratory)

Challenge: Research software is often on the cutting edge of ideas in technology, science, and cross-disciplinary collaboration. Such research projects often have aggressive deadlines where software requirements are fuzzy at the start and only take shape as the cross-disciplinary team sketches out what is needed to make a scientific impact. Software testing is the last mile of software engineering for assessment, quality, and correctness of software function. However, it is often not addressed with enough time to make a positive impact. The exit criteria, i.e., when to call testing complete, is often determined by how much time and budget are left. This can result in scientific software that is difficult to maintain, extend, and, most importantly, share with a broader audience

While overall adoption of sustainable software engineering practices is gaining traction in the scientific research community, there is still a gap in the broad knowledge and implementation of software testing practices, particularly in small and early-stage projects. While unit-testing may be common, more comprehensive end-to-end testing (integration testing, system testing) still relies on ad-hoc approaches and is often neglected. Moreover, given resource constraints, many teams do not have staff allocated for testing the software, and rely on the same developers to handle both development and testing, which may lead to blind spots. Industry practices like test-driven development and continuous integration are often under-utilized. Scientific software also presents novel testing challenges when its artifacts are the code used in varied analysis contexts (such as a Jupyter Notebook or a set of scripts) that produces a set of science results. There is a significant dependency on the underlying scientific data, and common testing methodologies don't always account for testing against real-world data, or data at different scales. With science gateways and web interfaces becoming increasingly popular as a means to interact with the scientific workflow, user-interface testing takes on greater importance, but is often left out of the formal automated testing process entirely. We must also recognize that production scientific software pipelines may be multi-stage workflows that depend on external data and computing resources across different stages of the workflow. Testing such a software system in a repeatable and controlled manner, without having to depend on expensive production resources requires innovative testing approaches.

We believe that a well thought out approach to scientific software testing, and a set of accompanying guidelines and practices are much needed by the community, along with the research, development, and implementation of software testing tools.

Opportunity: In order to produce quality scientific software that lives past its initial funding phase it is important to put practices in place from the beginning. We believe that there is significant value to be derived from developing a set of testing practices

and methods that will allow early-stage scientific software projects to produce high-quality, reliable software, while greatly reducing long-term support and sustainability costs.

Research is needed into determining the proper exit criteria for scientific software testing, which may go beyond algorithmic correctness, and depend on a comparison of methods producing a range of acceptable values and errors. Scientific analysis artifacts need to become first-class products of the software engineering ecosystem, and appropriate testing tools need to be applied directly to these. Innovation will be needed to enable testing on more complex scientific software with advanced interactive user interfaces or distributed resources. We can look to industry practices for these, but more research and development will be needed to adapt these to the scientific space. From a socio-technical standpoint, we should consider best-practices for cross-disciplinary teams to undertake testing as a key component of the development process. While research software may not have the same resources as large commercial endeavors, we can take advantage of domain experts on the team being closely involved in the process, so that testing becomes a core component of the project, without over-extending available resources. For instance, we have seen success with “hackathon” style testing sessions where science stakeholders, software developers, and potential users all work in a dedicated, concentrated session to systematically uncover bugs in the system with everyone in the same room (physically or virtually).

Timeliness: As scientific software systems grow increasingly complex, software testing takes on greater importance in being able to ensure the reliability and accuracy of the scientific results. Scientific reproducibility depends on properly functioning, reliable software. Perversely, the complexity of these systems coupled with funding constraints makes it so that testing often takes a backseat in the development cycle. It is our belief that testing should be baked into the fabric of every scientific software project. The time is right to invest in testing for scientific software - as a research area, as a set of community practices, and as a set of tools and technologies that projects can leverage. This will ensure that projects at all scales are able to produce high-quality software that can be supported in a sustained manner, keeping costs down, enabling broad use and distribution of the software, and ultimately ensuring successful and reproducible science.

References:

1. National Academies of Sciences, Engineering, and Medicine. 2019. “Reproducibility and Replicability in Science.” Washington, DC: The National Academies Press. https://doi.org/10.17226/25303
2. Greg Wilson. 2014. “Why We Don't Teach Testing (Even Though We'd Like To).” https://software-carpentry.org/blog/2014/10/why-we-dont-teach-testing.html
3. Atlassian Blog. “The different types of software testing.” https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Adopting Industry Software DevOps Practices to Improve Scientific Software

Struan Clark (struan.clark@nrel.gov), Nalinrat Guba (nalinrat.guba@nrel.gov),
Rachel Hurst (rachel.hurst@nrel.gov), Kristi Potter (kristi.potter@nrel.gov)
Computational Science Center, National Renewable Energy Laboratory

1 Challenge

Scientific software development has significantly different goals and measures of success than software development in industry. In industry, the software development and deployment processes directly contribute to market competitiveness and as a result there is significant incentive for continuous improvement. The difference between a elite software development business which produces performant, reliable software and a mediocre business can be directly measured in profitability and user satisfaction.

In the scientific software development world, there are significant challenges faced by scientists in their pursuit of discoveries, with major roadblocks in the form of "management of large code-bases and collaboration, maintainability, and reuse opportunities" [1]. Scientific software is developed by smaller, ad-hoc teams that are primarily interested in the underlying research being conducted, vs. learning and practicing state of the art software engineering techniques. These teams may be made up of leaders in the field for their respective research but who are not formally trained on software best practices or newly available tools and have no funding or time allocated to learn good techniques for collaborating on increasingly large and complex research code bases. Scientific software is also generally limited in scope to meet the specific goals of the research being conducted. There are no natural incentives for building software that will continue to be maintainable and usable after the research has been completed.

This results in scientific software being a means to an end, vs. an end in and of itself as it tends to be in industry. Scientists and researchers are interested in developing software to solve problems in their fields and should not be expected to also solve the less interesting (to them) problems with the software development process itself.

2 Opportunity

Software development in industry has already implemented solutions to some of these problems. The concept of combining software development and IT operations (commonly referred to as DevOps) has resulted in the maturity and widespread adoption of enabling technologies that reduce the pain of building maintainable, reproducible software that is approachable for those without a strong software engineering background. The opportunity exists to adapt and adopt some of these technologies and techniques from industry as-is, without the need to develop a bespoke solution. Two of the major concepts that would be immediately beneficial to investigate are containerization and continuous integration/continuous delivery (CI/CD).

Containerization is the concept that application code and its execution environment can be bundled into a single logical object which can then be shared with others and run without tedious setup steps. While originally designed for isolation and security of application processes, containers are also an extremely convenient way to share code, dependencies, and even raw data via a single compressed archive. Perhaps best of all there is no dependency on a specific language or toolchain. Containerization is adaptable to most software workflows and allows for teams to continue to use tools they are familiar with, but in a format that is easier to share, maintain, and reproduce which results in greater productivity across teams and a more reliable product.

Containerization can be implemented as a manual step as part of a software team's workflow, however even more benefits can be realized by building containers as part of an automated CI/CD workflow. CI/CD allows for all the tricky manual steps of a software build, test, and deployment

process to be executed automatically. Essentially, this means that for the investment of a one-time cost into scripting the software build and test process, this same process will then execute automatically every time the code is updated. This enables automated comparison of outputs from previous iterations of the software to be conducted allowing for verifying that the results of the software are accurate. In addition, deliverables are automatically created and deployed for each version of the code without error prone manual packaging steps providing a further boost to trustworthiness of the software. Finally, CI/CD infrastructure allows team members to contribute to code changes without needing to load all of the software or data onto their local machines allowing for a broader set of individuals to contribute and be productive without an extended ramp-up time spent tweaking configuration and environment settings and manually downloading large data files.

3 Timeliness And Maturity

There is an opportunity to adopt industry DevOps practices in the scientific software community immediately. The formalization of DevOps methodologies and maturing of DevOps infrastructure, containerization specifically, have led to DevOps and containerization being recognized as the most effective way for software development organizations to develop, build, test, and deploy code. The DevOps Research and Assessment (DORA) research program provides an academically rigorous quantification of software in industry, by measuring an industry-standard set of four key software metrics: change lead time, deployment frequency, mean time to restore (MTTR), and change fail percentage. The 2019 State of DevOps report produced by DORA states that "Our research continues to show that the industry-standard four key metrics of software development and delivery drive organizational performance" [3]. In addition, industry analyst firm Gartner states that "By 2025, more than 50% of enterprises will adopt a centralized platform engineering and operations approach to facilitate DevOps self-service and scaling, which is a significant increase from less than 20% in 2020. By 2025, more than 85% of global organizations will be running containerized applications in production, which is a significant increase from fewer than 35% in 2019" [2]. A comparable gain in the scientific computing world would be extraordinary for research reproducibility. Scientists would be able to download a ready-to-go environment and run the software within immediately. However, community and organizational support and demonstration that containerization is a benefit to productivity will be vital to navigate barriers including lack of expertise and lack of confidence in the utility of the technology.

By leveraging these mature practices from industry, and tracking performance using industry-standard metrics, scientific software developers will be able to realize the same improvements, resulting in more time available for them to focus on the actual subject of their research instead of spending unnecessary time overcoming roadblocks in software configuration and packaging. The overall impact of success of this initiative would be scientific software teams that are more productive and scientific software applications that are more accurate, trustworthy, and reliable.

References

[1] Elvira-Maria Arvanitou et al. "Software engineering practices for scientific software development: A systematic mapping study". In: *Journal of Systems and Software* 172 (2021), p. 110848. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110848. URL: https://www.sciencedirect.com/science/article/pii/S0164121220302387.

[2] Arun Chandrasekaran. "Best Practices for Running Containers and Kubernetes in Production". In: (Aug. 2020). URL: https://www.splunk.com/en_us/form/gartner-best-practices-for-running-containers-and-kubernetes-in-production.html.

[3] Dr. Nicole Forsgren et al. "State of DevOps 2019". In: (Sept. 2019). URL: https://services.google.com/fh/files/misc/state-of-devops-2019.pdf.

Engaging a Complex Domain Science in Software Engineering Best Practices: What Works and What Doesn't

T. Daniel Crawford,[a,b] Paul Saxe,[b] and Theresa L. Windus[b,c]
[a]*Department of Chemistry, Virginia Tech, Blacksburg, Virginia 24061 U.S.A.*
[b]*The Molecular Sciences Software Institute, Blacksburg, Virginia 24060 U.S.A.*
[c]*Department of Chemistry, Iowa State University, Ames, Iowa 50011 U.S.A.*

Challenges: The innumerable scientific breakthroughs in the fields of computational chemistry, physics, and biophysics in the last several decades have been made possible by the evolution of dozens of community codes, some with lifetimes reaching back to the earliest days of computing. These programs include both open-source and commercial packages, contain millions of lines of hand-written code in an amalgam of programming languages, and make use of essentially every algorithmic motif. The Gordian complexity of such programs is a natural reflection of the intricacy of the problems they are designed to solve, the rich feature set they offer, and the hardware for which they were developed. Such diverse code stacks require vast numbers of human hours and hundreds of programmers to develop and maintain.

The vast majority of the software development in the field is carried out by graduate students and postdoctoral associates for whom the natural motivation is short-term scientific expediency — a "just get the physics working and then publish it" approach[1] – rather than establishing a foundation for the sustainable, long-term effort needed to address large, complex scientific challenges. Further compounding the problem is that many of these young researchers are simultaneously working to understand the scientific problem and how to write the code to solve it. Indeed, the complexity of the science often presents even senior investigators with unexpected software challenges because the equations are complicated, approximations are required, and it is not *a priori* obvious which approaches will work and which will fail. Thus, the entire process of developing the code to implement a new scientific idea is often initially unclear and difficult to predict, leading to an iterative learning process.

The current paradigm for software development in the field has developed organically over many years, often yielding a community-driven approach in which multiple labs contribute to a single package. This approach creates many challenges to progress and sustainability, especially towards interoperability of packages, the efficient use of new, disruptive computing architectures, and scaling to larger numbers of developers. The challenge is further exacerbated by the rapidly changing hardware landscape. The scientific problems many community codes are designed to address require considerable computational resources, both in terms of computing speed and memory. While some community codes have done an excellent job of multi-CPU and GPU parallelization, their software complexity has also increased dramatically, often with completely distinct code paths for different hardware. Most community codes, however, are simply unable to take full advantage of modern supercomputing resources, a problem that will only expand as computing technology advances.

While historically, funding for such software development has been indirect, at best, more recently, initiatives by the U.S. National Science Foundation and the Department of Energy have provided support specifically to raise the quality of software engineering within scientific domains. Programs such as the NSF's Cyberinfrastructure for Sustained Scientific Innovation,[2] and the DOE's Exascale Computing

[1] C. R. Ferenbaugh, 2013, Workshop on Sustainable Software for Science: Practice and Experiences, http://arxiv.org/abs/1309.1428v1. Experiments in Sustainable Software Practices for Future Architectures".
[2] https://beta.nsf.gov/funding/opportunities/cyberinfrastructure-sustained-scientific-innovation-cssi

Project[3] are specifically intended to facilitate improved software development across a broad spectrum of domain sciences. Nevertheless, much of the scientific code development underway within the field is carried out as a by-product of more traditional grant support.

Opportunities: Given this picture of the science and software in our field, it is perhaps unsurprising that fruitful collaborations between computational scientists and software engineers have been few. While there are a number of notable examples where such partnerships have yielded excellent results such as NWChemEx[4] and NAMD,[5] most scientific software development (particularly in academia) is carried out solely by the scientists themselves. In our opinion, however, there are at least two areas where the deep knowledge and skills within the computer science community could significantly benefit domain computational scientists.

The easiest and most obvious is at a low level: teach the domain scientists how to use modern software engineering tools to be more productive and effective. In general, computational scientists will readily accept new methods and approaches, provided they are trained in how best to use them in the context of their science. Given that computational scientists are focused on the science and that coding is primarily a means to an end, a recipe-based approach is often the fastest and most efficient way to help them adapt quickly to using code management systems, building tools, containers, continuous integration methods, unit-testing systems, strategies for modularity, etc. All of these are excellent targets where guidance from the computer science and software engineering communities can truly make a difference. Additional information and guidance on the best teaching methods to integrate software engineering tools into the domain science could benefit from educational expertise.

At a higher level, the incredibly diverse software ecosystem that characterizes computational chemistry, physics, and biophysics could provide a rich environment for data scientists and sociologists to explore how various modes of software development and community engagement are particularly popular, effective, etc. Within our scientific domain, codes with similar functionality are written in different languages, with different structure and modularity. There are successful codes written in FORTRAN decades ago that are competitive and maintained, but there are also codes that have been rewritten multiple times using more modern languages and software engineering. Some packages have been developed in collaboration with software engineers, and others written purely by domain scientists. Some codes are written by small, cohesive groups and others by large, diffuse communities, and there are both open-source codes and commercial codes.

Timeliness: There is no consensus within the community about the strengths and weaknesses of various software development approaches (though many strong opinions abound!). Could studying these approaches as they are currently used in our field, both from a software standpoint as well as a sociological perspective, identify the most successful? Are different approaches better suited for certain types of projects or teams? The seismic shift in the computing environment is exposing deep flaws in the current approach limiting the ability to fully use the new capabilities. Developing more effective approaches will help science keep up with the hardware. Knowledge from computer science is already having an impact, but empirical evidence about the impact of different software methodologies would be a significant step forward towards understanding the “science of scientific software development”.

---

[3] https://www.exascaleproject.org/
[4] https://www.exascaleproject.org/research-project/nwchemex/
[5] https://en.wikipedia.org/wiki/NAMD

How to measure and improve the quality and quantity of software developed by agile teams of scientific software developers

Paul S. Crozier, Sandia National Laboratories, pscrozi@sandia.gov

Challenges

Agile software development teams need better tools and metrics for measuring their output, both in terms of software quality, and in terms of code quantity. Such metrics would allow teams to better understand how to control and improve their output, and assess the value of team experiments.

There is an ongoing debate among practitioners of scientific software development regarding which are the best teaming constructs under which to develop. While the broader software development community seems to have coalesced around what are termed agile software development practices, including scrum, Kanban, and so forth, the scientific software development communities have been slower to adopt these teaming constructs. For a team of scientific software developers considering adopting an agile approach, it is difficult to gauge how much efficiency will likely be gained by doing so. Many agile practitioners, including certified scrum professionals, are quite zealous about the benefits of going agile, and quite particular about which practices must be adopted to be considered truly agile. Others have a more practical do-what-works approach. What is lacking is that neither group has the tools, metrics, or methods to scientifically compare different software development teaming constructs and variants. Much research needs to be done to develop an approach for comparing teaming constructs in a fair way.

In our scientific software development work at Sandia, we have encountered many practical examples of situations where we are not able to address questions around which software development teaming construct would be best, or just how successful experiments have been at improving team productivity. Should we use waterfall or agile? Scrum or Kanban? Should a scrum master be required? Is it better to have a code developer or user as a product owner? What is the optimal team size? Is pair programming better than solo programming? Should continuous integration be required? What level of autonomy should code development teams have to determine their own team practices?

Currently, our approach to assessing the value of various teaming constructs and experiments is highly subjective and non-scientific. While we do look at team velocity (number of story points a team completes per time), we do not do so in a rigorous scientific way and typically suffer from a myriad of confounding factors.

Opportunity

If we could develop universally applicable metrics for assessing the performance of scientific software code development teams, those metrics could then help teams determine the best teaming constructs and team practices for their particular circumstances, and the level of success of experiments that are attempted. Assessment metrics could be developed that rigorously and scientifically determine the quantity and quality of code developed by a team. It would also be helpful to develop metrics to gauge the satisfaction and professional wellness of the code developers and stakeholders.

The broader software development community outside of scientific software development has a broader wealth of experience at addressing challenges like these. While scientific software development has its own unique challenges, there is much in common. There is an opportunity for the scientific software development community to learn from the broader software development community. Research should be done to glean ideas from the broader community that could be successfully applied to scientific software development. The best ideas from the broader community for measuring and assessing the performance of software development teams could be borrowed and made more scientifically-rigorous.

Timeliness

Over the past few decades, software development teams have gradually migrated towards modern and agile software development practices and teaming. Over that same timeframe, software development, particularly for heterogeneous HPC hardware, has become more challenging. The workforce itself has become more transitory. Code development funding and talent are more constrained. The tools used by code developers have become more sophisticated. Lone wolf programmers are becoming rarer because this challenging environment requires teams with diverse backgrounds and skillsets to be successful.

If headway could be made on measuring and improving the productivity of scientific code development teams, more and higher quality code could be produced, limited resources could be more wisely stewarded, best code development teaming practices could be adopted and shared, and team members would be more successful and happier.

Acknowledgement

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Best practices to enable a culture of effective cross-functional scientific software collaboration for DOE science

Rob Crystal-Ornelas[1], Ann Almgren[2], Keith Beattie[2], Dan Gunter[2], Charuleka Varadharajan[1]

[1]Earth and Environmental Sciences Area, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, rcrystalornelas@lbl.gov
[2]Computing Sciences Area, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

Challenges

U.S. Department of Energy science requires large collaborative teams with diverse scientific backgrounds and skill sets to tackle complex problems (Ramachandran et al., 2021). Domain scientists who may be used to a more independent approach to research are often integrated into cross-functional teams tasked with developing scientific software. Understanding best practices for collaboration in cross-functional teams can help the DOE foster autonomy, creativity, and high-impact scientific software.

Cross-functional software teams must be **culturally** and **procedurally** equipped to make the most of the talents and expertise of all their members (IDEAS-ECP Team, 2020). Existing research has explored the challenges of integrating data scientists into agile and software development teams (Davenport & Patil, 2012; Hukkelberg & Berntzen, 2019). However, there is little research investigating how to best integrate domain scientists, especially those representing the diversity of career stages (staff scientist, postdoc, intern) typical of DOE projects (IDEAS-ECP Team, 2020), into cross-functional scientific software development teams.

Here we summarize **cultural challenges** to cross-functional team collaboration. First, cross-functional teams often need to integrate members who are new to *team science*. For many researchers coming from university-based work, the archetype of the “solo scientist” is not a choice, but a necessity (Nosek, 2019); group problem solving takes practice. Second, researchers join cross-functional teams with a *range of skill sets,* and meeting the new employees at their skill level requires patience and support. Lastly, domain scientists who develop software may not see the value in investing in *long-term software development* beyond the targeted research question, while computer scientists typically develop code that serves diverse users through software best practices and requires additional resources to maintain.

The **procedural challenges** of building effective cross-functional teams are related to the distributed chain of command typical of DOE science teams as well as the increasing number of geographically scattered collaborations. The *distributed chain of command* found across DOE science can create a barrier if a team lacks a person or process to enable consensus-building and effective decision-making. Many teams also involve *geographically scattered collaboration* which requires platforms that can enable asynchronous collaboration.

Opportunities

Overcoming the cultural and procedural challenges to cross-functional scientific software development will have the overall effect of bringing an even greater legitimacy and job satisfaction to the scientific software development process while making sure all the people in cross-functional roles feel valued and that their voices are heard. Recognition and job satisfaction are both essential to maintaining a strong, resilient workforce.

Cultural opportunities

When DOE science teams effectively integrate those *new to team science* into projects, they expand the community (e.g., domain scientists, postdocs, interns) of people who "bridge the gap" between software and domain science (Honeyman & Treloar, 2021). The opportunity is that highly functioning teams can have output that's greater than the sum of their parts. Researchers, postdocs, and students can learn new skills that are valuable to the job. Project leaders face the challenge of placing members in roles that not only fit their current skills, but enable them to grow in directions in which they express interest. Ultimately this is a balancing act, and a successful team is one in which members both have autonomy in their tasks and can safely ask for help. A final opportunity is for computational scientists to develop tools and templates that can enable domain scientists to follow *software best management practices relevant to scientific research*, and for education on both sides.

Procedural opportunities

Overcoming the challenge of DOE science's *distributed chain of command* can potentially lead to greater sense of dedication to a project if effective consensus building is used in decision making. Having more effective *widespread collaborations* can expand the network of people that are using and contributing to scientific software (Ramachandran et al., 2021). Innovative use of existing tools for software collaboration are already bringing more domain scientists into the software development space (Crystal-Ornelas et al., 2021).

Timeliness or maturity

The IDEAS-ECP report (2020) states: "It is the nature of software projects that the poor practices of one subteam will have a disproportionate negative impact on the whole project." We also believe that the converse is true, that widespread procedural and cultural best practices, when integrated into cross-functional teams, have the potential to accelerate scientific impact. DOE science is inherently team-based, and projects are increasingly collaborative, involving a greater diversity of backgrounds in software development, domain expertise, and career stages. Outlining best practices can help cross-functional teams at the DOE foster the autonomy, creativity, and high-impact scientific software that our teams are known for.

References

Crystal-Ornelas, R., Varadharajan, C., Bond-Lamberty, B., Boye, K., Burrus, M., Cholia, S., et al. (2021). A guide to using GitHub for developing and versioning data standards and reporting formats. *Earth and Space Science*, *8*(8). https://doi.org/10.1029/2021ea001797

Davenport, T. H., & Patil, D. J. (2012). Data scientist. *Harvard Business Review*, *90*(5), 70–76.

Honeyman, T., & Treloar, A. (2021). *A National Agenda for Research Software*. https://doi.org/10.5281/zenodo.4940274

Hukkelberg, I., & Berntzen, M. (2019). Exploring the Challenges of Integrating Data Science Roles in Agile Autonomous Teams. In *Agile Processes in Software Engineering and Extreme Programming – Workshops* (pp. 37–45). Springer International Publishing.

IDEAS-ECP Team. (2020). Advancing Scientific Productivity through Better Scientific Software. Retrieved from https://www.exascaleproject.org/wp-content/uploads/2020/01/IDEAS-ECP.Report.v1.0.pdf

Nosek, B. (2019). Strategy for Culture Change. Retrieved November 4, 2021, from https://www.cos.io/blog/strategy-for-culture-change

Ramachandran, R., Bugbee, K., & Murphy, K. (2021). From open data to open science. *Earth and Space Science*. https://doi.org/10.1029/2020ea001562

Title: **The proof is in the Pudding: Detecting problems with software in production**

Authors: Ewa Deelman, University of Southern California (deelman@isi.edu contact author), Miron Livny, University of Wisconsin Madison (miron@cs.wisc.edu), Prasanna Balaprakrash, ANL (pbalapra@anl.gov), Mariam Kiran, LBNL (mkiran@es.net), and Anirban Mandal, RENCI (anirban@renci.org)

Challenge: Currently, a growing number of application codes are developed in small environments and do not scale well for use in complex workflows and do not scale well to take advantage of the available cyberinfrastructure (CI). Scientists often develop codes on local workstations or laptops first and then work on scaling them up. To execute at scale they often rely on the high-throughput computing (HTC) paradigm, where the parallelism is achieved via invoking the same workflow on individual members of a dataset or via parameter-sweep approaches, running the same workflow with different parameter settings for the constituent applications.

When these codes are integrated into the larger workflows and are migrated to distributed environments such as OSG [1] or are distributed across the DOE computational facilities, the codes fail because of authentication issues (to resources and data), portability issues (missing or incompatible libraries, wrong environment settings), misconfigured servers, missing file systems, and/or mismatches in data flowing through the workflow, among others.

Opportunity: There is an opportunity to provide more automation in the software development process, where the users' codes are automatically and frequently tested at scale. When the software is being developed, it could be automatically tested as a unit on several different environments using workflow technologies. For relatively complex functional and integration tests, scientific codes that involve distributed memory codes often rely on pseudo-oracle. A pseudo-oracle is a reference program that takes the same input as software under test so that the outputs from it can be compared for testing. Typically, these are quite expensive to run and require a significant number of compute nodes, computation time, or both. This challenge prevents rigorous and extensive testing for MPI-based applications. Perhaps, one of the grand challenges in scientific software testing is the oracle problem, which verifies whether a program is working correctly without humans in the loop. In fact, due to their complexity and the lack of test oracles, a significant number of multi developer scientific softwares did not have end-to-end, acceptance, performance, and smoke testing. To that end, iterative metamorphic testing is a promising approach that has achieved significant successes in testing complex software such as GCC and LLVM. The approach takes high-level metamorphic relations (MRs), generates mutation tests for the software, and increases the strength of the MRs iteratively for further stronger tests. Nevertheless, developing such iterative metamorphic testing for scientific software requires scale and complex workflow technologies.

We envision a workflow management system (WMS) that finds a set of diverse resources, executes the codes on them, analyzes the results, potentially using AI techniques, and reports them back at an appropriate level of abstraction to the scientists. The system could potentially recommend fixes or do simple fixes itself. Such a WMS would have to differentiate between application-level system-level failures and anomalies, potentially retrying the application on different resources of the same type, and/or reasoning about the problems associated with software dependencies, and/or reasoning about solutions to problems previously seen with that or other applications or resources. To support these capabilities we would need better means of collecting and communicating data about what went wrong. Differentiating application-level versus system-level problems would require the development of tools and methods that continually gather information from the computing environments, detect anomalies, and attribute them to specific issues. Additional capabilities would include reasoning about potential fixes or performing these fixes and test their correctness using explainable and interpretable AI/ML methods. This methodology can then be extended to testing of entire workflows at scale. Furthermore, we envision AI-enabled self-optimizing and

intelligent workflow systems that can continually learn and improve their own performance (resource-aware data staging, dynamic resource allocation, tuning parameters of systems) as well as react to the failures by evaluating and exploring different workflow configurations.

The cyberinfrastructure can also be potentially simulated in order to better determine the causes of test failures or to evaluate codes on future systems. Using advanced cyberinfrastructure and workflow modeling, simulation, and AI/ML, we envision a digital twin to enable validation, verification, and correctness of the scientific software. Specifically, a well designed digital twin can provide configurable cyberinfrastructure platforms on which various aspects of the scientific software can be tested. With digital twins, software developers will be able to run the scientific software in a virtual environment, test for various edge case scenarios which are harder or impossible to try in a real world setting. Moreover, the digital twin allows software developers to inject anomalies and study their effect on scientific software. The what-if type of studies in digital twins can allow DOE infrastructure providers to work with the software developers and improve the validation and verification of the scientific software as well as inform new system design and procurement

This type of multi-faceted work would involve collaborations between application scientists, computer scientists, system software developers, workflow management system developers and service/infrastructure providers.

Timeliness or maturity: We now have a number of technologies and methods at our disposal to push the automation of the software development and testing process. There are opportunities for automation not only to assist in the software development itself but also in the process of testing the work at scale. In particular, advances in AI methods and tools that have been developed in industry can be applied and potentially extended to automate the identification of anomalies at the application-level and at the cyberinfrastructure level [2,3]. Workflow management systems have been used to automate the execution of interdependent science applications operating at scale on large data sets and utilizing distributed computing platforms to generate results in reasonable amounts of time.

The impact of automating the development and testing of science workflows will result in better quality software and improved productivity on the part of the application developers. They will be able to focus more on the scientific aspects of their codes rather than the software testing. Additionally, automated testing could also be extended to automated validation of the software results, however, this would require that the validation tools incorporate domain-level knowledge.

References

[1] OSG: https://opensciencegrid.org/

[2] Platform for Explainable Distributed Infrastructure (PoSeiDon) Project, http://www.poseidon-workflows.org/

[3] P Krawczuk, G Papadimitriou, S Nagarkar, M Kiran, A Mandal, E Deelman “Anomaly Detection in Scientific Workflows using End-to-End Execution Gantt Charts and Convolutional Neural Networks”, Practice and Experience in Advanced Research Computing, 1-5

Utilizing Facility Data and Best Practices to Guide Sustainable, Portable (and Useful) Scientific Software Development

Jack Deslippe, Nicholas Wright, Rechard Gerber
National Energy Research Scientific Computing (NERSC),
Lawrence Berkeley National Laboratory

Challenges

There are a number of challenges in “Scientific Software Development” that are unique from, for example, commercial software development. Here we highlight two of the most impactful challenges:

1. Because scientific software is typically used at HPC (high performance computing) facilities, software developers don’t typically have direct access to data on how their end users are utilizing their applications. In fact, they often lack even basic information about things as simple as how frequently their applications are used.

   This situation is even more complex for library and tool developers whose software may be utilized for only a fraction of the runtime of a typical HPC application.

2. Because of the competitive nature of scientific discovery, real world time to completion for a modeling/simulation or data analysis workflow is typically the most impactful quantity. Developers are often experts at optimizing application performance on a single system (e.g. achieved FLOP/s), but there are many more intangible factors at play in real world time to solution - for example: the ability to effectively utilize multiple HPC systems, minimizing the time to adopt a code to a new systems, minimizing the time to train a new developer.

   What is needed is a scientific way to quantify the advantages and disadvantages of different development models. For example, the community has long debated the merits of performance portable programming approaches, but quantitative metrics have also long been lacking.

Solutions Facilities can Provide

Almost by definition, HPC facilities are in the ideal position to broker the exchange of information between software developers and the user community. At NERSC, we collect a large set of data on how often applications are used, by which users and, to a certain extent, in what ways they are used and how performantly. Schedulers, link-line tools like ALTD [1] and performance counter data can, in principle, provide a rich set of data from which to guide further software development. While this data is sometimes communicated upstream to software teams, at this point, it often represents a missed opportunity.

To more fully take advantage of this opportunity, we propose the standardization of both how such data is shared as well as the development and integration into scientific software of a standardized *utilization* API: a set of low-overhead callback functions that scientific software developers can utilize in a customized way to gather usage data from their HPC users.

Facility operators would be tasked with implementing the standardized data collection API and exposing the collected data in a democratic way.

Facilities have other expertise to share as well! Due to the consistent deployment of HPC systems in an evolving technology landscape, HPC facilities like NERSC have developed expertise in transitioning applications to new systems.

At NERSC, this program is called the NERSC Exascale Science Applications Program (NESAP). Through this program, we have developed a consistent method for application development and optimization. The methodology involves a workflow of collecting application use-cases/input-decks from the user community (something the facility is well situated to do), collecting and analyzing application profiles at appropriate scales and addressing code bottlenecks in an unbiased way.

During the analysis phase, it has proven important to analyze performance in an absolute sense. While improving the performance of an application by, for example, a factor of 2 is always nice, it doesn't provide any information about how much better an application could in principle run on a system. What is required is an understanding of performance against all the relevant ceilings on the system (CPU capability, Cache/Memory bandwidths, Network latencies and bandwidths etc.). This informs developers which avenues of optimization will prove most profitable - e.g improving their vector/warp utilization vs. minimizing their data movement. Most importantly, it allows software developers to know when they have reached a point of diminishing returns.

At facilities like NERSC, we use the roofline and extended roofline performance models [2-5] to guide the application development and optimization process. We've found that this performance model provides a unique balance between simplicity and the right actionable information to guide development.

We also believe that this approach can provide quantitative guidance around less immediately tangible questions. For example, we've shown that the roofline modeling approach provides a way to quantitatively measure "performance portability." [6] **We believe a quantitatively guided approach, like the one developed and promoted by NESAP (and similar programs at other facilities) is what is needed to address the challenges laid out above.**

References

[1] Hadri, Bilel, and Mark Fahey. "Mining software usage with the automatic library tracking database (ALTD)." *Procedia Computer Science* 18 (2013): 1834-1843.
[2] S. Williams, A. Waterman, D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Communications of the ACM (CACM), April 2009
[3] Doerfler, Douglas et al. "Applying the roofline performance model to the intel xeon phi knights landing processor." In *International Conference on High Performance Computing*, pp. 339-353. Springer, Cham, 2016.
[4] Koskela, Tuomas et al. "A novel multi-level integrated roofline model approach for performance characterization." In *International Conference on High Performance Computing*, pp. 226-245. Springer, Cham, 2018.
[5] Yang, Charlene, et al. "Hierarchical roofline analysis: How to collect data using performance tools on intel cpus and nvidia gpus." *arXiv preprint arXiv:2009.02449* (2020).
[6] Yang, Charlene, et al. "An empirical roofline methodology for quantitatively assessing performance portability." In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 14-23. IEEE, 2018.

High Productivity Data Analysis Infrastructure for Scientific Discoveries

Bin Dong (dbin@lbl.gov), Kesheng Wu (kwu@lbl.gov)
Lawrence Berkeley National Laboratory

1. Challenge:

This position paper focuses on the problem **"what unique changes in scientific software development for data analysis would improve productivity, accuracy, trust, and reliability?".** Take particle physics data analysis as an example [1]. A popular simulation code named Vector Particle-in-Cell (VPIC) simulation produces data at terabyte scale, which include data about both fields and particles. Both the field data and particle data can further have many time steps and many subspecies. One simple operation on the VPIC data is to subsample them and feed the subsampled one into a visualization tool, such as ParaView, to study how magnetic fields reconnect, how particles accelerate, and so on. The core operation for subsampling can be expressed with around 6 lines of C++ code. But, in order to deal with the large data files with parallel read and write operations on different time steps, different particle types and field types, the whole program for subsampling may take 200 lines, as shown by Table I. This significantly reduces productivity for domain scientists to develop their data analysis code because they need to write 30X extra code to complete this simple data analysis task. The situation is even worse for domain scientists with a long and complex data analysis pipeline, such as training an AI model for particle classification. These complex pipelines may need sophisticated load-balancing for parallelization, checkpointing for reliability, and other advanced functions to support. Even though these operations such as load-balancing and check-pointing are common to nearly all parallel computing programs, they are often reimplemented in each scientific domain separately, sometimes, with minor customization. **Each such customized program requires considerable developer efforts to produce quality programs, and additional development time, staffing resources, and the cost of maintaining, sustaining, and evolving software capabilities as computing platforms evolve.** The challenge for the Science of Scientific-Software Development community is to refactor the development process to define a **High Productivity Data Analysis Infrastructure** for Scientific Discoveries. In the past few years, we have been focusing on developing a High Productivity Data Analysis Infrastructure that can effectively support various data analysis operations but hide all underlying data management, reliability, and other tasks, eventually increasing the productivity of scientific data analysis .

Table I Comparison of the number of lines for the Core part (decimation logic) and the Whole Decimation code for VPIC data.

	Core in C++ Subsampling Program	Whole C++ Subsampling Program
Lines of Code	6	> 200

2. Opportunity

Modern data processing systems, such as MapReduce[2] and Spark[3], can be candidates for the High Productivity Data Analysis Infrastructure for Scientific Discoveries because they can bridge the data analysis operations and underlying data management tasks through a standard

and simple API, i.e, Map and Reduce, The Map and Reduce support various operations via its user-defined function and can transparently parallelize and manage data for these operations in a reliability way. The High Productivity Data Analysis Infrastructure for Scientific Discoveries can take the same approach to provide a set of standard and simple API for general scientific data analysis. Transparently, the parallelization, reliability and data management tasks can be implemented as the backend for the API. In this way, domain scientists can focus on data analysis logic and computer scientists can optimize the backend. However, as explained in the following paragraph, the MapReduce and its following systems can not support scientific data and its data analysis program development efficiently.

The FasTensor [4] follows the essential spirit of MapReduce but it supports scientific data and its data analysis operations natively. MapReduce uses the key-value list as the fundamental data structure but the scientific data are mostly stored as multidimensional arrays, such as the 2D or 3D field data in the VPIC example above. By contrast, FasTensor supports arrays and allows users to compose any analysis on top of them through its **Stencil** data abstraction. The general idea of the FasTensor has been evaluated extensively. In this position paper, we propose to extend FasTensor to be a general High Productivity Data Analysis Infrastructure for Scientific Discoveries, where new research challenges need to be addressed. **These challenges include but are not limited to full-semantic data analysis operations, cache efficiency algorithms, general communication mechanisms and interactive programming environments (like Python, Jupyter Notebook).**

3. Timeliness or maturity

FasTensor has been widely evaluated in many scientific data analysis applications and its first version is available to the public in 2021. Extending the general idea of FasTensor to be a general High Productivity Data Analysis Infrastructure can improve the productivity and reliability of data analysis program development. The new infrastructure can seamlessly connect I/O related research (such as ExaIO, DataLib and ADIOS) funded by ECP to high-level data analysis functions. **The new infrastructure can also have potential to support Stencil-based simulation code widely used in scientific software.** The success of the work may impact most on-going AI research projects, such as the autonomous driving network, which has lots of data processing requirements.

4. References

[1] Bin Dong, Patrick Kilian, Xiaocan Li, Fan Guo, Suren Byna, and Kesheng Wu. 2019. Terabyte-scale Particle Data Analysis: An ArrayUDF Case Study. In Proceedings of the 31st International Conference on Scientific and Statistical Database Management (SSDBM '19).

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI'12

[4] Dong, Bin; Wu, Kesheng; Byna, Suren, User-Defined Tensor Data Analysis (SpringerBriefs Book in Computer Science) 2021

Understanding the patterns driving the development of software to support science

M. Doucet (doucetm@ornl.gov)[1], W. Godoy (godoywf@ornl.gov)[2],
S. Yoginath (yoginathsb@ornl.gov)[2], C. Stanley (stanleycb@ornl.gov)[3]

[1] Neutron Scattering Division
[2] Computer Science and Mathematics Division
[3] Computer Science and Engineering Division
Oak Ridge National Laboratory, Oak Ridge, Tennessee

Challenge: *Understanding how scientific software is currently written in order to make it more sustainable*

In the past decades, software has taken an increasingly important role in scientific discovery. A lot of recent work has been dedicated to trying to improve the sustainability of this software. In this position paper, we argue that there is a need to better understand how the code that is developed to produce scientific results is written, by whom, and under which circumstances. We believe that a thorough understanding of the data landscape in experimental sciences is a key investment to ensure the success of future generations of scientists.

Whether it is custom software to support instrumentation or project-specific software to extract knowledge from data, the development of software is often following a more organic process as the effort gets closer to where scientific knowledge is generated. This is in contrast to software developed in the private sector, where project management approaches have been honed over the past few decades. A lot of recent work has focussed on helping scientists develop better software by following industry standards, but little has been done to understand the unique needs that drive their current approach. Knowing the environment and understanding the needs that drive the development of scientific software would make it easier to understand why a rigid project management approach often fails, and why scientific software often isn't sustainable.

Stakeholders need to recognize that high-quality software development implies a bigger investment done by specialized individuals. The scientific discovery pipeline and ecosystem has gotten more complex and requires specialized roles focussed on domain knowledge, computing, data, hardware, software. At the same time, appropriate orchestration must adapt to the business needs on the research and development spectrum. That spectrum is wide, and covers both controlled production work (100% development) to research work of individual scientists (100% research).

We see several components that need to be studied:

- What are the characteristics of scientific software projects compared to private sector software? Where do these projects fall in the spectrum between pure research activity, pure development, or pure maintenance? What are the software development practices that are best matched to the various types of scientific software projects, and how do they differ from private sector projects?
- Software is a tool to enable quantitative interpretation of observations, but unlike mathematics, which is part of our core education, scientists are not often trained in software development. Do changes need to be made in how we train scientists?

- Software developers who support research activities often come from other domains of expertise, yet domain knowledge is usually essential. What kind of staffing multi-disciplinary profiles produce the best results?
- What is the ideal lifecycle of software that starts as a one-off prototype and becomes essential to the operations of an instrument?
- How do we reconcile the fact that science is intrinsically a trial-and-error approach, whereas software development focuses on delivering functionality according to established requirements? Can we identify how the incentive structure differs?
- Experimental facilities have a large number of unique instruments. How does building and working with such instruments change the way scientists think about software?
- How do we understand the non-homogeneous landscape of software development given that scientific software is often seen as a tool to reach a goal, rather than a goal itself like a publication?

Opportunity: *The Department of Energy user facilities offer a rich ecosystem for the study of scientific software development*

The U.S. Department of Energy (DOE) user facilities are a good example of a software ecosystem that brings together various software needs and a wide range of developers and approaches to software development. Looking around the world, similar facilities have followed wildly different approaches to developing the unique software they need. Different approaches are also used within the DOE complex, with a varying degree of success. This makes those facilities a great environment to study the topics outlined here.

Forcing scientists focussed on scientific productivity to write sustainable software may not be realistic. Studies should be performed to establish the correlation between software development practices and scientific output. We should also look at the impact of scientific software as a function of quality/sustainability, according to established, specific, criteria. We believe that looking at the way DOE laboratories produce software to support science offers all the data needed to shed light on these questions.

Timeliness: *The tide of AI is here, and scientists need to adjust*

Now is the time to answer these questions and truly understand how best to help scientists at the cutting edge of their domain. Recent advances in computing and the access to machine learning packages have lowered the threshold to entry into advanced modeling methods and greatly expanded what is possible for the average scientist. The next step is to make these tools available to a wider group of people rather than a select few. Truly understanding the role of software development in scientific endeavors will be crucial to establish the curriculum the next generation of scientists should follow. Several other research areas in computing have a large impact on scientific research in general. Topics like connected instruments or integrated workflows all mean that scientists outside computer science will also be exposed to software development. Understanding how we can help the community make the most of these new tools means that we have to understand how scientific software is developed, not by the practitioners, but by the scientists in the field. There is a need to identify the skill profile needed for individuals to succeed, but also to identify the makeup of a successful science team in the age of performance computing.

Sustainability of Software Meant for Exploration

A. Dubey
Mathematics and Computer Science Division
Argonne National Laboratory, Lemont, IL 60439

November 10, 2021

It is commonly assumed that the reason why many standard practices used in enterprise software do not always fit scientific software has to do with peculiarities of science. Commonly attributed reasons are: the community is different, the incentives are different, the priorities are different, resources are scarce, and it is inherently more complex. All of these are good reasons, however, they do not present the whole picture. it turns out that some of these challenges also occur in the software being worked on in the research departments of the industry. They struggle with adoption of practices in the research departments in the same way that projects in scientific research world do. The inescapable conclusion, therefore, is that the exploratory nature of the software has a great deal to do with why a mismatch exists between promoted best software engineering practices and their efficacy in the scientific world.

A more fundamental difference between research software in the industry and software for scientific research is that imperfections are filtered out in the industrial research software during its transition to production software. Therefore, the industry, and the software engineering research it promotes and consumes, has limited incentive to understand the nuances of applying quality, productivity and sustainability practices to their research software. In the scientific research world, on the other hand, the research software is also the production software, and there is no mechanism for filtering out the artifacts of lack of quality control. A badly tested software can lead to unreliable results, which in turn can spread misinformation in the concerned scientific community and lead to non-trivial setbacks in scientific discovery. Therefore, if we want to assure validity of scientific results we have to promote quality and sustainability of scientific software. And because we cannot rely upon the incentive to do so to come from the industry, we must take on the task of understanding the needs of research software and develop best practices that match those needs.

Software engineering research has lead to great leaps in understanding how to develop quality software without significantly overrunning costs. An implicit assumption in many of these studies is that the end goals of the software in any development cycle are deterministic and known. This assumption nicely constrains the design space of the studies and allows inferences to be drawn about which practices lead to what benefits in software development. Thus it

is possible to design a study where, for instance, one can correlate the impact of requirements gathering on the outcome, or the impact of throwing more resources at a stalled project. It would be immediately obvious to anyone working on scientific software that neither study, done straight-up, is likely to lead to many meaningful inferences in their world. The study on requirements gathering impact would have to account for the possibility that requirements may change drastically during the course of the project, or that they may be undergoing changes on a nearly continuous basis. Similarly the study on resource allocations would have to account for the corresponding expertise of assigned developers and the cost of associated learning curve.

Within the world of research software for science we have plenty of anecdotal evidence that applying quality improvement and sustainability measures result in better software and therefore better science. However, some of these anecdotes can be purely opinion based. We, the human beings, can very easily persuade ourselves that the evidence is supporting whatever unconscious bias we may have started with. One person's just enough process is too much process for another and in the absence of quantitative evidence neither side is likely to be persuaded about the possibility of being wrong. We often lament that people cannot be persuaded about the efficacy of software engineering practices. The only way to do so is to present persuasive evidence obtained through well designed studies that take into account the imperatives of scientific software, and draw inferences that are meaningful in the context of scientific research. That in turn implies that we urgently need to research software quality and sustainability methods for our kind of software ourselves.

Outlining the Complexities of Scientific Software Development and Use

Caifan Du, The University of Texas at Austin

This position paper briefly illustrates the complex problem space of scientific software development and use. This problem space is complex because scientific software lies at the intersection of scientific progress and increasingly sophisticated software technologies. Consequently, purposeful activities with regard to scientific software oftentimes involve a diversity of practices and stakeholders. It is often the case that technological, advocacy, and institutional efforts toward scientific software have to grasp and account for this diversity on the ground. Years of effort in advancing software citation and identification constitute one example (e.g., [1], [2]).

Approaching scientific software development and use also need to tackle the complexity raised by their wide range of practices and stakeholders. Below, I outline the multiplicity of scientific software, its workforce, and the activities involved in its development and use.

Varieties of Scientific Software

Even just in the field of computational science, scientific software could range from single scripts to customized notebooks to multi-gigabyte software systems with sophisticated architecture [3]. There exists software purposefully developed for science (e.g., LAPACK [4]). On the other hand, there are software packages that are not primarily built for research, but researchers heavily use them (e.g., tidyverse [5] and TensorFlow [6]). Particularly, software capabilities that support computation science include those based on numerical methods, non-numerical methods (e.g., agent-based modeling), and other analytic methods (e.g., machine learning models; other statistic techniques; scientific visualization and imaging) [7]. Recognizing these varieties of scientific software is important because different development purposes and targeted user groups influence the dynamics of software development and use. For instance, software engineering researchers have studied the development of ML/AI-based applications within Microsoft, concluding that the AI domain is fundamentally different from prior software engineering domains [8]. Analogously, the development of numerical libraries and "simulation codes" [7] may have their unique patterns as well.

Diversity of Scientific Software Development Workforce

The diverse actors involved in scientific software development, partly, are the consequence of scientific software varieties. Contributors to software impactful on science could come from industry R&D, scientific laboratories, university-industry collaborations, or open source communities. Their practices could bear an influence on software and its impact on science. These contributors range from professional software developers with or without the title of Research Software Engineer, to computational scientists who write computer programs. While research software engineering requires distinct skill sets, many scientist-programmers have received varying degrees of computing education, and the excellent ones among them could be self-taught [3]. An anecdotal example is when epidemiologist Neil Ferguson and his team released a forecasting code supporting the British Government to take measures against the COVID-19 pandemic, many software practitioners voiced concerns about the quality of the modeling code [9, 10, 11, 12]. Furthermore, the maturity of scientific software and relevant practices vary a lot across scientific disciplines. To account for the diverse background of the scientific software development workforce, uniform quality standards are nearly impossible. Focused team efforts producing scientific software could prefer disciplined engineering processes evaluated by metrics; individual scientists may be expected to write data analysis code that is clear and readable to their readers and reviewers [14].

From Scientific Software Development to Use to Development

Well-developed software does not jump into users' hands. It is often underestimated the efforts taken by scientific and scientific software practitioners to disseminate their software work. Oftentimes, it

requires user training, onboarding, and even on-site assistance to facilitate adoption and active use. The settings of scientific software use are diverse, too: Group and individual users may require different setups and instructions to put the software in use. It is also common that scientific software is used in educational settings [13], included in curriculums, allowing for interaction between instructors and students. Finally, the patterns of scientific software use can inform the subsequent development and iterations. But to capture the knowledge about use patterns demands the awareness of user constituencies and their varied use scenarios.

Toward the Science of Scientific Software Development and Use

In summary, approaching scientific software development and use need to account for:

- Multiple classes of scientific software: numerical, simulation, ML/AI, visualization, etc.
- Multiple classes of developers: RSEs, professional engineers, computational scientists, voluntary contributors, etc.
- Multiple classes of users: individual researchers, scientific collaborations, educators, students, etc.

These multiplicities imply that scientific software development and use are not only relevant to software teams and collaborations, but to distributed contexts where scientific software is built, disseminated, and adopted. Both the broad scope and multiplicities of the problem space suggest that a universal solution is less possible, but embracing a diversity of research methods is more beneficial. Quantitative data collection and analysis and qualitative observations and interviews are both useful to systematically study software engineering practices and practitioners' experience (e.g., [8]). A science of scientific software development and use can benefit the most by embracing diverse research approaches as well as accounting for the multiplicities of its reality.

References

[1] Katz, D. S., Bouquin, D., Hong, N. P. C., Hausman, J., Jones, C., Chivvis, D., ... & Zhang, Q. (2019). Software citation implementation challenges. *arXiv preprint arXiv:1905.08674*.

[2] Research Data Alliance/FORCE11 Software Source Code Identification WG, Allen, A., Bandrowski, A., Chan, P., Di Cosmo, R., Fenner, M., Garcia, L., Gruenpeter, M., Jones, C. M., Katz, D. S., Kunze, J., Schubotz, M. & Todorov, I. T. (2020). Use cases and identifier schemes for persistent software source code identification (V1.1). *Research Data Alliance*. https://doi.org/10.15497/RDA00053

[3] Morin, A., Urban, J., Adams, P. D., Foster, I., Sali, A., Baker, D., & Sliz, P. (2012). Shining light into black boxes. *Science*, *336*(6078), 159-160.

[4] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., ... & Sorensen, D. (1990, October). LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (pp. 2-11).

[5] Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D. A., François, R., ... & Yutani, H. (2019). Welcome to the Tidyverse. *Journal of open source software*, *4*(43), 1686.

[6] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. Software available from tensorflow.org.

[7] Leroy, D., Sallou, J., Bourcier, J., & Combemale, B. (2021). When Scientific Software Meets Software Engineering. *Computer*, *54*(12), 60-71.

[8] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., ... & Zimmermann, T. (2019, May). Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 291-300). IEEE.

[9] Boland, H., & Zolfagharifard, E. (2020, May 16). Coding that led to lockdown was "totally unreliable" and a "buggy mess", say experts. The Telegraph.

[10] Denim, S. (2020, May 10). Code Review of Ferguson's Model. The Daily Sceptic.

[11] Hudson, C. (2020, May 9). Of bits, bugs and responsibility in the public square | Chris von Csefalvay. Chris von Csefalvay: Bits and Bugs.

[12] Hinsen, K. (2020, May 18). An open letter to software engineers criticizing Neil Ferguson's epidemics simulation code.

[13] Rude, U., Willcox, K., McInnes, L. C., & Sterck, H. D. (2018). Research and education in computational science and engineering. *Siam Review*, *60*(3), 707-754.

[14] Hinsen, K. (2013). Software development for reproducible research. *Computing in Science & Engineering*, *15*(04), 60-63.

The plural of anecdote is data: Targeted studies and broad surveys

Ben Dudson (dudson2@llnl.gov), Alex Friedman (friedman1@llnl.gov)
Lawrence Livermore National Laboratory

Challenge: Software development has evolved into a social activity, the success of which depends on the backgrounds, personalities and motivations of the people involved. This context of scientific software development differs significantly from those in proprietary or open-source development, where the software development process has been studied for many decades [Brooks]. Within the Computational Science and Engineering (CSE) community, targeted studies should address the aspects of policy and technical decisions that influence the sustainability and adaptability of software; alongside this, a concerted effort to gather representative data in a standardized form which can be analyzed across communities and over time would advance this emerging field and inform decision-making.

Community-wide studies have been carried out by e.g. [Hasselbring], surveying publications which cite openly available software repositories (e.g., on Github), and repositories which cite publications. That study found wide differences among communities, and trends in software lifetime (a proxy for sustainability). That study did not control for the relative sizes of research communities, and so was limited in the conclusions which could be drawn. More use could be made of publicly available data such as Github repository contributions to study community interaction and development, and potential links to software sustainability. Complementing this quantitative analysis, surveys and qualitative analysis provide insight and suggest directions for policy and training. Studies to date e.g. [Stodden] have been limited in their breadth but could be extended to provide a broader sample of the community, and to understand differences between communities.

An interesting and important aspect of this emerging field is to understand how the high-level design of software influences the interactions within and between the teams using it, and how team organization affects the software produced. The way that organizational and funding structures are designed may influence the architecture of the resulting software, which in turn may affect the community that forms (or doesn't form) around it, and its long-term sustainability. This is a variation on Conway's law [Conway], an adage that "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." This has been the subject of systematic studies [MacCormack, Nagappan]. These studies focused on proprietary software contexts, where the management structure and team direction are clearer than is typically the case in scientific software development. The two-way interaction between technical software design and organizational structure, is a specific topic which if addressed in the context of scientific software development would have both technical and community-wide impacts.

Opportunity:

To maximize its usefulness for scientific investigation, quantitative and qualitative (survey) data gathered should be representative of the community. Github has risen in importance as a community resource over the past decade, and is now a source of rich data which should be studied. Opportunities for gathering such data more broadly, including from communities which don't use Github, include funding applications and journal submissions; in particular, journal papers should be accompanied by a specification of the software environment used in a machine-readable format. In addition to aiding reproducibility, this could provide data on the use of software packages, which are usually unacknowledged in papers (journals should be encouraged to ask for such acknowledgements). Funding

applications could include a software sustainability plan, with a free form component for referee, and a machine-readable form to aid in automatic gathering and analysis. Funding agencies should recognize the importance of their communities' code bases, and factor in long-term support as warranted.

Policy and funding structures have already had a significant impact on the field: The ECP initiative and IDEAS process have resulted in advances in the availability and widespread use of scientific software libraries, in particular those focused on solving mathematical problems abstracted from the problem domain (e.g. linear algebra) and support infrastructure (e.g file input/output). Application software, however, typically remains localized within small communities, even where those communities share common underlying libraries and frameworks. Addressing this issue will require re-examining the design of application codes, and considering how they could be structured in less monolithic and more modular ways to promote reuse. This requires input from social scientists and policy experts to understand the social structures; software engineers and domain experts to define robust design patterns and APIs; and applied mathematicians to further develop techniques for, e.g., coupling of ODE and PDE systems, *in situ* data reduction, and other processes.

TImeliness:

In the broad computational science community, conferences and journals are increasingly requiring artefacts (data, code) to accompany submissions. This represents an opportunity that should be supported by funding bodies, for example the Wellcome Trust's open access initiatives. There are also existing international collaborations which aim to study the process of scientific research such as the Research on Research Institute (https://researchonresearch.org/) which can provide tools and experience to use as a starting point for further investigations into computational scientific research.

In Fusion Energy Sciences there is a growing move towards developing and adapting software to directly support the mission of designing and operating fusion reactors, both private and public. This is driving an acceleration in the complexity of software models, and the size of the teams required to build them. The study of existing community frameworks can shed light on the interaction between community and software structure, and ensure that the resources invested into the next generation of scientific software has the biggest impact possible.

This work was performed under the auspices of the U.S. DoE by LLNL under Contract DE-AC52-07NA27344.

References

[Brooks] Fred Brooks (1975) "The Mythical Man-Month: Essays on Software Engineering" ISBN 0-201-00650-2

[Conway] *Conway, Melvin (1968). "How do committees invent"*.

[Hasselbring] W. Hasselbring, L. Carr, S. Hettrick, H. Packer and T. Tiropanis, "Open Source Research Software," in *Computer*, vol. 53, no. 8, pp. 84-88, Aug. 2020, doi: 10.1109/MC.2020.2998235.

[MacCormack] MacCormack, Alan; Rusnak, John; Baldwin, Carliss Y. (2011). "Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis" . SSRN Working Paper Series. doi:10.2139/ssrn.1104745.

[Nagappan] Nagappan, Nachiappan; Murphy, Brendan; Basili, Victor (2008). "The influence of organizational structure on software quality". Proceedings of the 13th International Conference on Software Engineering - ICSE '08. New York, New York, USA: ACM Press: 521. doi:10.1145/1368088.1368160

[Stodden] Victoria Stodden (2010) "The Scientific Method in Practice: Reproducibility in the Computational Sciences" MIT Sloan Research Paper No. 4773-10 http://dx.doi.org/10.2139/ssrn.1550193

Put an end to just in time planning

Contributors: Kjiersten Fagnan, DOE Joint Genome Institute, Berkeley Lab (kmfagnan@lbl.gov); Jeffrey Johnson, Cohere, LLC (jeff@cohere-llc.com)

We've all experienced it. The insidious ping or notification flashing past our screen indicating a new email or message. When deep in thought or focused on a task, those are pretty easy to ignore, however, when those pings might contain guidance that could preempt a current task, you feel *compelled* to look at them. This context switch is a productivity killer.

While current practices differ for scientific software development across the Department of Energy (DOE) complex, it is easy to identify certain problematic patterns. One core issue is the manner in which staff are allocated to projects. As staff progress in their careers within the national labs, they will inevitably find themselves split across 5 or more projects, leading to high levels of context switching, and significant challenges in project management. Context switching negatively impacts productivity; this impact is well documented[1]. Humans are not as efficient as computers at multitasking. Every time your work is disrupted it takes on average 5-30 minutes to get back to the original task[2].

More projects require more meetings to coordinate. These meetings consume significant time that could otherwise be spent on productive technical work. These additional meetings can erode staff morale, who must spend time planning and discussing things abstractly, rather than actually working toward tangible results.

The challenges of context switching, multitasking, and just in time work allocation can be addressed through changes in project management, planning, and funding. In order for staff and collaborations to be effective, there must be a well-defined engineering process supported by the appropriate roles and clear responsibilities[3].

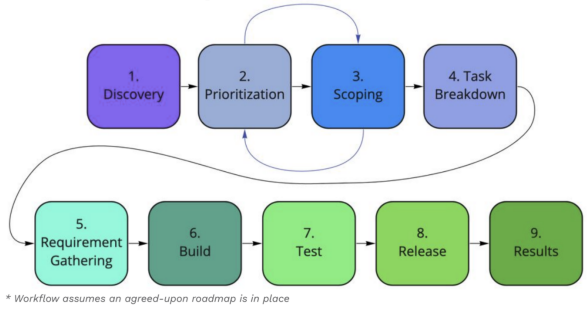


Figure 1: Idealized software delivery workflow

In this figure we show an idealized software delivery workflow that begins with a discovery phase where the team engages in user research to identify the key problems and ideate on possible solutions. This is followed by a prioritization and scoping phase where elements like a minimum viable product (MVP) are identified. The highest priority items are broken

[1] Newport, C. (2016) Deep Work: Rules for Focused Success in a Distracted World. Grand Central Publishing.

[2] https://www.petrikainulainen.net/software-development/processes/the-cost-of-context-switching/

[3] Hukkelberg I., Berntzen M. (2019) Exploring the Challenges of Integrating Data Science Roles in Agile Autonomous Teams. In: Hoda R. (eds) Agile Processes in Software Engineering and Extreme Programming – Workshops. XP 2019. Lecture Notes in Business Information Processing, vol 364. Springer, Cham. https://doi.org/10.1007/978-3-030-30126-2_5

down into tasks that become issues that feed into what is likely a more familiar development cycle - identify requirements, build, test, release.
Here we define some key roles that we have seen make research software engineering teams more productive and effective. Note that role does not need to be a permanent job function and can potentially rotate to different members of the team.

Product manager - gathers and translates business goals from SMEs. Collects and organizes internal and user feedback. Has the authority to make prioritization decisions at a more detailed level. Communicates project priorities and roadmaps regularly to the development team and internal and external stakeholders (PIs, other leads, DOE). Develops a roadmap that aligns all cross-team activities.
Delivery Manager - monitors project staffing and support, suggesting changes to the cadence as needed. Facilitates cross training and sharing of resources and time where needed. Directs information to the proper channels when challenges arise with a project or process, managing within the team or raising concerns to leadership.
Designer - works within teams as an embedded member to assist in the design of new features. Performs user research to understand usability challenges, communicating them in meetings to establish priorities and scheduling.
Subject Matter Expert (SME) - possesses expertise specific to the project's domain. In some cases, a SME is a PI or other person with scientific expertise; in others a software engineering expert.

A successful project makes the best use of its resources. In a scientific context, this means that scientists should do science and not project management. They must use their expertise to perform technical work, evaluate results in the context of their intended purpose, and ensure that the project produces scientifically valid and meaningful results. A scientist can play the role of a scientific director, product owner, or liaison with the scientific community, but should not "own" large infrastructure projects.

Well-established approaches exist in both the public and private sectors for building high-functioning software teams that deliver solutions to real scientific problems. DOE SC should prioritize and support the adoption of agile, user-centered design for research software engineering. It must be recognized, however, that no single one-size-fits-all protocol exists for "agile" development. An adept product manager can adapt the tools and processes to the needs/culture of a specific team. Clear project orchestration sets the team members up for success and higher productivity because each day of the sprint, they know exactly what they're meant to work on. That is, they can ignore the pings and focus on building the infrastructure science needs.

Coercive Software Process Improvement – Addressing the Imperative from the Bully Pulpit

Andy Gallo (andrew.gallo@ge.com), Eric Tucker (eric.tucker@ge.com) – GE Research

Challenge: Recognizing, as evidenced in organizations and events including this call for papers, that software development is an increasingly important yet under-studied component in the pursuit of scientific discovery, the workshop organizers ask, "*What methods either uniquely apply or uniquely do not apply to the development of scientific software?*" Such questions intend to generate other questions, for instance "what development methods generally applied also apply to the development of scientific software?"

In short, our position is that nearly all aspects of a generalized domain-agnostic software engineering process also apply to the production of scientific software, and that *most* of the benefit to be gained in studying improvements in scientific software development can be had simply by adopting existing generalized techniques. At some level of complexity-managing abstraction there's little discernable difference between developing for example robust supply chain management (SCM) software and a HEP simulator. Upon demonstrating the abundant cross-domain software similarities and to provide rigor we'd then offer numerous studies showing it *pays* to learn the practice of writing good software.[1] Unfortunately defending these claims or arguing the counter can be difficult. Even among the converted, priority resources specifically time and talent must be allotted. This is the challenge and the current limitation of the state of the art – there is no *force* moving the needle on software process adoption. Preferably, a prioritization of the urgent problem would be openly and organically acknowledged by the scientific practitioners. A desire to change incrementally and continuously declared. A willingness to trust in the outcomes avowed.

Or the government can provide deliberate and enforceable encouragement.

Opportunity: Procurement of government contracts which contain a critical software component, as most do, can come with certain procedural requirements informed by Federal GSA guidelines to be followed by the awarded, for example around security, IT operations, and software development practices including those implemented with agile methods.[2] Time on a government supercomputer awarded for the intent of science, where software development is a recognized important component, might similarly impose software process requirements. This forces the issue – teams desiring of time on the hardware must be able to speak to the maturity of their software, beyond just inquiring on the grant application if the codes are "ready for GPU".[3] Precedent exists for such requirements, as evidenced by the broad adoption of Model Based Systems Engineering practices, and the subsequent flow-down of requirements as part of DOD procurement and sustainment.

As agile afficionados have successfully argued [4], the software development process utilized by the scientific team does not need to be heavyweight. What does need to be true is simple – have a process and follow it. Any standards, themselves to be debated, would be general enough to be applicable broadly, light enough to be adopted widely, and common to all federated facilities granting time on leadership-class machines, reinforcing the definition of

"leadership". As is the case with MBSE, the government need not prescribe a singular specific framework or approach – existing standards bodies can serve the purpose of validating minimum requirements and practices. This is the "bully pulpit" approach often used by government, especially when attempting to stimulate change in an otherwise free market of ideas – for example, the public need not care if you drive an electric car or pedal a bicycle, but they assert you need to be emission-free by some date certain.

Left to their own devices, and for various very good reasons, like consumers of fossil fuels the scientists will not move in this process improvement direction on their own. Therefore, there needs to be some, shall we say, encouragement. It is suggested that the process standards be based on flexible and agile approaches, but the compute time grant application process should imply that the team's attention to software engineering as it pertains to the computational science is as important as the science itself.

Timeliness: With the end of ECP in sight, thoughts turn to the next chapter in this journey. How will the lessons so far learned be transferred and adopted – from the novel hardware architectures to the software technologies and applications, to also the software engineering practices and tooling which were so vital to the co-development? If the next grand challenge in scientific computing is not about increasing the scale of compute, but instead the workflow and process enablement of complex scientific modeling tasks, how can we ensure that going forward any strategic leadership-class program, including and especially on dynamic cloud-centric platforms, continues to hold software processes and technologies in critical focus? One way is through further transference of *responsibility* to the general community, and that might include the mandating of minimum standards.

The good news might be that teams at the national laboratories, having been critical to the success of the ECP and the rollout of new computing platforms and software technologies, are now also (in theory) expert in the supporting software engineering processes, and available to help evangelize and support the new technology *and* process adoption mission.

The impacts of success are potentially numerous. Wouldn't a research team which employs a proven software engineering process have a better chance of adapting their code in a timely fashion to new emergent hardware architectures? To new software stacks which algorithmically provide higher levels of performance? For novel code to be developed more quickly and with fewer defects? To be capable of sustainably producing repeatable scientific results over time? These are national strategic imperatives.

References:

1. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/developer-velocity-how-software-excellence-fuels-business-performance
2. GAO Agile Assessment Guide: https://www.gao.gov/assets/gao-20-590g.pdf
3. ERCAP form: https://www.nersc.gov/users/accounts/allocations/request-form/
4. Agile Manifesto: https://agilemanifesto.org/principles.html

Automating Software Integration for Sustainability

Todd Gamblin, Lawrence Livermore National Laboratory, tgamblin@llnl.gov

1 Challenge

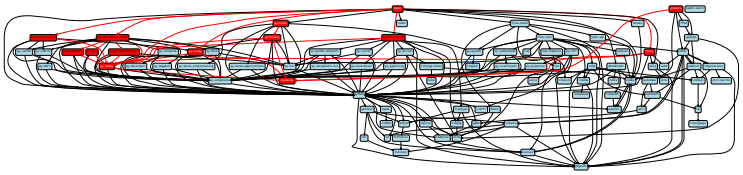


Figure 1: MuMMI Dependency graph with conflicts highlighted.

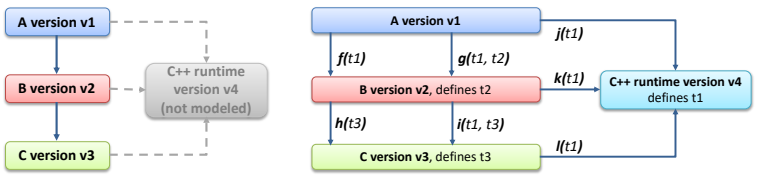
Scientific software development, particularly in HPC, differs from regular software development in the timespans and complexity of the software being implemented. HPC software *must* take advantage of the most bleeding edge platforms while integrating packages that span generations of scientists and engineers. Modern HPC codes contain libraries implemented in C, C++, Fortran, Python, Lua, and other languages, and they leverage programming models spanning multiple compilers and hardware devices, including MPI, pthreads, OpenMP, CUDA, HIP, SYCL, and others. As an example, the MuMMI [1] code shown in Figure1 comprises 98 packages and 248 dependency edges. This package integrates C, C++, Fortran, Python, and Lua packages by conduct fine-grained MD simulations, analyze them and build surrogate models using ML, and finally integrates the surrogate models to perform coarse-grained simulations. The packages in this code span decades – some are AI packages developed in the past 10 years; others trace their lineage to the early days of HPC.

The challenge faced by developers of MuMMI and codes like it is the constant need to port this broad software stack to new machines. Even porting from Sierra to Summit, two machines with the same Power9/Volta architecture, required hundreds of hours of integration time, due to incompatibilities with system libraries. One such incompatibility is highlighted in the graph; the authors had to iterate repeatedly to build versions of `gromacs`, `faiss`. and `numpy` that were compatible with the system `openblas`. This is not an uncommon problem – porting to a new HPC stack can often take days or weeks due to the many inbuilt assumption in optimized software stacks. Industry and the standard Linux distribution packaging it typically uses deals with these problems by standardizing on a particular platform and toolchain, and accepting any performance penalties that may incur. Each software distribution is its own ecosystem – an *island of compatibility* where only packages within the distribution can work together.

2 Opportunity

Since the mid-2000's [3], package managers have solved these types of integration problems *within* a single ecosystem using solvers from the domain of combinatorial search and optimization techniques (SAT, SMT, ASP, etc.). Distribution maintainers collect repositories of metadata about packages, their versions, and their (in)compatibilities. Unfortunately, they do this in terms of *versions*, and they ignore compatibility data that would allow cross-toolchain or cross-ecosystem solves. There is an opportunity to eliminate this restriction and build much more capable solvers to automate the tedious aspects of software integration.

Compatibility Models: Because HPC software must integrate at a very low level, often using multiple compilers and languages, traditional version compatibility models are insufficient. There is an opportunity to build much deeper models of compatibility between packages at the level of function interfaces – we can match the exit calls



(a) Simple version-based model

(b) Model with function and type information

Figure 2: Compatibility models.

of one library to the entry calls of its
dependencies, and check their types for compatibility. Instead of modeling human-imposed version constraints, we can model the actual types, function signatures, and ABI information and record this data with packages in addition to version constraints (Fig 2). This information may or may not exist in many binaries, but we can use debug information in conjunction with binary analysis to deduce it and to provide metadata for precompiled binaries. A proof of concept in this area could eventually lead to updates to standards like DWARF and ELF to include richer metadata to enable better binary interoperability.

Solvers: Just as package ecosystems have evolved over the past 20 years, so have logic solvers. Package solving is a combinatorial constraint problem with optimization, and many of the techniques for solving this problem boil down to SAT. Historically thought of as "too hard" to bother with, modern SAT solvers have evolved to solve problems with millions of constraints. The problem of package integration (with hundreds of nodes) is actually small compared to industrial search problems, and we should push its limits. New solving paradigms like Satisfiability Modulo Theories (SMT) and Answer Set Programming (ASP) enable much richer program semantics to be encoded in solvers, and we can use them to encode and solve systems with much more detailed compatibility models.

Learning from Builds: Static analysis is not the only way to deduce the likelihood compatibility of software packages. We can also learn from combinatorial builds. Recent advances in continuous integration and fuzz testing open new possibilities for simply *sampling* the combinatorial space of package builds and conducting sensitivity analysis on them just as we would for uncertainty quantification in a simulation. We can use these to build models that can further refine statically derived compatibility models with empirical information, allowing users to pick the right configuration parameters the *first* time.

3 Timeliness

U.S. exascale and pre-exascale systems span three different GPU architectures, and the diversity of computing hardware is only expected to increase. The integration challenges required by GPU stacks include complex constraints and compatibility rules, adding to the already complex compatibility constraints imposed by ABI issues in C++ and Fortran. Scientific developers are increasingly building code in high-level languages, and interoperability issues with lower-level stacks in C, C++, Fortran, and Rust are common. Even in modern software distributions, there is a growing schism with language-centric package managers – distributions do not know how to package code from languages like Rust and Go (which allow arbitrary software versions). Tooling that can model lower-level software interactions and ease development is desperately needed in both the scientific software community *and* in the increasingly diverse broader ecosystem. Many of these ideas have been prototyped in the Spack [2] package manager, but more work is needed to enable the techniques to be used broadly.

4 References Cited

[1] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. Kokkila-Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P.-T. Bremer, J. N. Glosli, and H. I. Ingólfsson. A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[2] T. Gamblin, 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.

[3] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 199–208, 2006.

Reliable, Efficient Workflows for Complex Scientific Software Development

Derek Gaston derek.gaston@inl.gov, Logan Harbour Logan.Harbour@inl.gov, Fande Kong Fande.Kong@inl.gov, Guillaume Giudicelli Guillaume.Giudicelli@inl.gov, Alexander Lindsay alexander.lindsay@inl.gov

Challenge - Gone are the days where a single scientist would create an entire code from scratch. Today's scientific discoveries often demand multiscale, multiphysics simulation capabilities that can run on everything from laptops to supercomputers. This requires a multidisciplinary team made up of domain specialists, computer scientists, applied mathematicians, hardware and software optimization experts, and infrastructure administrators. **The coordination of all this human capital for creating ground-breaking scientific software requires well-designed, partly automated workflows that enhance productivity and ensure reliability.**

Opportunity - As scientific software has continued to become more complex, so has software in other disciplines. Therefore, it is natural to look to the broader software development community for best practices on how to manage the development and maintenance of complex software systems. One of the most disruptive software development concepts over the last twenty years has been "agile" methods. However, agile development methods were only made realistically feasible by key pieces of workflow technology: continuous integration (CI), continuous delivery (CD), decentralized repositories (i.e., Git), and project management systems (i.e., GitHub, GitLab). **Borrowing these ideas and capabilities, while specializing for unique requirements in scientific software development, would allow the software to continue to grow in complexity (and therefore prediction capability) while ensuring reliability and trustworthiness.**

Most software begins as a conversation, whether it's dialogue with a customer, an internal monologue, or a group discussion. Capturing these discussions and requirements is the first step of organizing a development team to tackle a scientific problem. However, in the past, this step was not always well preserved, with notes in laboratory notebooks (or on Post-its) quickly becoming code and being left behind. In the early 2000s, the broader software community sought technological solutions to capturing requirements and design ideas; this has ultimately culminated in modern software project management websites such as GitHub and Gitlab.

The Multiphysics Object Oriented Simulation Environment (MOOSE) project [Permann 2020] headed by Idaho National Laboratory (INL) first utilized Subversion and Trac for project management, moving to GitHub in 2014 [Gaston 2014]. While GitHub initially only had rudimentary issue tracking, it has since gained capability for milestone and project planning, and more recently "Discussions." GitHub Discussions are a vast improvement over the usual scientific software staple of the "mailing list," allowing questions and discussions to immediately be turned into actionable requirements captured as tickets and providing superior capabilities for search and archival. An objective study of the degree to which scientific software makes use of such discussion and issue tracking capability, correlated to the software's success or longevity, would improve understanding of this critical step in development. The relative impact on the development and support of the user base could be quantified through analyzing search and discussion metadata, and looking for patterns in successful or unsuccessful support tickets.

While scientific software development used to rely on copying folders of code for "versioning," modern projects utilize the decentralized development capability afforded by Git. While useful even for one person, Git shows its true worth with larger software teams by allowing developers to work independently, yet efficiently integrate code changes. One aspect of utilizing Git that should be studied is the impact of various Git workflows and organization strategies on scientific software development. For instance, the MOOSE project utilizes a "fork, branch, rebase, pull-request" strategy [Slaughter 2015], but many other workflows exist. The effect of seemingly simple choices, like whether to rebase before merging, could be scientifically studied. Even though all those components exist, it is not always obvious which elements are best for a given project. Further research is required to understand optimal solutions for different-scale projects.

Relatedly, branching in Git affords new opportunities for collaboration. Developers can work collaboratively on a branch without impacting other developers or users. Studies can be made of the multiple schemes for having "development" versus "release" branches and how/when to move changes between them to gauge the success of such efforts. The complexity of some of the most common workflows should be quantified. There also exists the possibility of utilizing automated systems powered by AI, to decide when to promote changes from one branch to another.

The core MOOSE ecosystem includes the MOOSE library itself, along with over 40 interdependent applications and physics modules that are maintained by independent development teams. CI is a critical piece of technology for ensuring that the hundreds of people working on these projects remain productive. MOOSE utilizes the CIVET [Alger 2016] CI/CD capability that: automatically tests every code change before it's allowed to be merged, tests integrations of multiple code changes, updates inter-code submodule dependencies, promotes code changes to be released, builds binary packages, distributes binaries to INL HPC machines, publishes binaries via the Conda [Anaconda 2016] package management system, and updates project websites with new documentation.

Every week, CIVET utilizes over 2,500 dedicated processors to execute over 20 million tests on multiple architectures for more than 80 new pull-requests. Automated integration systems such as these can save person-years during the develop/release cycle and help ensure that bugs are not being introduced as the software continues to gain features. We are at a point now where these techniques are well understood and can be widely deployed by the scientific software development community. Still, significant challenges remain, such as test portability, testing system stability, and performance test consistency. A broad range of research activities is required to understand these issues and the efficacy of CI/CD for scientific software.

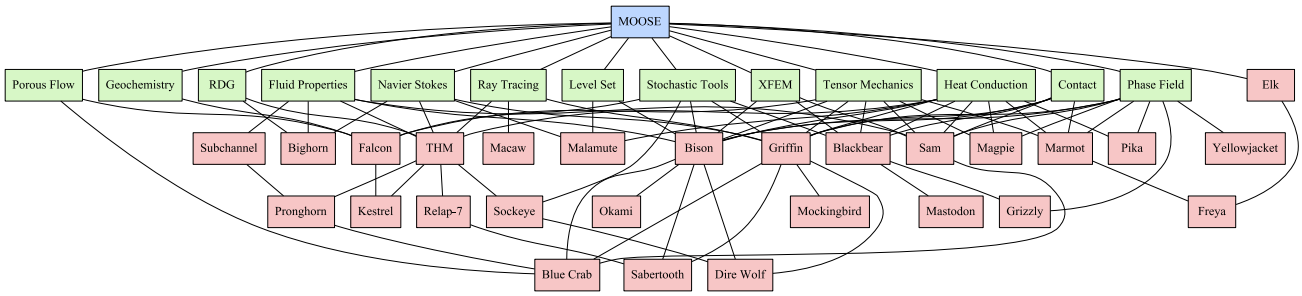


Figure 1: MOOSE ecosystem that is continually tested by CIVET. Blue: MOOSE Framework, Green: Physics Modules, Red: Applications.

As shown in Figure 1 the MOOSE ecosystem is an incredibly complex set of libraries and applications, all under active development. The historical way of dealing with dependencies was "releases," versions of code frozen in time that a downstream project would depend on. These versions would rarely, if ever, be updated. In the modern world of multiscale, multiphysics simulation, a greater degree of coordination is needed. The MOOSE team has enabled this through a system of automated Git submodule updating and branch promotion [Slaughter 2021]. When changes are proposed in a pull-request to a library an application depends on, the *application* tests are also run against those changes. The merge is not possible until all tests through the whole ecosystem pass. If a pull-request is merged, it instantly cues CIVET to rebuild and test any dependent applications. If the application tests pass, then the submodule in that application repository denoting the working version of the library is *automatically* updated by CIVET. This advanced CI/CD process plays out many times per day as changes are merged throughout the ecosystem. Study is required to determine the extent to which projects are utilizing advanced CI/CD—and how they are being used—to enable efficient scientific software development.

Timeliness or Maturity - The last 20 years have seen tremendous progress in software engineering for large, complex codebases. These advancements have been continually adopted by the scientific community, though not as broadly or as timely as they could be. The technology platforms for enhanced software development productivity, such as GitHub and CI/CD, are quite mature at this point, though more study is needed to understand their efficacy for scientific software development. In the future, these mature platforms can be utilized to address some unique challenges of scientific software development such as the large amount of inter-dependent library and application development. Ideas such as automated submodule updating, AI for generation of test cases and test specifications, continuous integration with user-generated code inputs, predictive input file editing, and extensive automated cross-platform validation are all on the horizon and can continue to ensure productivity and reliability for the next generation of scientific tools.

References - Permann, Cody J., Derek R. Gaston, David Andrš, Robert W. Carlsen, Fande Kong, Alexander D. Lindsay, Jason M. Miller et al. "MOOSE: Enabling massively parallel multiphysics simulation." SoftwareX 11 (2020): 100430.
- Gaston, Derek, John Peterson, Cody Permann, David Andrs, Andrew Slaughter, and Jason Miller. "Continuous integration for concurrent computational framework and application development." *Journal of Open Research Software* 2, no. 1 (2014).
- Slaughter, Andrew E., John W. Peterson, Derek R. Gaston, Cody J. Permann, David Andrs, and Jason M. Miller. "Continuous integration for concurrent MOOSE framework and application development on GitHub." *Journal of Open Research Software* 3, no. INL/JOU-15-34179 (2015).
- Alger, Brian, et al. *CIVET: Continuous Integration, Verification, Enhancement, and Testing*. No. CIVET; 005300MLTPL00. Idaho National Lab.(INL), Idaho Falls, ID (United States), 2016.
- Slaughter, Andrew E., Cody J. Permann, Jason M. Miller, Brian K. Alger, and Stephen R. Novascone. "Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance." *Nuclear Technology* (2021): 1-8.
- *Anaconda Software Distribution*. Computer software. Vers. 2-2.4.0. Anaconda, Nov. 2016. Web. <https://anaconda.com>.

The US-RSE Association: Accelerating Changes in the Scientific-Computing Community to Improve Practices for Software Development and Use

Sandra Gesing[1], Jeffrey C. Carver[2], Ian Cosden[3], Julia Damerow[4], Charles Ferenbaugh[5], Chris Hill[6], Daniel S. Katz[7], Christina Maimone[8], and Lance Parsons[3]

[1] University of Illinois Discovery Partners Institute, sgesing@uillinois.edu
[2] University of Alabama, carver@cs.ua.edu
[3] Princeton University, icosden@princeton.edu and lparsons@princeton.edu
[4] Arizona State University, jdamerow@asu.edu
[5] Los Alamos National Laboratory, cferenba@lanl.gov
[6] MIT, cnh@mit.edu
[7] University of Illinois at Urbana-Champaign, dskatz@illinois.edu
[8] Northwestern University, christina.maimone@northwestern.edu

1. Challenges

The importance of research software and its sustainability has gained increasing traction in the scientific-computing community in the last decade, evident in rising numbers of job offerings for Research Software Engineers (RSEs) [1], of events addressing RSEs such as the RSE-HPC workshops [2] at the SC20 and SC21 conferences, and of RSE associations forming worldwide [3]. The US Research Software Engineer Association (US-RSE) [4] has grown from several dozen people to over 900 members in the last three years. Despite the increased awareness of the role of research software, there is still a long way to go to achieve the necessary cultural change in academia and to reach a critical mass within the community to improve practices for software development and use. To do this, we have identified three major groups that must be reached and equally importantly, must feel that they are part of the community and must become interested in good software engineering practices.

- The first group includes people who create and maintain research software in silos, for example, PhD students or postdocs who are not necessarily experts in software engineering and are not aware of the RSE community. They may be reached by further outreach.
- The second group is those who are aware of the RSE community but do not feel that they belong to the RSE community even though they develop research software, where further outreach is not the solution.
- The third group in the community are researchers in different disciplines and communities who perceive software as a byproduct of their research.

The challenge is to change the perception of the importance of sustaining software in such communities. Changing perception of a person or group about software or whether they belong to a group in an academic environment is a research question that cannot be addressed by a single effort but needs a multifold approach.

2. Opportunity

US-RSE has created momentum with activities such as regular events, panels, publications, presentations, newsletters, and working groups. Most of the members joining now seem to be in

the category of “I finally found a community I fit in.” This is a crucial core community for US-RSE, as they are trained or are interested in being trained in good software engineering practices. One opportunity is to learn from people interested but new to the space how to relay such practices for different disciplines. The US-RSE website is planned to be a source for good practices for software development and its various aspects. The US-RSE working group on education and training, for example, has self-organized to publish links to content for different levels of expertise and effort to learn aspects of the software engineering process.

With the increasing number of people joining from different disciplines and with different foci, there is an opportunity to analyze how to reach communities that are not currently well-represented in US-RSE. We need to explore how to reach more people outside the current communities, which addresses the first group of people presented in the challenges. For the second and third groups, providing information is a necessary step to support changing their perception about research software, but many more steps are needed to be successful. One opportunity is to connect to experts in communities who are trusted in those communities and could represent allies for US-RSE. US-RSE collects anecdotal and qualitative evidence of the importance of research software but we need to combine it with metrics and quantitative data to recruit such allies and build a network of researchers to change academic culture. Collaborating with social scientists on the data and connecting them with members as well as non-members of US-RSE can open up more opportunities to understand the dynamics better in the academic community. Specifically, US-RSE could be a treasure trove of information and willing participants for social science researchers to tap to ask *their* questions.

3. Timeliness and Metrics of Success

Activities around analyzing gaps in the community landscape of research software are timely, with the momentum that has been created by US-RSE and projects such as BSSw (Better Scientific Software). We now have more qualitative and quantitative data available than we had 2-3 years ago, resulting from activities of such projects and associations, for example, the periodic international RSE survey. Metrics of success include the increase in members of US-RSE who belong to communities who haven’t been represented before and members who state that they are not RSEs but support the goals of improving research software sustainability and use.

References

[1] https://us-rse.org/jobs/
[2] https://us-rse.org/rse-hpc-2021/
[3] https://society-rse.org/international-rse-organisations/
[4] Carver, J., Cosden, I., Hill, C., Gesing, S., Katz, D.S.(2021). Sustaining Research Software via Research Software Engineers and Professional Associations. ICSE 2021 BokSS Workshop, June 2021, arXiv:2103.01880 [cs.SE]

SGCI: Advancing Access to Scientific Software via Science Gateways Improves the Trust within the Computational Sciences Landscape

Sandra Gesing[1], Maytal Dahan[2], Linda Hayden[3], Marlon Pierce[4], Claire Stirm[5] and Michael Zentner[6]
[1]University of Illinois Discovery Partners Institute, sgesing@uillinois.edu
[2] University of Texas, Austin, Texas Advanced Computing Center, maytal@tacc.utexas.edu
[3] Elizabeth City State University, haydenl@mindspring.com
[4] Indiana University, marpierc@iu.edu
[5] University of California, San Diego, San Diego Supercomputer Center, cstirm@ucsd.edu
[6] University of California, San Diego, San Diego Supercomputer Center, mzentner@ucsd.edu

1. Challenges

Computational sciences can be defined as being "concerned with the design, implementation and use of mathematical models to analyse and solve scientific problems" [1]. While this definition includes a variety of research domains and applications, we understand computational sciences in a much broader sense in the context of science gateways. Science gateways are computational end-to-end solutions for collaborating on research topics, for sharing data, methods and knowledge at different scales, and for providing easy access to computational tools and resources – from campuses to national or international level. Quite a few mature solutions have evolved over the last 15 years - e.g., HUBzero® [2], Apache Airavata [3], Tapis [4] (see Fig. 1 for the example of nanoHUB [5]) - and the usability of existing solutions has increased significantly. Two major trends can be observed regarding usability: one is the mostly black-box approach that allows beginners in a field to use complex scientific methods without having to become acquainted with complex research infrastructures, enabling them to focus on research questions. The second trend are environments such as Jupyter that allow researchers to work with their own scripts and ease the connection to complex research infrastructures and portability. While openly shared methods and data can help to increase the users' trust in the methods and results, complex methods pose a challenge. They are often so multiplexed that it is difficult to verify whether results can be trusted and whether the transformation from a theoretical model into a computational simulation is correct [6].

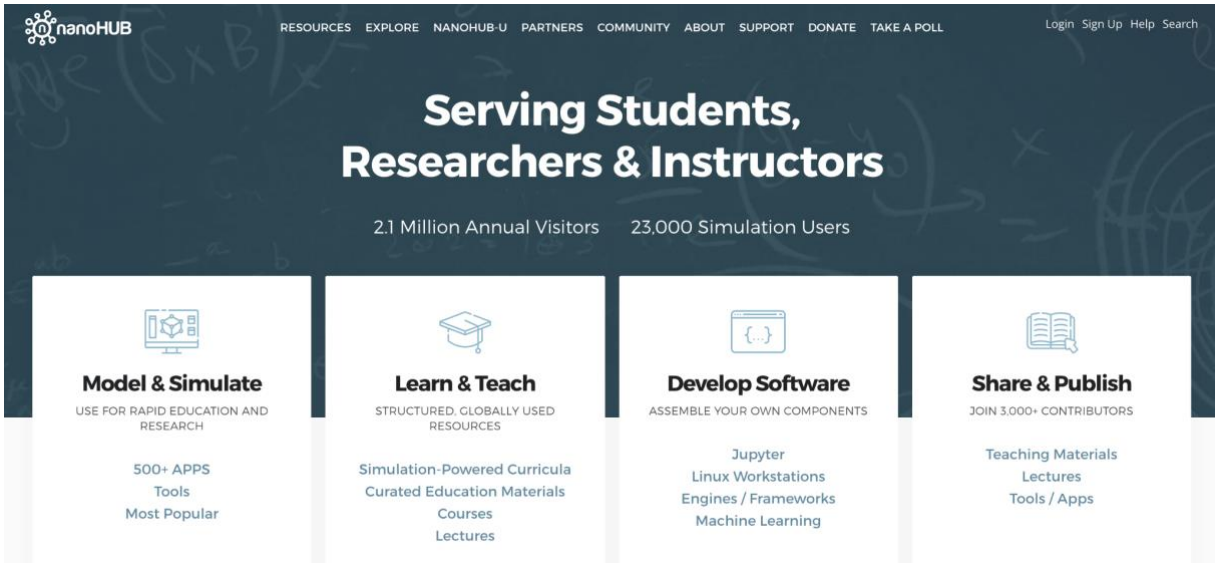


Fig. 1: nanoHUB is a science gateway for the nanotechnology community that offers a diverse set of features for integrating scientific software and teaching the content of theoretical models.

2. Opportunity

While verification between a theoretical model and the computation is a challenge that cannot be solved for every use case with one specific method, necessary steps can be supported via science gateways. The Science Gateways Community Institute (SGCI) [7] serves the community of users, developers and providers of science gateways to achieve a more efficient research process as well as a more efficient computational solutions development process. Experts advise on different aspects of creating and maintaining science gateways. Through an engagement, experts consider existing science gateway frameworks, user-centered design, cybersecurity, and sustainability of the science gateway. Science gateway frameworks generally incorporate well-defined building blocks and support crucial operations such as connecting to research infrastructures, invoking and monitoring jobs as well as visualizations of results. They allow for reproducing results of shared methods and data, which addresses the challenge of verifying results for trusted methods and forms a starting point for understanding a scientific method. The opportunity to understand a scientific method better is adding information about the expected results for different input parameters and data. Adding visualizations of results can also support a better understanding. Another method for improving the insights into the theory are simulations for intermediate steps that include methods that are trusted and verified. Extensive documentation – if applicable with formulas – can address partially to increase trust, especially for black-box approaches.

3. Timeliness and Metrics of Success

Novel applications and methods applying machine learning methods, artificial intelligence and virtual reality are exciting and allow us to address research topics that could not be addressed in the past. They also open new possibilities in the computational sciences. Users applying them for their research though, cannot be expected to understand all the details of the underlying algorithms and theory. Supporting the integration of such complex concepts into science gateways will help to build communities around experts with measures for elucidating computational methods and results. Metrics for success would include novel science gateways with users of computational methods that they did not use prior to the existence of the gateways, and their ability to reason about why they are using such methods from the perspective of trusted results. For users who prefer the black-box model, the trust can be built via proven methods in a trusted network of experts. With the fast evolving computational sciences landscape, such measures can provide valuable starting points for in-depth verifications.

References

[1] Computational Sciences https://www.nature.com/subjects/computational-science

[2] McLennan, M., and Kennell, R. (2010). HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering. Computing in Science & Engineering, vol. 12, no. 2, pp. 48-53, DOI: 10.1109/MCSE.2010.41.

[3] M. Pierce *et al*., "Apache Airavata: Design and Directions of a Science Gateway Framework," *2014 6th International Workshop on Science Gateways*, 2014, pp. 48-54, doi: 10.1109/IWSG.2014.15.

[4] Cleveland, SB, Jamthey, A, Padhyy, S, Powelly, J, Stubbs, J, Daniels, MD, Pierce, SA, and Jacobs, GA.. "Tapis-CHORDS Integration: Time-Series Data Support in Science Gateway Infrastructure". *Conference: Science Gateways 2019At: San Diego, CA* ().

[5] Madhavan, K., Zentner, M., and Klimeck, G. (2013). Learning and research in the cloud. Nature Nanotechnology 8, 786–789; DOI:10.1038/nnano.2013.231.

[6] Jay, C., Haines, R., Katz, D.S., Carver, J., Gesing, S., Brandt, S.R., Howison, J., Dubey, A., Phillips, J.C., Wan, H., and Turk, M. The Challenges of Theory-Software Translation. F1000Research 2020, 9:1192, doi: https://doi.org/10.12688/f1000research.25561.1

[7] Gesing, S., Wilkins-Diehr, N., Dahan, M., Lawrence, K., Zentner, M., Pierce, M., Hayden, L.B., and Marru, S. Science Gateways: The Long Road to the Birth of an Institute Proc. of HICSS-50 (50th Hawaii International Conference on System Sciences), 4-7 January 2017, Hilton Waikoloa, HI, USA, http://hdl.handle.net/10125/41919

Performance Portable Data Movement Support for MPI using C++

Sayan Ghosh[1], Anthony Skjellum[2], Purushotham V. Bangalore[3], and Andrew Lumsdaine[4]

[1]Pacific Northwest National Laboratory sg0@pnnl.gov
[2]University of Tennessee at Chattanooga Tony-Skjellum@utc.edu
[3]University of Alabama pvbangalore@ua.edu
[4]University of Washington al75@uw.edu

1 Challenge

The heterogeneous landscape of future extreme-scale architectures and the possible convergence of classic HPC simulations with machine learning workloads gives rise to the emerging category of *converged science applications*. The latest innovations in HPC system architecture have made arithmetic operations relatively cheap, but data movement overheads (both within the memory hierarchy and across the network) remains non-trivial. Additional complications arise when data movement is required between models with disparate runtimes, often requiring coarse-grain synchronization or leads to silent interoperability issues. With the advent of converged applications, it is crucial to support portable, productive and performant data movement abstractions that can integrate existing programming systems and third-party scientific toolkits. Fig. 1 highlights the criticality of abstractions in the context of converged applications. There is a growing consensus in industry

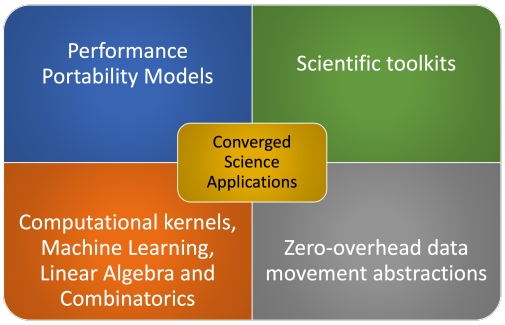


Figure 1 Converged applications need better abstractions to increase productivity on heterogeneous systems.

and research organizations on using C++ for high performance code development by targeting a variety of hardware architectures and application domains. By way of complement, MPI is supported on the widest variety of HPC platforms and extensively used by the scientific community. A large-scale study of MPI usage in open-source HPC applications [6] found that C++ was the most widely used language among open-source HPC applications. The US DOE ECP software technology capability assessment report also claims high criticality and importance of C++ and MPI [4]. C++ bindings for MPI began while MPI-1 was still being ratified [1, 7, 9, 10] and continued until the official introduction of the C++ bindings in MPI-2. However, the MPI C++ bindings were removed with the MPI-3 version because the maintenance overhead of the bindings exceeded their apparent end-user demand. C++11 paved the way for modern C++ through numerous improvements to the core language [5], heralding the new era for C++. Since then, there has been continuous enthusiasm among application developers and the MPI forum for modern C++ bindings to MPI (e.g., Boost.MPI, `mpl` [3]). As such, the performance-critical components of current performance portability models (e.g., KoKKoS, RAJA, SYCL/DPC++) are built using modern ISO standardized C++. Thus, we propose a revival of the C++ language bindings to MPI, by utilizing modern C++ (beyond C++20).

2 Opportunity

While a canonical MPI C++ interface could benefit from straightforward application of C++ standard library containers through idioms such as RAII (initialization/cleanup of resources) and inherent type safety, there are some considerations in improving the overall productivity in the context of MPI:

- Usage of modern C++ idioms, the generic container interface, operator overloading, iterators/ranges, reference counting, type traits

based conditional compilation, etc.
- Separation of MPI and primitive types, enabled by automatic type handling or parameter deduction. Runtime parameter deduction is challenging because C++ has limited support for introspection and modification of objects at runtime (a.k.a `reflection`).
- Unlike the legacy MPI C++ interface, one-to-one bindings can significantly increase the maintenance overhead, and is inimical to high level abstractions exposed by third-party performance portable models.

We are convinced that recent C++20 features such as `std::concepts` and `std::ranges` can further augment a modern C++ interface [8]. Concepts are constraints on template parameters that are evaluated at compile time and can allow emitting concise and intuitive messages upon substitution errors. Whereas C++ ranges provide interfaces to abstract a collection of items, generalizing and extending iterators. Standardizing C++ bindings for MPI would save library developers the hassle in developing and maintaining independent MPI C++ interfaces, and provide the necessary flexibility to enhance existing codes to heterogeneous partitioned-memory platforms.

3 Timeliness/Maturity

Using a compact subset of a C++17-based MPI messaging library (a.k.a `mpl-subset`), we have shown that it is possible to build zero-overhead abstractions on top of MPI [2] On the other hand, the MPI standard has recently advanced to MPI-4, with the June, 2021 approval of the latest edition of the standard. With this version of the standard, there are features that also need to be supported by a C++ interface, and for which C++ offers potential for greater performance. Key new features include:

- 64-bit API (aka "big MPI" or "big count") extends the specification by introducing new functions that replaces integer based *count* parameter with the new `MPI_Count` datatype.
- Sessions, a means for expanding the "world model" by allowing granular resource isolation for MPI programs.
- Persistent collective communication; persistence with collective operations offers a new opportunity for designing a factory method abstraction.
- Partitioned point-to-point communication. This offers channels with pipelining or concurrent producers and consumers, representing an MPI+X extension model.

We are convinced that this is an opportune moment to revive the effort in standardizing C++ language bindings to MPI, acknowledging the previous attempt as a learning experience. Along the way, we plan on actively engaging with the application and middleware developers, and encourage community participation. Both greater performance and productivity for exascale MPI applications will be the ultimate outcome of a successful effort.

References

[1] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and features. In *In Proceedings of Object-Oriented Numerics Conference (OONSCI'94)*, pages 323–338, April 1994.

[2] Sayan Ghosh, Clara Alsobrooks, Martin Rüfenacht, Anthony Skjellum, Purushotham V. Bangalore, and Andrew Lumsdaine. Towards Modern C++ Language Support for MPI, November 2021. ExaMPI'21 workshop.

[3] Heiko Bauke. MPL - A message passing library. `https://github.com/rabauke/mpl`.

[4] Michael A Heroux, Rajeev Thakur, Lois McInnes, Jeffrey S Vetter, Xiaoye Sherry Li, James Aherns, Todd Munson, and Kathryn Mohror. Ecp software technology capability assessment report. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2020.

[5] Danny Kalev and Bjarne Stroustrup. The Biggest Changes in C++11 (and Why You Should Care). *Smartbear. http://blog. smartbear. com/c-plus-plus/the-biggestchanges-in-c11-and-why-you-should-care*, 2011.

[6] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A large-scale study of MPI usage in open-source hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. ACM.

[7] B.C. McCandless, J.M. Squyres, and A. Lumsdaine. Object Oriented MPI (OOMPI): A class library for the Message Passing Interface. In *Proceedings. Second MPI Developer's Conference*, pages 87–94, 1996.

[8] Marius Iulian Mihailescu and Stefania Loredana Nita. New features in c++ 20. In *Pro Cryptography and Cryptanalysis with C++ 20*, pages 125–134. Springer, 2021.

[9] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan Doss. Explicit parallel programming in c++ based on the message-passing interface (MPI). In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pages 465–506. MIT Press, July 1996.

[10] Anthony Skjellum, Diane G. Wooley, Ziyang Lu, Michael Wolf, Purushotham Bangalore, Andrew Lumsdaine, Jeffrey M. Squyres, and Brian C. McCandless. Object-oriented analysis and design of the message passing interface. *Concurr. Comput. Pract. Exp.*, 13(4):245–292, 2001.

The Unfledged State of Heterogeneous System Testing

Ganesh Gopalakrishnan (**contact**), Ignacio Laguna, Ang Li,
Pavel Panchekha, Cindy Rubio-Gonzalez, Zachary Tatlock
ganesh@cs.utah.edu, ilaguna@llnl.gov, ang.li@pnnl.gov,
pavpan@cs.utah.edu, crubio@ucdavis.edu, ztatlock@cs.washington.edu

CHALLENGE: Drawn by the promise of increased parallelism and reduced energy consumption, the HPC community has jumped in with both feet into the bandwagon of heterogeneous computing that will involve increasing uses of tightly coupled CPUs, GPUs and other accelerators. This will also be the era where another form of heterogeneity---HPC combined with ML---will be in vogue, with each using its own type of hardware and number representations. It is critical that we attain the capability to confidently design, debug and deploy such systems within the next five years or less. Yet, the community is in an extremely unfledged state with respect to software tools for verification and testing. We now list some sobering facts and paths to success via community mobilization.

Inadequacies in Verifying Concurrency: A parallel shared memory program is incorrect if it harbors data races. Yet, *there isn't a single usable data race checker for GPUs-- commercial or academic*. For instance, the highly useful CudaMemCheck from NVIDIA does catch some memory issues and shared-memory races but ignores all global memory races. A tool that provides automated and incremental data-race related feedback while one writes a kernel can dramatically improve designer productivity.

Given that the number of loosely coupled memory spaces will increase with rising heterogeneity, it is high time that both the development and teaching of weak shared memory models be emphasized. Tools must also be urgently developed to verify proposed synchronization and communication idioms. The sad reality is that the state of shared memory consistency models is in a schizophrenic state both with OpenMP and CUDA. The unreliable legacy practices of mixing C-volatiles (deprecated by the C language committee for the danger of issues such as word-tearing) is mixed with an aspirational mention (without uptake evidence) of the C++11 standard.

Inadequacies in Verifying Numerical Aspects: The plethora of number systems and the inability to shoehorn a number system designed for ML for use by HPC is talked about in a highly aspirational way without hard science backing such discussions [1]. For example, an HPC developer is reluctant to embrace single-precision with its 23 mantissa bits; it will be a hard sell to give them BF16 with 8 mantissa bits (and the same exponent bits). The execution dynamics of today's systems are very poorly understood. We have recently conducted dynamic analysis of widely used DOE (procurement-related) proxy apps, only to show that they silently generate hitherto unknown exceptions such as Infinity and NaN. All this is happening amidst the sobering fact that *leading GPUs have no facilities to catch such exceptions in hardware.*

While discussions with respect to reduced precision abound, very few know the safety margins with respect to the dynamic ranges in today's deployed systems, let alone the unbuilt future systems. Root-causing for numerical error is hard, given that compiler optimizations, data races, or combinations can affect it.

OPPORTUNITY:
* Formal Methods: In the Souper project (https://github.com/google/souper), new instruction sequences are synthesized using SMT solvers. Also, compiler LLVM optimizations are routinely being checked using Formal Methods in the Alive Compiler Explorer (https://alive2.llvm.org/ce/) project. These are ripe for adoption in HPC. Likewise, in the Herbie project (https://herbie.uwplse.org/), one can see the seeds of promising methods for improving numerical accuracy through expression rewriting. Adapting these methods for HPC to handle complex array- and tensor-oriented expressions will give rise to methods for "repairing" numerical code after porting to new platforms.

* Fuzz-Testing Methods: The curious lack of use of fuzz-testing (highly successful within traditional software testing) within HPC points to two things: (1) HPC system components have broad interfaces that are also poorly specified. This makes it impossible to fuzz inputs without crashing the system. (2) Collaboration with domain scientists is essential to make progress in this arena, as the sets of inputs one can administer under fuzzing must be tied to the application semantics.

* Machine Learning to aid Software Testing: Given the growing variety of compilers, optimization flags, target architectures, and precision choices, it is no longer reasonable to expect traditional anomaly detection methods based on bounding roundoff error to be sufficiently general to confirm the health of a piece of code after porting. More sophisticated anomaly detection methods---say based on creating latent models using auto-encoders---seem promising, but must be developed through collaboration with domain scientists.

* Integration of verification in existing programming models: RAJA and Kokos are widely used. Integrating verification methods into such portability layers would increase the adoption of such methods. Testing can be made more generalizable if paired with RAJA/Kokkos. Formal specifications in this space promise to shift testing from per-code-instance testing to testing/verifying the transformation pipeline itself.

TIMELINESS / MATURITY: The growing use of Machine Learning to cut down simulation effort is particularly noticeable. For instance, the SARS CoV-2 Spike Protein was characterized in a Gordon-Bell Award Winning Project in 2020 using judicious use of molecular dynamics aided by Adversarial Variational Autoencoders. As these combined uses increase, software tools to verify both the "HPC side" and "ML side" will become essential. Also, as we pointed out, many of the technologies have begun to mature, now needing the impetus of focused challenge projects where domain scientists and computer scientists work hand-in-hand on a few well-chosen Grand Challenges---for example, take the software pipeline used in the SARS CoV-2 study, identify mini-apps and prox-apps that help software tools advance, and begin applying these tools to future mixed HPC/ML components.

By analyzing the aforesaid success stories and transplanting them carefully into other areas of Heterogeneous Computing in both senses---combined uses of CPUs and GPUs as well as combined uses of HPC and ML---we will move closer to achieving the goals of reliability and explainability. Without these imaginative directions, we will expend more resources, yet obtain poorer overall software quality.

[1] Guarding Numerics Amidst Rising Heterogeneity", Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, Zachary Tatlock. https://correctness-workshop.github.io/2021/

Data Lifecycle Assurance in Scientific HPC

Carlo Graziani (cgraziani@anl.gov)
Argonne National Laboratory

Challenge: Open science academic HPC centers—e.g. the NNSA ASC ASAP/PSAAP centers, or SC SciDAC centers—are large collaborations with heterogeneous skill mixes, relatively long lives, intense collaborative activities, and managed operations. They generate and acquire, annotate, transform, analyze, flow, and store very large masses of heterogeneous data in processes directed at generating new scientific knowledge. Currently this data is not managed systematically across its lifecycle in a manner that protects its scientific payloads from failures such as corruption or loss of provenance or simply disappearance. Consequently, some scientific payoffs of large investments in HPC are lost. This loss was not as consequential when Moore's law was still in force, because that scaling palliated the problem by deflating every "Hero" HPC computation from a few years ago into a routine problem for a current University research cluster. Nowadays, however, the end of Moore's scaling makes the science content latent in current computational data far more valuable, and loss due to incomplete curation of the data lifecycle that much less defensible.

The trouble is the very young age of the scientific enterprise represented by such centers—about 25 years, dating from the original ASC ASAP centers [1]. Those centers successfully defined the skill mixes and collaborative modes of modern academic HPC centers, but brought along the habits of mind of the individual domain scientists, computer scientists, and applied mathematicians who joined the centers from their academic research groups. Informal small-collaboration habits do not naturally give rise to a notion of comprehensive operations management. This is unfortunate, because that very notion is central to the successful operation of essentially all mature collaborative scientific enterprises, such as chemistry or biochemistry wet labs, or ground-based astronomical observatories, or high-energy physics (HEP) experiments, or space-science missions. In each case, a clear management role is established, charged with protecting the integrity of all scientific information generated by the enterprise at all stages of its life, and granted responsibilities, authority, budget, and staff commensurate with the duties necessary to carry out that charge. At the smallest data-rate scales—"classical" chemistry labs, say—that role may be called the "Lab Manager," and the role's associated duties include ensuring that lab notebooks of all bench scientists are kept up to the highest standards. At the other end of the current data-rate scale—the LSST Rubin Observatory—we have an "Operations Director" who is a peer of the Project Director, is in charge of creating detailed specifications for data storage, formats, transmission, and use [2], and oversees significant budget and technical staff. Similar management roles are found in other mature collaborative sciences all along the data-rate scale, where they signal a project leadership's commitment to safeguarding the science gained from the project's investments in the effort of every collaborator and in the resources put to use. *Data lifecycle assurance* (DLA)—a term introduced here to encompass full accounting of data, metadata, analyses, and provenance from birth through archiving—is a paramount concern of these scientific enterprises. Open-science HPC centers are a conspicuous exception, a fact due to the relative immaturity of the field of computational science.

Opportunity: The Department of Energy (DOE), recognizing aspects of this issue, has issued a "Policy for Digital Research Data Management" [3] which requires a Data Management Plan (DMP) as part of every research grant proposal. At the moment, the policy is not very prescriptive, and DMPs for new HPC projects, while growing in importance in peer-review weighing of proposals, do not typically offer full DLA comparable to, say, a space astrophysics mission data plan [compare the *Chandra* Observatory's detailed data layout specified in 4]. It is important to understand that the reason for this is not sloth or ignorance: the real reason is that this kind of assurance is *difficult*, *time-consuming*, and *expensive*. This is particularly true if we include, as part of the "data lifecycle," the idea of "future-proofing" derived data whose meaning in many cases resides not just in metadata but also in the analysis codes devised to transform and analyze it. Such codes must also somehow be future-proofed, which is considerably more difficult than what is expressed by the Reusability ("R") requirement of "FAIR" [5]. An example of the complexities of future-proofing analysis codes in the HEP context is discussed in [6]. Moreover, it is crucial to provide for DLA beginning at data birth, because retrofits are exponentially more difficult and expensive.

Consider a thought experiment: how would we write a DMP for a new SciDAC center proposal that *does* offer DLA comparable to that of an astronomical observatory? From the above discussion we can see that the place to begin is not from the perspective of software development or tool selection or archiving partner choice, but rather from a discussion of *management relations*. There need to be *new types of center personnel* tantamount to gathering an entire new group inside the center, alongside the domain scientists, the computer scientists, and the applied mathematicians: a DLA group. The DLA group's charge is to protect the science payload of the center's data from corruption and loss. The DLA group has budget, staff, responsibilities, and authority commensurate with this charge. It chooses tools, supplies infrastructure, writes documentation, and provides procedures that simplify collaboration while incentivizing proper DLA practice.

In principle, DOE could turn this thought experiment into reality by modifying its digital management policies to *require* some similar management arrangement in every proposal, requiring that proposal *budgets* feature lines for DLA group personnel, materials and supplies, and specifying proposal evaluation criteria that include the credibility of the co-investigator named as DLA group leader.

This is the opportunity: by embedding these kinds of requirements in funding opportunity calls for proposals, DOE could realign the incentives that have militated against implementation of comprehensive DLA practices in scientific HPC, bringing about a renewal of the scientific culture in the field. The new culture would place a high value on data curation from birth or acquisition to archiving, and would bear a strong resemblance to the analogous culture in a mature scientific field such as observational astronomy. The outcome would be a higher yield of scientific knowledge from each computation, and greater longevity for each computation's relevance.

Timeliness: In a sense, scientific HPC has now been around long enough that it is past time for it to enter a "mature" science phase, by adopting the kinds of management relations that older collaborative sciences were forced to adopt in order to succeed [7]. But it is particularly important that this should happen now. In the first instance, we have lost the comfort of Moore's law, and cannot simply wait for expensive computations to become cheap so as to recover results lost to inadequate data curation. Furthermore, if the HPC machines to be procured beyond the current crop of Exascale machines are as highly-optimized for ML/AI computational loads as is currently foreseen, those machines are unlikely to provide HPC simulation scientists (such as fluid dynamicists, or high-energy physicists) with the kind of performance advances that such generational upgrades have brought in the past. Data from the upcoming Exascale machines are therefore likely to have value well beyond the lifetime of the machines themselves, if the lifecycle of that data can be extended and secured. Those Exascale machines are nearly upon us, though. We'd better get started.

References

[1] T. Hoang, J. Aidun, N. Hoffman, R. Voigt, and R. Watson. *Advancing Simulation Science: The Legacy of ASC Academic Strategic Alliance Program*. Apr. 2011. URL: https://www.yumpu.com/en/document/read/4058562/the-legacy-of-the-asc-academic-strategic-alliance-program.

[2] R. Blum. *Vera C. Rubin Observatory Data Policy*. RDO-13. July 2021. URL: https://ls.st/RDO-013.

[3] *DOE Policy for Digital Research Data Management*. URL: https://www.energy.gov/datamanagement/doe-policy-digital-research-data-management.

[4] *The Chandra Proposers' Observatory Guide, Cycle 23*. Dec. 2020. URL: https://cxc.harvard.edu/proposer/POG/.

[5] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J. Boiten, L. B. da Silva Santos, P.E. Bourne, et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific data* 3.1 (2016), pp. 1–9. URL: https://www.nature.com/articles/sdata201618.pdf.

[6] Y. Kemp and D. Ozerov. "Preparing experiments' software for long term analysis and data preservation". In: *Journal of Physics: Conference Series*. Vol. 396. 6. IOP Publishing. 2012, p. 062011. URL: https://iopscience.iop.org/article/10.1088/1742-6596/396/6/062011/pdf.

[7] C. Graziani. *HPC and the Lab Manager*. Nov. 2021. URL: https://bssw.io/blog_posts/hpc-and-the-lab-manager.

Understanding Complex Software Dependencies for Better Scientific Reproducibility

Dan Gunter, Lavanya Ramakrishnan, Drew Paine {dkgunter, lramakrishnan, pained}@lbl.gov
Lawrence Berkeley National Lab

Challenge. In the NAS report on reproducibility in science [1], one of the key recommendations (6-3, p.118) is to invest in "..research and development of open-source, usable tools and infrastructure that support reproducibility for a broad range of studies across different domains..". To achieve this goal, we must consider how those results are used in subsequent "studies", in particular the challenge when *encapsulation of these results in computer code and data are part of a network of software dependencies*. In particular, what happens when one code (or its input data) is updated to reflect corrections or enhancements? The automated packaging and dependency management in modern software allows dependencies to propagate quickly and unpredictably. Incidents in the commercial sector highlight the potential for rapid changes, such as the removal of the *leftpad* library and its domino effect on thousands of JavaScript libraries; or malicious misspellings of popular Python libraries on the Python Package Index (PyPI).

Although scientific software typically operates in a smaller and more closed ecosystem, scientific software stacks are rife with complex legacy codes and are themselves increasingly complex and dependent on methods derived from data that in turn are built from other methods derived from data, and so on. Software may change without a clear indication of how or why, so that what may seem like a deterministic artifact (version X of software Y) actually has the hidden dynamics of a distributed system, which Leslie Lamport is famously credited with describing as, "... one in which the failure of a computer you didn't even know existed can render your own computer unusable".

This situation raises important questions that should be investigated as a community. How can we create tools and understanding that can be used to address multi-level change in software? What is being done now, and what would it require to change that behavior? How do current scientists and computing professionals choose which software to trust, and what factors cause that software to gain or lose trust in the community? Is reluctance to grant trust stifling innovation? When can we, and should we, "trust but verify"? Will the growth of complex large-data-driven analyses such as AI/ML be an important factor in making computational (and scientific) results harder to trust? Is there additional work needed to protect against malicious changes, e.g. hijacking resources or destroying public confidence in targeted domains?

Opportunity. The broad spectrum of research in the national laboratory system, from astrophysics to earth sciences, provides the perfect opportunity for cross-cutting studies of how science is performed. We can systematically examine the impacts of the scientific domain and many other factors, such as the scale of the computing, and computational resources, the complexity of instruments used to perform experiments and gather data, and the history of research groups, in tandem. To understand software complexity and dependencies, we believe that four key areas that should be included are software releases, verification and testing, documentation, and versioning.

Software releases. The way software changes and dependencies impact users depends on how software is released and what assumptions are made about how the users will access it. Source code package managers, facility operator installation, virtual machines or Docker containers, software as a service, and commercial software are all used for scientific results. How do these different approaches interact with other software packages? Who is using which approaches, why, and what are the pros and cons? Is there any set of best practices that can be derived that would match practices to use-cases?

Testing. Software testing is crucial for understanding when a change in the underlying software stack (your own code, or others') affects the results. Adoption of "best practices" in software testing is still uneven in our cross-functional teams, and based on our own experience [2] this is a social problem that requires an array of approaches. Technical improvements are also needed to meet the unique demands of scientific computing (e.g., special hardware; long, complex runs; high sensitivity to inputs). Our community has a unique depth in statistical approaches, including AI/ML, that could be leveraged to perform large-scale analyses across software networks to detect potential conflicts or bugs.

Documentation. The semantics for correctly using scientific software conveyed by documentation are potentially crucial, yet under-examined. Anecdotally, failures of the documentation are a prime source of overt and subtle errors — but is this objectively true? How are resources in our environment allocated to documentation, and how is its effectiveness evaluated and communicated; how can user research and design help?

Versioning. Semantic versioning provides minimal information on change in software. In our experience, personal communication is needed to understand many details of exactly what changed and why. Yet this approach is not scalable. This raises questions of how software versioning needs to be improved in the scientific software context.

Timeliness. The scientific software environment continues to grow in complexity and increasingly common use of complicated AI/ML analyses makes confidence in the underlying code even more critical. We should begin now to move the cultural needle from a largely folkloric understanding of software dependencies to a more evidence based and secure model. *The expected impact of this work is a more direct and efficient path for DOE software teams to create quality software in which both they and the scientific community can have confidence.*

References

1. National Academies of Sciences, Engineering, and Medicine. 2019. *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press. doi: 10.17226/25303.
2. K. Beattie and D. Gunter, "Lessons learned working with protected assets in an open-source collaborative scientific software project," 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS), 2021, pp. 13-14, doi: 10.1109/BoKSS52540.2021.00014.

Automated Validation and Verification for Scientific Software

Yanfei Guo, Ken Raffenetti, Hui Zhou, Travis Koehring,
Sudheer Chunduri, Xiaodong Yu, and Rajeev Thakur

Argonne National Laboratory

November 2021

Software validation and verification is an integral part for many successful software projects. It has been adopted in scientific software projects and is a major focus of the Exascale Computing Project (ECP). However, the current software testing practices in test coverage and usage of automated tools in many scientific software projects are not sufficient. Moreover, due to the scientific software's unique emphasis on performance and optimizations for supercomputers, the common software verification and validation techniques do not fully meet the needs of scientific software. We believe more research is needed in automated testing of scientific software (for correctness, performance and scalability), and many scientific software packages will benefit from greater use of even existing tools and approaches that automate testing.

1 Challenge

Complexity of Scientific Software. Scientific software usually contains multiple algorithms and optimizations. They should all be tested in the development cycle, but the current software testing practice makes it difficult to achieve good test coverage. In some cases, it is hard to quantify the test coverage. For example, MPICH contains more than 1,000 individual tests. But these tests are a combination of hand-crafted tests and mini-apps that exercise ad-hoc use cases and those collected from users that represent a specific failure mode. Not only does the test suite not contain all the valid use cases, it is often infeasible to measure the test coverage. More importantly, software projects that have good test coverage can fall short in the future as more and more new features are added. This is especially challenging for scientific software that faces highly heterogeneous hardware architectures of future machines. Manually creating and maintaining software tests would be impractical.

Lack of Regular Performance Evaluation. Scientific software has a unique emphasis on performance. But a regular performance evaluation of the software is often not adopted in the development cycle. Most performance evaluations in research projects are done when a new algorithm or optimization is introduced. Our study has shown than even a small change in rank-address translation in MPI can have significant performance impact on message rate [1]. The inability to identify performance regressions in daily development could require more investigative and development effort in finding performance bottlenecks. It even runs the risk of missed opportunity for optimization, where overall performance improvement masks local inefficiencies.

Gaps Between Synthesized Tests and Real World Applications. There are two gaps between the synthesized tests and real world applications. First, it might not be feasible for the developer of a scientific application to foresee how the software would be used and create tests for it. Second, the software developer may not have the resources or access to systems that are large enough to recreate the real world application environment. Therefore, it can be impossible to test specific failure modes. Creating mini-apps that are modeled from the target applications would be helpful in bridging the gap, but it requires knowledge about the target application. It is also a burden to the application developer to maintain mini-apps.

High Maintenance Cost of a Test Suite. One final challenge related to testing is the cost and maintenance of the tests. For many large open-source projects, the ratio between test and feature code is between 1:1 to 2:1. This represents a large burden for the scientific software team that a great portion of their effort must be dedicated to developing the test suite rather than adding new features or optimizing performance.

2 Opportunity

The core solution to address these challenges is automating software testing:

Adopting Analytic Tools. There are many commercial or community-based analytic tools that can broaden the test coverage of a scientific project. Examples including static analysis tools that identify common coding problems and LLVM-based sanitizers that check for undefined behavior and detect memory errors. These tools provide portable ways to test against common software problems, which improves the quality of software testing without requiring significant development effort. Note that one limitation of these tools can be the lack of knowledge of the semantics in the software being tested. Therefore, they could report false positive results.

Guided Automated Test Generation. Most large software projects employ automated test generation. For example, unit testing libraries allow the developer to define only the target code component and correctness conditions for testing without manually creating the entire test code. Taking this approach one step further, the developer can define expected function behaviors and let the testing tool automatically generate different use cases. One example would be automatically generating tests for MPI point-to-point communication. The testing tool can create combinations of different send and receive functions based on the MPI Standard specification. Other tools exist that can generate mini-apps captured from real-world applications based on execution traces. Guided automated test generation allows us to create a more customized test suite for the target application that evaluates multiple aspects including correctness, robustness, and performance.

Randomized Testing. Randomizing API parameters is an effective way to test a large range of API use cases. Fuzz testing is one such implementation of this technique. Fuzzing tests randomized (sometimes invalid) input in an effort to expose several classes of bugs, often with a focus on data integrity and security. This technique can be combined with other automated test generation approaches to create more sophisticated tests without significantly increasing the cost of testing. Many projects recognize the utility for automatically generated randomized testing today. One notable implementation is libFuzzer [2], which is part of the LLVM compiler toolchain. Google has developed syzkaller [3], which is a tool originally designed for fuzz testing the Linux kernel but has expanded to support testing several other operating systems. Syzkaller sends automated bug reports to developers when problems are discovered, leading to fixes and overall higher quality software.

3 Timeliness and Maturity

We have already started adopting these approaches and have some initial success. MPICH uses Coverity for static analysis and LLVM tools to check for undefined behavior and safe memory accesses. We also explored using new tools, such as Coderrect [4], for identifying deadlock in multithreaded codes. We have developed an automated test generator for testing MPI datatypes. It creates MPI derived datatypes with random layout characteristics. In fact, we were able to find bugs that were undetected for years despite regular testing with manually written test codes. This test generation capability was also useful when we expanded the test suite to support GPUs. We simply allowed it to create buffers on both CPU and GPU, and our tests are now able to test any combination of buffer locations.

We have also adopted automated code generation for MPI language bindings for C and Fortran [5]. The binding generator simply parses the MPI interface specification from the standard documentation and generates the MPI functions with appropriate checks on input/output parameters and error handling. The developers only need to implement the checking and error handling functionalities once, and the binding generator will apply them to all MPI C and Fortran language interfaces. This reduced the code that needed to be maintained from more than 40,000 lines down to only about 3,000 lines. It also lowered the possibility of human error in inconsistent implementation for the same functionalities.

These experiences encourage us to further explore the possibility for automated test generation that can exercise different code paths in MPICH. We have experimented with static analysis tools and the LLVM compilers to identify branches for these code paths and their corresponding conditions. We use these tools to understand how changes in the input parameters affect the internal execution, which in turn automatically identifies boundary conditions that need to be tested.

References

[1] Yanfei Guo et al. "Memory Compression Techniques for Network Address Management in MPI". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Orlando, Florida, 2017.

[2] *LibFuzzer – a library for coverage-guided Fuzz Testing.* URL: https://llvm.org/docs/LibFuzzer.html.

[3] *Syzkaller: Syzkaller is an unsupervised coverage-guided kernel fuzzer.* URL: https://github.com/google/syzkaller.

[4] *Coderrect.* URL: https://coderrect.com/.

[5] Hui Zhou et al. "Generating Bindings in MPICH". In: *Proceedings of the 28th European MPI Users' Group Meeting.* EuroMPI '21. ACM, 2021.

The role of culture in scientific software teams

Authors: Rinku Gupta (rgupta@anl.gov), Argonne National Laboratory

Introduction

Scientific computing research environments, where teams often work on open science projects, have collaborations spanning local and remote individuals, as well as teams working in multiple countries and cultures. Individuals on these teams have diverse cultural backgrounds, experiences and working styles. It is common to have composite teams with members who have lived, grown up, studied or worked in various parts of the world. In addition, such research teams have a great mix of students, post-docs, software engineers, and scientists -- individuals who are at distinct stages of their careers. While such culturally rich and diverse environments may exist in industry, the diversity is indeed (arguably) more pronounced in scientific research environments. Acknowledging and appreciating this cultural diversity is essential to build understanding of how to make scientific software teams successful.

Research projects usually start with a nascent idea with rough project requirements, ambiguous next steps, and hazy but hopeful end goals. When working on such projects, it's important to build confidence, trust and respect among team members. Every successful project has a successful team behind it. It's important for team members to understand their own conscious and subconscious working and interaction styles, as well as to recognize, understand and overcome any inherent subconscious cultural biases and stereotyping tendencies. A successful team is the one where individuals can work together, feeling integrated, valued, appreciated and united, irrespective of the differences in their cultural backgrounds. In the last few years, there has been increased momentum in diversity, equity and inclusion (DEI) initiatives in the scientific computing community. This paper supports DEI initiatives and stresses the need to gain an understanding of cultural backgrounds as an important step towards improving team productivity. The rest of this document focuses on the importance of education and training, as well as interactions between team members and leadership, as first steps towards productive multi-cultural teams.

Education and Training

Team members in scientific computing organizations are often provided with organizational-level training (also called human resources training). Individual teams also provide on-boarding training, focusing on getting newcomers integrated into the team, process and technicalities. Many organizations also provide training on high-level DEI initiatives. However, a major training component missing relates to cross-cultural training. Members of a team rarely have insight on the overall cultural background of foreign minorities in the group. This lack of understanding can be the root cause of conflicts in a group, stemming from differences in communication styles and behavioural expectations. This indicates a need for training focusing on cross-culture education, intercultural conflict-management and perspective-focused discussions. Also useful would be to organize peer seminars focusing on sharing experiences to show benefits of cultural changes in a team, how they have been achieved and the pros/cons of the methods followed. The goal should be to increase the cultural intelligence quotient [1] of team members, where cultural intelligence refers to the skill to relate and work effectively in culturally diverse situations. Cultural intelligence is usually measured on a scale, similar to that used to measure

an individual's intelligence quotient. While the term cultural intelligence is common in industry, its exact definition and application in the scientific computing community needs more research.

Team Leadership

In the scientific community, leading a culturally-sensitive team and project, whether local or remote, could be a formidable task, depending on the cultural intelligence and sensitivity levels of the team members. Team leadership needs to be cognizant that cultural gaps exist, and that attention, budget and effort are needed to educate members with an aim of improving individual and team happiness and productivity. Several parts of the world have strict hierarchy at the workplace that brings forth a different communication style between leadership and subordinates. It is up to the leader to gauge and understand the vibes of the team and listen to verbal and non-verbal diverse opinions. Sometimes employees face language barriers while speaking but not writing. Understanding such strengths and weaknesses of employees and using that to improve individual as well as team productivity is also a prime charter of the leader. Some efficient ways to help shape culturally diverse teams include: inculcating a culture for sharing credit for work, paying attention to hidden effort done by the more introverted members of the team, partnering employees with appropriate mentors, and rewarding/acknowledging efforts publicly. Designating work while keeping in mind major holidays of specific cultures and acknowledging the significance of those events creates a relationship of mutual trust and respect between team members. A common culture in industry is that of "team building", where the employer budgets a small amount for employees to go off-site and have fun, a few times every year. This usually results in building camaraderie and friendships that go beyond work. Such budgeted "team building" should be encouraged. Another way to build cultural integrity in the team is to organize activities during important cultural festivals and occasions and encourage employees to comfortably participate. Several of the above suggestions come from industry and could be made applicable to scientific computing teams, provided that support is given to team leadership at an organization level.

Conclusion

This document explores the significance of education/training and team leadership, as important steps towards creating culturally intelligent teams. However, these are just the starting steps. Several ideas on the topic of culturally intelligent teams have been introduced in industry. While these ideas can be appreciated, the scientific computing community needs to do its own research to explore if and how these ideas can be adopted effectively in scientific environments. As outlined in [2], multicultural teams have to deal with several dimensions related to conflict management, expectation management, team cohesion, performance improvement, task orientation, trust establishment, understanding the value in diversity etc; the scientific computing community needs to understand these dimensions better. This document focused on individual and leadership-level changes, but organization-level awareness and education also are critical for changes to have significant impact in the community. Organizations need to be willing to invest the time and resources to empower teams to work on building culturally intelligent teams.

References

[1] Cultural Intelligence: Individual Interactions Across Cultures, P. Christopher Earley and Soon Ang, Stanford University Press, 2003,

[2] Managing Diversity: 10 Steps to Multicultural Team Success, Claude Koehl, Oct 2016, The Training Magazine, [Last viewed: Nov 2021]

Quantitative methods for research software community management

Kyle Harrington[1,2]

[1]Image Data Analysis Group, Max Delbrueck Center for Molecular Medicine, Berlin, Germany
[2](Joining Dec 6, 2021) Multimodal Data Analytics Group, Oak Ridge National Laboratory

Challenge

There are multiple problems that commonly arise in research software communities that inhibit their growth and productivity: ad hoc governance, redundant efforts, and hostile behaviors. These problems are social and can be addressed with community management and tools that support community management.

Community management is an important aspect of successful research software projects, but there are no well defined standard of practice and community management support tools are lacking. Some large projects (especially in industry) have realized the benefits of having dedicated community managers. However, in most cases community management becomes the responsibility of research software engineers (RSEs), RSEs are often unaware of the impact of their behaviors and interactions on community growth and maintenance (although there are exceptions). In positive cases, RSEs encourage users to take ownership of tasks like bug reporting, and may assist users with contributing their first software fixes. In detrimental cases, RSEs may be dismissive of users and other developers for being less knowledgable, and sometimes may simply ignore community engagement due to lack of resources to invest in community level activities. These behaviors can: grow or shrink the community, facilitate conversion of users into contributing developers, contribute to developer attrition, reduce community diversity, and lead to duplicate efforts. While these projects can attract millions of dollars worth of investment, the failure to fully appreciate their respective research software communities leads to significant unrealized potential.

This paper uses the open source scientific image analysis community as an example of a healthy, growing research software community. In this community, a few large-scale projects that have hundreds of contributors coexist (e.g., Fiji/ImageJ, scikit-image/napari, ITK/VTK). The funding strategies followed by these three example projects represent notably different mechanisms, where Fiji and ImageJ was originally supported by a single NIH researcher followed by numerous grants at institutions across the world, scikit-image and napari are now largely supported by the non-profit foundation: the Chan Zuckerberg Initiative, and ITK and VTK are supported by an open-source company: Kitware. Funding strategies have an impact on the capacity to directly support community management due to financial flexibility, institutional continuity, and the potential necessity to focus on developing new features and projects. This example community can be used to establish a broader model of behavioral dynamics in research software communities that can be used to evaluate management policies, the impact of individual behaviors, and determine useful quantitative measures for assessing community health, success, and productivity.

Opportunity

This paper proposes to use computational models and social coding metrics to guide and assist in the identification of individual strategies and community policies that can reduce existing problems and make research software communities more robust and productive. Consider that each community is a population of individuals (including users and developers) with skills (application of tools, programming, ...), needs (analyzing data, getting credit for tool creation, ...), and behaviors (collaborative, lone wolf, codependent, ...), and that this community is governed by a code of conduct. By simulating population models of research software communities with different parameters, it is possible to see how interactions between individuals and community governance impact the dynamics of the community. Does the user base grow? Is there developer attrition? Is the community robust to the loss of key individuals? This approach can ultimately be used to identify the impact of individual behaviors and help shape community policies to maintain positive engagement and grow the number of users and contributing developers.

Establishing this model of research software communities involves a range of fields, including software engineering, agent-based modeling, game theory, cognitive science, and even project management. We will draw upon all of these disciplines and utilize the latest technologies in collaborative open source software engineering to enable the initialization of simulations based upon measures taken from real-world communities. This will make it possible to answer many important questions:

- What type of policies can we adopt to improve engagement in research software communities?
- What is the role of community member interactions in facilitating the conversion of users into contributors?
- Can we detect and quantitatively measure beneficial and pathological behaviors of developers based upon their social interactions (e.g. Github issues, mailing lists, code review, etc.) and other activities?
- What factors are relevant to defining the success of a software community: growth of user base, increase in number of developers, greater number of code contributions, etc.?
- Can we eventually: predict the success of a software community, incentivize behaviors that lead to positive community growth, and other future forward strategies?
- Can we quantify major differences between research software communities across domains?
- When creating a new research software project, how heavily should one weigh each choice (e.g. programming language, social coding platform, communication mechanisms, amount of community engagement, etc.)?
- Can we identify macroscale commonalities between communities? Can we find shared goals or subprojects that could be achieved by two or more communities cooperating?

Timeliness

There have been many recent advances that contribute to the possibility of this research direction, including: (a) advances in modeling and simulation have laid a foundation for the establishment of models of research software communities, (b) new social analytics tools can quantify existing research software communities within modeling frameworks, and (c) a large number of open source research software projects have significant presences on social coding platforms. While general progress in these areas has established a groundwork for the proposed research, there are some key technological advances and social developments that are particularly important.

Social coding platforms and continuous integration tools now dominate the software engineering industry. Sites like Github and Gitlab are built upon principles of social coding, where source code hosting, issue management, community discussions, and more, all reside within a single platform. These platforms have open APIs that make it possible to retrieve data and statistics related to projects and users. These data can serve as a quantitative basis for establishing models of research software communities based upon the latest statistics. Continuous integration tools and pipelines are triggered based upon events within projects, like code updates, which can be used to trigger subsequent analyses of community interactions, such as community management recommendations.

Natural language processing (NLP) has significantly advanced with tools like GPT-3 that can generate realistic natural language. The NLP subfield of sentiment analysis serves as a way to algorithmically assess valence and tone of user interactions, such as detecting bullying behavior. This opens the possibility for supporting community managers at monitoring large volumes of community interactions. Additionally, tools like Codex can now explain code, providing an easy way to facilitate the transition from non-coding users to code contributors.

Some tech communities and individuals have developed notorious reputations when it comes to diversity, equity, and inclusion (DEI). Those who are sick fall behind; people who take parental leave lose opportunities; women are harassed. While DEI activities and agendas are finally being actively pursued, there are still many open questions in the scope of community management. If there is only private and/or in-person information about negative interactions between individuals, then how can these interactions be resolved within the community? How can disadvantaged individuals be supported when they have long periods of inactivity or require accommodations?

The maintenance and growth of sustainable research software communities should be directly addressed through the science of scientific software development. The proposed research will enable the assessment of strategies and policies, and development of management support tools to facilitate the advancement of research software development. These studies and tools can also inform and assist new projects when they are being established by suggesting a set of best practices. Ultimately, this work will empower research software communities to eliminate waste from redundant efforts and loss of skilled developers, and ensure that these communities can focus on the development and application of scientific software.

Adventures in Modularity - Why is Sharing Hard?

2021 December ASCR Workshop on the Science of Scientific-Software Development and Use

Cyrus Harrison (cyrush@llnl.gov), Arlie Capps (capps2@llnl.gov),

Richard Hornung (hornung1@llnl.gov), and Mark Miller (miller86@llnl.gov)

LLNL-ABS-829361

Summary

This white paper focuses on: Roadblocks faced by computational scientists in computational application use.

Sharing data across a modular scientific software ecosystem is a key challenge in computational science. When solving science and engineering problems, users trying to integrate tools face data semantics and scalability challenges. First and foremost, data semantics barriers impede adoption and seamless use of tools. Examining our efforts in modular software development and deployment in the Weapon Simulation and Computing (WSC) Program at LLNL, we conclude that the ability to adapt data shared between software components is much more important than the ability to agree on data representation. We see investment in strategies and tools that simplify data description and adaption as an important way to improve the development and use of scientific software.

Introduction

Investments in modular software are rapidly expanding our scientific computing ecosystem. Computational scientists now have access to an extensive set of freely available and capable software components. It is challenging to learn how to use such a broad set of tools. A key ingredient to successful modularity is the ability to share data between scientific applications, libraries, and tools. A common and often immediate roadblock to getting started is the disparity of data requirements between components. For example, a multi-physics application may use different computational meshes in different physics models. To couple these models, the simulation must faithfully exchange mesh data between different representations. Data sharing extends beyond physics models, to in situ and post-hoc capabilities including load balancing, visualization, and data analysis. Although a tool may provide a perfectly targeted capability, the cost of adapting data to use the tool may seem insurmountable, which may be a deal breaker. Sharing data is not a new challenge. But using more tools with varying data representations together elevates the challenge by making it highly visible in user workflows.

Challenges to Sharing

Sharing data is challenging due to the diversity of problem domains, software constraints, and especially human factors. In our ecosystem, two categories of sharing challenges stand out:

Data Semantics. This challenge occurs when human understanding of data and how it is represented are barriers to sharing. Creating software is still very much a human endeavor. At the core of successful data description semantics is the selection of constraints and abstractions that allow humans to implement capabilities that meet their needs. Human minds and software systems must agree.

Data Scalability. This challenge occurs when computational cost or data volume are barriers to sharing. Data movement is very expensive at the largest scales of computing. Even within a HPC system node, the cost of data adaption can present complex cost-benefit tradeoffs. Data representation choices can make, or break, algorithm scalability and different processing paradigms can require wildly varying data representations.

Should we Agree or Adapt?

For this white paper, we focus on data semantics challenges because they are usually a barrier computational scientists face well before data scalability. The reason for this is that scientists often test the waters at smaller data scales to understand the usability and applicability of a capability before investing more time. Also, every problem cannot afford or does not merit solutions that require the largest HPC systems.

How has our broader community addressed data semantics challenges? To constrain this discussion, we focus on examples of mesh-based data often used when solving a set of coupled PDEs. Representing mesh-based data is a common need for simulation applications and mesh data is commonly shared between meshing tools, simulation codes, visualization, and analysis tools, etc. Can standardization be a useful tool here? Can we build one-true data schema or API to unite all mesh use cases? Experience says perhaps, but not entirely. The DOE has invested in the pursuit of one-true schema or API at varying levels of scope. Some notable examples

include the ASCI Data Models and Formats (DMF) [1,2,3,4,5] project and the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) [6] project. These investments absolutely helped evolve our ecosystem. DMF helped establish HDF5 as a key substrate and shaped shared mesh-based data models that are in use. ITAPS showed it was possible to provide a common interface to unstructured meshes with multiple back-end implementations supporting the interface.

Even with these benefits, the one-true solution approach (for file-formats, APIs) has not come to full fruition. Too many stakeholders with a diverse set of needs make universal adoption and agreement intractable. Instead, there are many frameworks and libraries that provide rich mesh-based APIs. These APIs make data modeling choices that support their methods and algorithms. For example, visualization APIs need to support a wide range of mesh types, where a toolkit focused on solvers can be more efficient with a different set of abstractions. Although a broad set of mesh-based APIs is available, many applications and simulation domains still develop and use custom data models. Our state-of-the-ecosystem is a fractal mix of solutions, ranging from some with large communities to one-off bespoke solutions employed by a single developer.

Despite the diversity, we still share mesh data. But how? The VisIt [8] visualization tool provides an example. Many algorithms in VisIt leverage VTK, a key substrate that enabled rapid adoption of scientific visualization capabilities in the DOE (and beyond) [7]. VTK's "toolkit" style design allowed ParaView [9] and VisIt teams to share infrastructure while providing flexibility to enable multiple solutions. This is an example where visualization tool developers successfully share infrastructure across projects. But how do VisIt users get their data into to these algorithms? The answer is that VisIt provides 140+ data file format readers to ingest a wide range of data formats. Investing in support for a wide range of formats and data adaptors was key to get capabilities in users' hands. This was a large investment! We also note that ParaView leverages this investment by using VisIt's database readers. Another example is HDF5 [10], which is widely used in part because it provides the flexibility to different teams to build their own data schemas on top of HDF5's general I/O APIs. In fact, several VisIt readers exist to read different flavors of mesh data stored using HDF5.

The key takeaway: *The ability to adapt data is much more important than the ability to agree on data representation.* Data adaption may be as simple as adding metadata to existing bytes, or as complicated as transforming mesh data to a new mesh representation.

Opportunities to Simplify Adapting Data

We see two areas that need continued investment to simplify the process and reduce the effort required to adapt data:

Simplifying In-Memory Data Description

Most of our existing success with sharing data involves file based I/O interfaces. In-memory sharing poses different software engineering constraints and integration considerations. It remains a gap for seamless tool use. We can close the gap by investing in:

- Multi-language constructs for describing hierarchical data
- Support for data introspection and validation to allow rapid consumption of described data
- Support for the ability to apply metadata to existing data structures

Create Bridges and Share Common Data Transforms

With more software modularity, we continue to see more components and more cases where data adaption is important. Shared data interfaces may be a bridge too far while data transform capabilities may be more tractable. As more capabilities are connected, users will have to adapt data to leverage the work of others who have gone through the process previously. We can help users by investing in:

- Bridges from in-memory interfaces to a wide set of file I/O capabilities
- Bridges from existing file I/O paradigms to in-memory interfaces
- Bridges from in-memory interfaces to other community data models to other tools (ML/AI)
- Tools to map data between representations (Mesh Overlay, Flattening, etc)

References

1. Cook, L M, Matarazzo, C M, and Rathkopf, J. (1998). "Tri-Lab data models and format (DMF) project: parallel I/O and data exchange". https://www.osti.gov/servlets/purl/3723.
2. Miller, M., Reus, J., & Matzke, R., Arrighi, W., Schoof, L.A., & Hitt, R., Espen, P. (2001). "ASCI DMF (The Sets and Fields (SAF) Data Model) Enabling Interoperation of High Performance", Scientific Computing Applications: Modeling Scientific Data with the Sets & Fields (SAF) Modeling System.. 158-170. 10.1007/3-540-45718-6_18.
3. Butler, D., Pendley, M.H., (1988). A Visualization Model Based on the Mathematics of Fiber Bundles. Computers in Physics. 3. 10.1063/1.168345.
4. Butler, D., Bryson, S. (1993). Vector-Bundle Classes form Powerful Tool for Scientific Visualization. Computers in Physics. 6. 10.1063/1.4823118.
5. Butler, D. (2012). The Sheaf Data Model: A Rigorous Data Model for Scientific Computing. 1303-1305. 10.1109/SC.Companion.2012.159.
6. Diachin, L., Fix, B., Kraftcheck, J., Ollivier-Gooch, C., Tautges, T., & Trease, H. (2007). Interoperable mesh and geometry tools for advanced petascale simulations. Journal of Physics: Conference Series, 78(1) doi:http://dx.doi.org/10.1088/1742-6596/78/1/012015
7. Larzelere, A.R. II, (2009) “The History of the Accelerated Strategic Computing Initiative (ASCI)”. https://asc.llnl.gov/file-download/download/public/2326
8. Childs, H., et al. (2011). “VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data.”
9. Ahrens, J., Geveci, B., Law, C., (2005) “ParaView: An End-User Tool for Large Data Visualization”, Visualization Handbook, Elsevier, ISBN-13: 978-0123875822
10. Koziol Q. (2011) “HDF5.” In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_44

Provenance based software dashboards

Lynn von Kurnatowski
Institute for Software Technology
German Aerospace Center (DLR)
Weßling, Germany
https://orcid.org/0000-0001-5144-702X

Martin Stoffers
Institute for Software Technology
German Aerospace Center (DLR)
Cologne, Germany
https://orcid.org/0000-0003-2987-4345

Carina Haupt
Institute for Software Technology
German Aerospace Center (DLR)
Berlin, Germany
https://orcid.org/0000-0001-6447-1379

I. Challenge

Over the past years, software conquered a growing number of application areas. Particularly in research, where software is already an important innovation factor and integral part. This statement is substantiated by the results of a survey that we conducted DLR[1]-wide in 2018 [1]. It shows that many researchers develop software in their daily work. It even indicates that researchers, who are already involved in software development, are interested in investing even more time in software related activities.

The increasing role of software in research, in combination with the increase of complexity in scientific fields, has led to the necessity of complex software solutions. Thus intensifying the need to ensure the quality, reliability, and trustworthiness of software systems in research. Especially in safety-critical areas, where small software errors can lead to serious failures. In order to be able to make an assurance about quality, reliability, and trustworthiness of a software product, it is necessary to have a comprehensive and detailed understanding of a software project. The large amount of heterogeneous data, which is generated before, during, and after the development in so-called software repositories can serve as data sources [2]. These repositories already contain information about the source code and the software development process. One approach to explore this data is to analyse their *provenance*.

As stated by Moreau in 2010 in "The Foundations for Provenance on the Web":

> "Provenance [...] is becoming an important concern for several research communities in computer science, since it offers the means to verify data products, to infer their quality, to analyze the processes that led to them, and to decide whether they can be trusted." [3, Page 3]

Today, *provenance* [4] is used to verify data products and to analyze processes that led to them. Our work aims to standardize, generate, and use *provenance of software artifacts* and *provenance of software development processes* by defining a general provenance data model for software development processes. Our goal is to be able to trace and determine the origin of artifacts such as issues, source code files, build results, or documentation, and to understand and get insights into the process as a whole using analysis and dashboards.

[1]German Aerospace Center (DLR)

II. Opportunity

Software development is a creative and complex process based on human decisions. Consequently, developing software as error-free as possible is a challenging task. To assess the reliability and trustworthiness of software systems, the provenance of the development process and the produced software artifacts can be recorded and analyzed.

The aforementioned increase of complexity in software solutions results in the need of more extensive development processes. Thus, resulting in very large and complex provenance graphs, which are difficult to manually understand, interpret, and verify.

Visual analysis is an opportunity to easily and quickly verify a software project, since the human brain is an amazing pattern matching machine. Thus enabling it to process large amount of visual information at once. Our idea is to create an interactive dashboard solution based on a data set of provenance describing software development processes. This allows developers to easily and quickly get an overview about a software status regarding it's quality, reliability, and trustworthiness. For this, different metrics are visualised using appropriate visualisation methods, like graph visualizations, metrics visualizations (e.g., bar charts), time-oriented visualizations (e.g., Sankey charts), task-oriented and work process-oriented visualizations (e.g., Gantt charts), or hierarchy-oriented visualizations (e.g., treemap charts).

As first step, we developed a dashboard prototype that contains individual visualizations with interactive controls (Figure 1) [5]. For our prototype we used Python and the libraries Plotly, and Dash[2]. Before visualizing the data we query it with the libraryPy2neo from a Neo4j graph database instance[3] and use Pandas to prepare the data and temporary store query results.

Since not all popular software metrics and information are of equal importance to the user, we are conducting a focus group interview to identify what requirements users have for such a visualization. As a next step a first version of a dashboard will be developed which takes the user's requirements

[2]https://dash.plotly.com/
[3]https://neo4j.com/product/neo4j-graph-database/

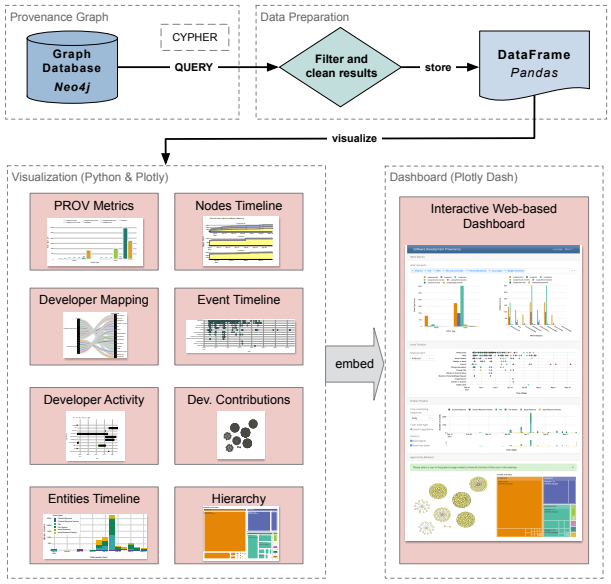


Fig. 1. Provenance visualization flow: querying the Neo4j database with CYPHER queries, storing query results and filtering and data cleansing with Pandas, visualizing with Python and PLOTLY, and displaying in the web-based dashboard with PLOTLY DASH [5].

into consideration. Finally, the resulting dashboard will be evaluated through real-world tests, to ensure that it provides an intuitive way to understand software projects.

III. TIMELINESS OR MATURITY

Due to the complexity and diversity of today's software systems, numerous development tools are used to support the the developer in his daily work. A typical tool suite consists of at least an integrated development environment (IDE), a version control system, an issue tracker, a continuous integration and continuous deployment infrastructure, and a documentation system. During the software development process, all these tools produce a large amount of different data. We use this heterogeneous data to extract the complete provenance of all process steps using W3C PROV Data Model.

The W3C specification PROV [6], which among other definitions defines the Provenance Data Model (PROV-DM) [7], allows to collect and record provenance in a well defined and machine readable formats.

Using the W3C PROV data model, we designed a general provenance model to extract *retrospective provenance* [8] about software development processes [9]. This includes provenance information of the underlying version control system as well as provenance of the respective issue trackers and release systems. The resulting provenance data then contains all activities (e.g., commits, issues changes, releases), the generated or changed entities (e.g., source code files or issues), and involved agents (e.g., developers, testers, or users) along with their relations.

Using our tool GitLab2Prov [10], [11] this information can already be collected from the web-based DevOps life-cycle tool GitLab[4]. The tool uses the GitLab API to generate the provenance information in the form of the text representation PROV-JSON, which is converted to other formats for further analysis and can be imported as a labeled property graph into the graph database Neo4j[5] using the tool prov2neo. These tools were already successfully applied to collect provenance information of inner-source and open source project [10], [12].

AVAILABILITY

GitLab2Prov is available as Open Source software under the MIT license: https://github.com/DLR-SC/gitlab2prov.

The tool prov2neo is available under the MIT license at https://github.com/DLR-SC/prov2neo.

REFERENCES

[1] L. von Kurnatowski, T. Schlauch, and C. Haupt, "Software development at the german aerospace center: Role and status in practice," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 157–160. [Online]. Available: https://doi.org/10.1145/3387940.3392244

[2] L. von Kurnatowski, M. Stoffers, M. Weigel, M. Meinel, Y. Wasser, K. Rack, and H. Fiedler, "Scientific software engineering: Mining repositories to gain insights into bacardi," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–10.

[3] L. Moreau, "The Foundations for Provenance on the Web," *Foundations and Trends® in Web Science*, vol. 2, no. 2-3, pp. 99–241, 2010. [Online]. Available: http://www.nowpublishers.com/article/Details/WEB-010

[4] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga, "The provenance of electronic data," *Communications of the ACM*, vol. 51, no. 4, pp. 52–58, 2008.

[5] A. Schreiber, L. Kurnatowski, A. Meinecke, and C. Boer, "An interactive dashboard for visualizing the provenance of software development processes," in *2021 Working Conference on Software Visualization (VISSOFT)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2021, pp. 100–104. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/VISSOFT52517.2021.00019

[6] L. Moreau and P. T. Groth, *Provenance: An Introduction to PROV*, ser. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2013.

[7] L. Moreau and P. M. et al., "Prov-dm: The prov data model," Online, 2013, retrieved: 03.10.2021. [Online]. Available: https://www.w3.org/TR/prov-dm/

[8] T. McPhillips, S. Bowers, K. Belhajjame, and B. Ludäscher, "Retrospective provenance without a runtime provenance recorder," in *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, ser. TaPP'15. USA: USENIX Association, 2015.

[9] A. Schreiber and C. de Boer, "Modelling knowledge about software processes using provenance graphs and its application to git-based version control systems," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 358–359. [Online]. Available: https://doi.org/10.1145/3387940.3392220

[10] A. Schreiber, C. de Boer, and L. von Kurnatowski, "Gitlab2prov—provenance of software projects hosted on gitlab," in *13th International Workshop on Theory and Practice of Provenance (TaPP 2021)*. USENIX Association, Jul. 2021. [Online]. Available: https://www.usenix.org/conference/tapp2021/presentation/schreiber

[11] C. de Boer and A. Schreiber, "Dlr-sc/gitlab2prov: Gitlab2prov 0.5," Jun. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5009043

[12] A. Schreiber, L. von Kurnatowski, and C. de Boer, "Analyzing software engineering processes with provenance-based knowledge graphs," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–11.

[4]https://about.gitlab.com/features/

[5]https://neo4j.com/product/neo4j-graph-database/

Scientific Software as Emergent Behavior

Gerd Heber, Elena Pourmal, Mike Folk, The HDF Group[1]

The development and use of software in science occur in an environment of rapid change and eminently finite resources. Researchers (Woods2018) have introduced the concept of *graceful extensibility* to help explain the emergent behavior of sustained adaptability in such environments. The concept blends two familiar terms - graceful degradation and software extensibility and might be the (hitherto unstated) maxim of our development of the HDF5 software.

In the case of HDF5, the practice of graceful extensibility is imperfect and sometimes outright messy. In addition to rapid change and finite resources, there is an ecosystem reaching millions of users to attend to. Perhaps the two greatest challenges currently facing HDF5 are to deal with immature software contributions and the limits of designs.

1. HDF5 receives external software contributions whose quality and usability are all over the map. Besides an immediate integration problem, there are long-term consequences. The contribution might depend on a very specific setup that is not available to us on a regular or permanent basis and can't be tested. The contribution might be so narrow in focus that it's either quickly disowned by its creators or it's just a matter of time for scope changes to emerge, and we are assumed to be the responsible party. Finally, many contributions are the proverbial band-aid, and cover up deeper issues.
2. Despite an emphasis on forward-looking designs, over time we had and will have instances where we have to face and correct design limits. Any designer's expectations for changes in conditions, contexts, uses, risks, goals, and relationships are imperfect and eventually outdated. We have successfully dealt with a few of these in HDF5, but we are far from being able to answer basic questions. For example, what are the urgency and scope with which we should address known limitations? How can we better equip ourselves to discover them sooner? What, if any, mitigation strategies are there to temporarily or permanently outmaneuver some of these limits? Can we achieve sustained adaptability in software?

We believe that the first challenge can be addressed by closer collaboration with contributors in the integration process and by new tools and techniques. If there were a distinct integration phase in the handover of contributions, a two-way knowledge transfer could occur: the contributor would receive free training on what is required to make a contribution to a multi-million user ecosystem rather than a research project. We as the future owners and maintainers would better understand the contributor's intentions, and be better prepared for future change requests. Unfortunately, there is currently no structure of incentives or recognition for being such a responsible contributor, and for taking the long view beyond the horizon of the contributor's immediate goals. A DOE-sponsored version of Google Summer of Code might be a good start.

[1] [gheber,epourmal,mfolk]@hdfgroup.org

The use of emulators, environments, and SDKs goes a long way to dealing with highly customized setups or setups that depend on special hardware, as long as there is a responsible party that maintains them at least for the contribution's design life.
The second challenge, we believe, is much harder and goes to the heart of this Call for Position Papers, the Science of Scientific Software Development and Use. Science, because it's about *understanding* the behavior of a complex adaptive system (interacting with other adaptive systems) before we can think about modifying or building parts of it. Science, because at this level we have to look beyond software at driving forces in the environment, the technology industry, an organization (The HDF Group), users, stakeholders, sponsors, economy, etc. Science finally, because we are dealing with emergent behavior: scientific software comes in totals greater than the sums of their parts: Software A or software B per se don't make scientific software. Instead, scientific software emerges at the "intersection" of many different ecosystems, each subject to its dynamics and reacting differently to the same stimuli.

Let's look at change and driving forces and how an ecosystem like HDF5 might be an ideal subject of study for a budding science of scientific software development and use!
Next to accelerators, perhaps the most important driver of change in computational science is data, its size, kind, its location, and how it's accessed. Hardware alone may not be sufficient to overcome "core data starvation." It appears that on the one hand, an entanglement of computation and data and the delivery of just the right data at just the right time might offer some relief. On the other hand, a new trade-off between data and computation in the form of computational storage is on the horizon, where data representation is fluid and tailored to computational capability. This is not just to better satisfy the needs of accelerators. One of the major roadblocks faced by computational scientists in computational application use is to be forced into a location-bound, fixed environment that lacks flexibility in a deployment that cannot easily or seamlessly scale up or down in computation *and* data, in model fidelity, etc. (Graceful degradation is *missing* in action!)
The DOE has shaped the history of the HDF5 ecosystem. But there were other other "influencers" such as NASA and NOAA. The HDF5 ecosystem is a gold mine of over 20 years of prime targets for scientific analysis by social and cognitive scientists, software "ethnologists," and AI NLP tools or copilots. There are very few projects with a comparable unbroken commit history, trail of release notes, and record of user discussions spanning over 20 years. We could correlate and study the HDF5 code base entropy and complexity over time, confirm a few known empirical findings, and perhaps add new findings as to which changes in culture, technology, economy, etc., had a lasting effect on scientific software development and use.

References

(Pines2014) David Pines, *Emergence: A unifying theme for 21st century science.* https://medium.com/sfi-30-foundations-frontiers/emergence-a-unifying-theme-for-21st-century-science-4324ac0f951e
(Woods2018) David D. Woods, *The theory of graceful extensibility: basic rules that govern adaptive systems.* https://link.springer.com/article/10.1007/s10669-018-9708-3

FOUR WAYS TO GROW SCIENTIFIC-SOFTWARE

SOU-CHENG TERRYA CHOI, YUHAN DING, CLAUDE HALL JR., FRED J. HICKERNELL, AND ALEKSEI SOROKIN

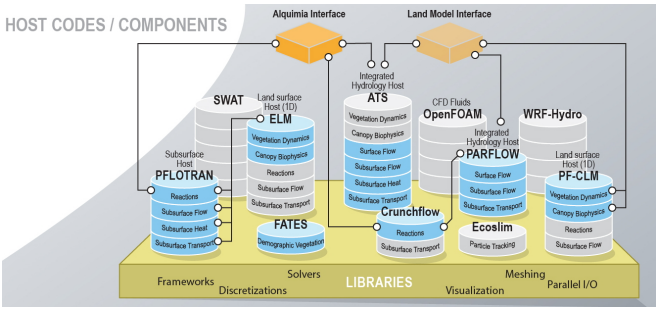
The opportunities for discovery, prediction, and optimization provided by advances in computer hardware and software are immense, but the challenges to taking full advantage of these opportunities are substantial. This position paper focuses on four challenges: i) connecting software libraries, ii) accurately estimating discretization errors from the computational data across multiple algorithms, iii) preparing developers and users to employ collections of libraries, and iv) educating developers and users to exploit large-scale computational environments.

1. Challenge

1.1. **Connectivity.** Developers wishing to contribute new (additions to) libraries and the practitioners wishing to utilize multiple libraries together find that basic mathematical objects may be encoded differently in different libraries. Must a function routine include its domain and the number of scalar inputs? Does an algorithm generating data sites provide a fixed number or an extensible sequence? What standards must developers of library enhancements meet so that their pull requests will succeed? The (quasi-)Monte Carlo and probabilistic numerics libraries that the authors know, such as [5, 13, 15, 16], lack adequate connectivity and consistency.

1.2. **Data-driven error estimation.** Complex computations may require the chaining of algorithms, e.g., the output of a (P)DE solver becomes the input of an optimizer. Since the next algorithm in the chain accepts inexact input from the previous algorithm, we need error bounds on algorithm output, as noted in [10]. A priori algorithm error bounds based on unknown norms of input functions are unhelpful. Reliable data-driven error bounds or estimates are needed. Moreover, error analysis for algorithms needs to factor in inexact inputs, and diagnostics should also be developed to ensure that necessary conditions for error bound validity are still met.

1.3. **Multiple Algorithms.** First computational mathematics courses [3] introduce students to numerical algorithms. Computational science courses [8, 14] combine numerical methods with modeling. Rarely are students taught to combine multiple algorithms to solve a problem. This leaves them unprepared for large-scale computational science, which requires not only multiple algorithms, but multiple libraries. The IDEAS Watershed project, whose codebase is depicted on the right [9], is an example.



1.4. **Large-scale computational environments.** Since the mid-1980s, computational scientists have become proficient at carrying out computations on their personal machines. However, many do not know how to take advantage of the variety of advanced architectures that are now common-place: GPUs, cloud computing, open science grid, and high-performance computing. Even with resources being made available through projects such as The COVID-19 High Performance Computing Consortium, many users and software library developers do not know how to take full advantage of them.

2. Opportunity

2.1. **Connectivity.** Code across multiple languages and platforms adheres to standards for certain data types, such as IEEE 754 for floating-point arithmetic. We propose that interested parties connected with larger scientific libraries develop shared standards for key objects that appear in algorithms, such as functions, sampling sites, and (partial) differential equations ((P)DEs). This would promote connectivity among software libraries and allow smaller libraries to be merged into larger ones. Developers could more easily contribute to existing libraries, and practitioners would find it easier to use multiple libraries.

2.2. **Data-driven error estimation.** Several of the authors and their collaborators have developed data-driven error bounds for multi-dimensional integration as well as univariate function approximation and optimization using numerical analysis and probabilistic numerics approaches [4, 6, 2, 1, 11, 12]. The key behind these error bounds is to identify reasonable cones of input functions that are not too spiky or peaky. This facilitates theoretical guarantees on the data-driven error bounds. The ideas referenced here could be extended to other algorithms such as differential equation solvers. There is also a need for the error bounds derived to allow for inexact input, which comes from chaining multiple algorithms together.

2.3. **Multiple Algorithms.** Computational science curricula need to be re-written to stress the combination of multiple algorithms, or even multiple libraries to solve problems, during introductory courses. To make room, we may need to eliminate detailed explanations of some numerical algorithms in the same way that we often avoid the details of the algorithm for computing the sine function. This curriculum revision may start by way of supplementary materials that will hopefully work their way into standard texts.

2.4. **Large-scale computational environments.** Educating the next generation to embrace cloud or cluster computing the same way that this generation views personal computing can be attained through partnerships involving academic institutions, national labs, and corporations offering test sites for students to learn and for developers to experiment. Software tools are also needed to help developers migrate their libraries from personal machines to environments taking advantage of multiple cores, GPUs, and other advanced architectures.

3. Timeliness

The Extreme-scale Scientific Software Development Kit (xSDK) is an effort to "address challenges in interoperability and sustainability of software developed by diverse groups at different institutions." The push to identify standards for objects such as functions and (P)DEs, would fit this program. The existence of a good number of large libraries makes it possible for healthy discussion among these communities to arrive at reasonable standards.

The rise in popularity of probabilistic numerics research provides one promising avenue for data-driven error bounds. So does the authors' work on adaptive algorithms.

The permeation of computing into multiple disciplines and the availability of multiple packages make it possible to bridge the gap between what is taught in the classroom and what is needed at the frontiers of research. Training grants may be used to develop and pilot programs that address what is lacking in our present curricula.

Just as large-scale computational environments banded together for COVID-19 research, they can do the same to provide basic competency to computational scientists in advanced architectures. This might look like Google Colaboratory but with access to more advanced architectures. Given the wider availability of these architectures, the time is ripe to provide developer tools to migrate libraries to exploit them.

References

[1] F. J. Hickernell and Ll. A. Jiménez Rugama. "Reliable Adaptive Cubature Using Digital Sequences". In: *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014*. Ed. by R. Cools and D. Nuyens. Vol. 163. Springer Proceedings in Mathematics and Statistics. arXiv:1410.8615 [math.NA]. Springer-Verlag, Berlin, 2016, pp. 367–383.

[2] F. J. Hickernell et al. "Guaranteed Conservative Fixed Width Confidence Intervals Via Monte Carlo Sampling". In: *Monte Carlo and Quasi-Monte Carlo Methods 2012*. Ed. by J. Dick et al. Vol. 65. Springer Proceedings in Mathematics and Statistics. Springer-Verlag, Berlin, 2013, pp. 105–128. DOI: 10.1007/978-3-642-41095-6.

[3] R. L. Burden, J. D. Faires, and A. M. Burden. *Numerical Analysis*. Tenth. Brooks/Cole, 2016.

[4] S.-C. T. Choi et al. "Local Adaption for Approximation and Minimization of Univariate Functions". In: *J. Complexity* 40 (2017), pp. 17–33. DOI: 10.1016/j.jco.2016.11.005.

[5] S.-C. T. Choi et al. *QMCPy: A quasi-Monte Carlo Python Library*. 2020. DOI: 10.5281/zenodo.3964489. URL: https://qmcsoftware.github.io/QMCSoftware/.

[6] N. Clancy et al. "The Cost of Deterministic, Adaptive, Automatic Algorithms: Cones, Not Balls". In: *J. Complexity* 30 (2014), pp. 21–45. DOI: 10.1016/j.jco.2013.09.002.

[7] R. Cools and D. Nuyens, eds. *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014*. Vol. 163. Springer Proceedings in Mathematics and Statistics. Springer-Verlag, Berlin, 2016.

[8] A. Holder and J. Eichholz. *An Introduction to Computational Science*. Springer, 2019.

[9] *IDEAS Watersheds Code Diagram*. 2021. URL: https://ess.science.energy.gov/wp-content/uploads/2021/01/IDEAS-Watersheds-software-ecosystem-road.jpg.

[10] I. Ipsen. *BayesCG: A Probabilistic Numeric Linear Solver*. Presentation at Dagstuhl Seminar on Proabilistic Numerical Methods – From Theory to Implementation. 2021.

[11] R. Jagadeeswaran and F. J. Hickernell. "Fast Automatic Bayesian Cubature Using Lattice Sampling". In: *Stat. Comput.* 29 (2019), pp. 1215–1229. DOI: 10.1007/s11222-019-09895-9.

[12] Ll. A. Jiménez Rugama and F. J. Hickernell. "Adaptive Multidimensional Integration Based on Rank-1 Lattices". In: *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014*. Ed. by R. Cools and D. Nuyens. Vol. 163. Springer Proceedings in Mathematics and Statistics. arXiv:1411.1966. Springer-Verlag, Berlin, 2016, pp. 407–422.

[13] P. Roy. *SciPy Quasi-Monte Carlo submodule*. 2021. URL: https://docs.scipy.org/doc/scipy/reference/stats.qmc.html.

[14] A. B. Shiflet and G. W. Shiflet. *Introduction to Computational Science*. Princeton University Press, 2014. ISBN: 9780691160719.

[15] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, version 2.28*. 2021. URL: http://mc-stan.org.

[16] J. Wenger, N. Krämer, and N. Bosch. *ProbNum*. 2021. URL: https://github.com/probabilistic-numerics/probnum.

(Choi, Ding, Hickernell, Sorokin) Department of Applied Mathematics, RE 220, 10 W. 32nd St., Chicago, IL 60616
Email address: schoi32@iit.edu, yding2@iit.edu, hickernell@iit.edu, asorokin@hawk.iit.edu

(Hall) Birmingham Southern College, 900 Arkadelpha Rd, Birmingham, AL, 35254
Email address: cdhall1@bsc.edu

The problem with modern CI

In the past, many researchers developed throw away code that used to produce results for a paper or a proposal, however, as software becomes more complex this approach does not work as well and scientific software has a much longer shelf life. Systems like the Visualization Toolkit (VTK) have been around for 26+ years. That increased shelf life requires more rigorous testing and development methodologies. For those methodologies to be adopted by domain specific scientific software developers, they must be easy to use and increase productivity. Many scientific software developers are not software engineers by training, but instead are subject matter experts that have picked up software development since software is the language of science. For this reason it is important for tools to be developed that automate many processes around good software development practices.

The CMake build system was created in the early 2000s to address this issue with respect to building software on multiple systems and compilers. Package managers like the Spack system have been created to help manage the build of large sets of loosely integrated software systems. In the area of testing software, there have been great developments in the area of modern continuous integration testing systems. Developers are able to have code automatically tested on a variety of target systems before that code is integrated into the main development branch of the code for others to use. Tools like github, gitlab and Jenkins all allow for this type of tested development. These CI systems are readily available to scientific software developers in most situations. Adoption of CI systems by scientific software developers requires ease of use and setup.

As good as these systems are at testing software, they are not designed to keep track of the software development process. As code bases continue to grow and the code has a longer life span, it is important that code once written continues to work as expected. There are tools available that can help with this process. For example coverage analysis tools, static and dynamic analysis tools, unit testing frameworks to mention a few. However, these tools are often not part of a scientific software developer's routine development process. For software to remain useful over decades it is crucial that all the software analytic tools available are deployed on a continuous basis as more code is added to the code base.

Modern CI systems are great at what they do. However, they have little ability to look inside the testing or build results. Most of the tools provide only a text log of build or test results. The CI systems break things down into individual development branches and do not provide needed insight into the health of the entire package. The integration of automatic software quality tools often requires software engineers to write python tools and do web development on top of the CI systems.

Scientific software development can benefit from the development of a tool that would automate the integration of various software quality metric tools like code coverage, static and dynamic

analysis, unit testing, and collection of build statistics. One such system is the CDash system which uses a database to collect and measure software metrics over time. The origins of CDash come from the six sigma methodology https://en.wikipedia.org/wiki/Six_Sigma. Although it is unlikely that software will be 99.99966% free of defects, there are lessons to learn from six sigma that can be applied to scientific software development. CDash can be used to measure the number of warnings/errors, tests passing/failing, percent of code coverage, and results of dynamic and static analysis tools. The concept is to create a software dashboard that is able to show the health of software at the moment and over time.

What is proposed is a software tool that is able to plug into modern CI systems and build tools and collect software quality metrics. In order to be useful the tool must be easy to deploy with a variety of popular CI systems. It should enable easy deployment of analysis tools like code coverage, static and dynamic analysis, and other tools as they are made available to software developers.

This tool would be able to collect data about the software and store it in a database. It would also aid in workflows integrating with issue trackers and other process tools. The data would be presented to the software development teams in an easy to use and useful way that would encourage the development of high quality software. The database of such a system could be input to AI algorithms to look for patterns in software development.

The goal of such a system would be to provide access to software quality tools to more scientific software developers. By automating the integration of testing and software quality tools and the collection of that data in a single easy to access data store it would allow for new types of software analytics to be performed.

A tool like this could be used by cognitive researchers to study software development productivity.

Theory-Software Translation Challenges

Caroline Jay (Caroline.Jay@manchester.ac.uk), University of Manchester
Robert Haines, University of Manchester
Daniel S. Katz, University of Illinois at Urbana-Champaign
Jeffrey C. Carver, University of Alabama

Software, specifically the scientific software development and use cycle, is now ubiquitous within research. In addition to the general challenges common to all software development projects, research software must also represent, manipulate, and provide data for complex theoretical constructs. Ensuring that this process of *theory-software translation* is robust is essential to maintaining the integrity of the science resulting from it. Better theory-software translation research could significantly advance both knowledge and scientific practice, yet there has been little formal recognition or exploration of the challenges associated with it.

In this position paper we present a summary of a published analysis [1] of the discussion sessions that occurred at the Theory-Software Translation Workshop held in New Orleans in February 2019, and use this to motivate future research. The workshop explored in depth the process of both instantiating theory in software—for example, implementing a mathematical model in code as part of a simulation—and using the outputs of software—such as the behavior of a simulation—to advance knowledge. It was attended by a range of people working within or studying HPC.

The results of our analysis demonstrate that research software is not merely used to perform a task, but to understand a problem and advance knowledge. While current software engineering research outputs and methods are relevant to addressing these challenges, theory-software translation research would involve tackling new problems that are rooted within the scientific domain. We summarise key, emergent areas for research as follows (see Jay et al. [1] for more details):

- The translation process moves from theory to algorithm to software (and vice-versa). Information is lost in moving from one domain to another as the way in which ideas are represented changes. Can we quantify or explain this loss/difference, and articulate the trade-offs resulting from translation?
- How does incorporating theory in software (e.g., a simulation) differ from standard requirements engineering? The development of software and theory happen together. While requirements changes are generally constrained for typical software, theory can change much more dramatically, resulting in not just an addition, but a fundamental divergence from what the software was initially designed to do.
- How do we understand the results of a simulation, and translate this back to the underlying theory? How should real world data be used in the verification process?
- How do we go from viable theory to validated, verified code in a time-efficient way? Can theory be expressed as a set of tests?

- There is a distinction between theory, model, numerics, and code, and there are difficulties mapping between them. There can be errors in any part of the mapping process that may affect the resulting science. How can we detect and handle these errors?
- Is a true separation of concerns (theory, and its implementation in software) possible? What are the implications for how we write scientific software?
- Should theory be recoverable from software (where software includes documentation)? Can theory today be represented solely in papers, or is it really also in the code? How can we help people to read and understand it?
- The increase in model complexity and sophistication of questions that models are designed to answer makes translating them into software increasingly difficult. How do we determine the appropriate level of complexity? Is it possible to optimise for complexity reduction, as well as performance?
- How can we define functional reproducibility? Long-term reproducibility is likely to remain out of reach due to software collapse. Can we create ways of representing theory that will persist and remain usable, so as to increase software sustainability and prolong the period of reproducibility? Is there a way of providing a reference implementation that can be used as a blueprint for the theory?
- Can we identify the range of practices currently used, and gather empirical data concerning their efficacy, to result in evidence-based best practice?
- What are the benefits and implications of the increasing use of automation in research software engineering?
- Software is unlikely to ever be 100% ‘correct’. Can we measure how well software represents theory, and estimate the error that might have been introduced during the translation process? Can we develop ways to quantify uncertainty for software functioning such that appropriate probabilities can be applied to results?

Studying these challenges will require a variety of methodological approaches, including but not limited to the hypothesis-driven scientific method, data mining, and qualitative research. It is essential we incorporate expertise from computational scientists themselves, and from the cognitive and social sciences as well as the software engineering research community.

As science and society become increasingly dependent on research software, and trust in science is debated and politicized, it is increasingly important to ensure that researchers, policymakers, and the public have faith that results are based on accepted theory that has been accurately translated into software. While accurate theory-software translation does not solve problems related to trust in science, inaccurate translation can destroy trust. Trust builds slowly over time, but can be lost quickly.

References

[1] Jay C, Haines R, Katz DS et al. The challenges of theory-software translation [version 1; peer review: 2 approved, 1 approved with reservations] F1000Research 2020, 9:1192 https://doi.org/10.12688/f1000research.25561.1

More Frequent and Regular Engagement for Success of Scientific-Software and HPC Software on Next-generation Supercomputers

Vivek Kale (vkale@bnl.gov)

Challenge

In the last decade, applications, i.e., scientific-software, have rapidly evolved given new numerical algorithms and new classes of applications, in particular those involving AI/ML techniques. At the same time, the platforms the applications run on have accelerated in development, with new hardware features in processors, new interconnect networks, heterogeneous processors on a node (CPU, GPU, FPGA), high-bandwidth memory, and lightweight operating systems. These rapid advancements in applications and platforms results in significantly rapid changes in HPC software products, e.g., LLVM's OpenMP, MPIch, Kokkos, RAJA, Charm++, BUPC++: new HPC software features (including improvements and bug fixes to existing ones) are implemented in vendor HPC software products in a short release cycle, every 2-4 months depending on the product. For example, LLVM's OpenMP went from version 10 to version 13 between March 2020 through October 2021, and has had minor releases and pre-releases in between each of the releases [1, 2]. Yet, the new features are adopted and tested in production on applications on supercomputers in biannual or yearly timescales, e.g., hands-on sessions in tutorials for ECP's HPC Software products like MPI(ch) and (LLVM's) OpenMP at the ECP Annual Meeting [3, 4]. Since application teams must digest and use HPC software all at once rather than incrementally in this scenario, development time for applications to use new features increases, which causes production-level applications to be delayed in using next-generation supercomputers efficiently. The lack of frequent testing of the latest HPC software products in applications or mini-apps brings down the quality of the HPC software, e.g., inefficiencies from memory allocation by LLVM's OpenMP on a GPU.

Opportunity

There is opportunity to improve the use of rapidly developed HPC software in scientific-software and the quality of HPC software through well-defined and frequent interactions between scientific-software developers and HPC software developers. We can have scientific-software teams and an HPC software developer from each product all devote a fixed time and duration, e.g., 3 hours a day for 3 days in a week, for a mini-hackathon at the end of each quarter to learn about and/or integrate new features of the HPC software products into scientific-software. To make sure the mini-hackathon is efficient and productive, we would: (1) request developers of each HPC software product to release stable versions of their latest product at the end of each quarter; and (2) ask teams to have a recent and stable version of their mini-app or benchmark representative of their full application ready at the end of every quarter (doing this will ensure that the HPC software developers' time is not unnecessarily spent on application-specific setup problems).

The research hypothesis is that regular and frequent mini-hackathons where scientific-software teams can meet with HPC software product developers will lead to scientific-software having (a) shorter development time to incorporate new HPC software product features and (b) fewer performance bugs. One could test the research hypothesis over the course of 2-3 years of quarterly mini-hackathons in a number of ways:

- Have multiple scientific-software teams port a mini-app each with different levels of support from HPC software developers and measure the quality of the code (correctness, performance, numerical reproducibility, performance variation) and the time to port the code.
- Have multiple teams port similar but different application codes, e.g., planewave DFT codes, and see how well each team is faring based on the support they get (not every team will want to participate in regular mini-hackathons).
- Measure different levels of support ranging from: (1) DIY; (2) some training; (3) engaging mini-hackathons; (4) formal training and hackathon; for a team and see how the development effort goes as a function of the level of support a team had.

Depending on the team, the experiments can come at a different cost (number of lines of code varies across applications) as well as a different quality of the results (teams may consider themselves done once they have ported only a few features or obtain a certain level of performance, doing the rest later). Given this, these experiments could be organized by running a competition where teams are awarded based on the amount of performance gains obtained or the number of new HPC software features used, normalized by the number of lines of code in their mini-app.

Timeliness and Maturity

Having more frequent and regular sessions of engagement between scientific-software developers and HPC software developers is mature because of advancements in software management and deployment technology and timely due to societal changes from COVID-19.

The advancement of software packaging tools for HPC such as Spack [5] make building and using new versions of HPC software on supercomputers much more streamlined. Without such tools, there is a large startup overhead by scientific-software developers to build each new version of HPC software. Reducing this startup overhead makes it more viable to have mini-hackathons in which scientific software developers can quickly get set up in order to focus on programming new features in the new version of HPC software in their scientific-software.

Before the COVID-19 pandemic, software and hardware companies that develop HPC software could easily send its developers to work with scientific-software developers. With the pandemic, the HPC software developers at these companies as well as scientific-software teams are more remote and this may continue even after the pandemic. The nature of increased remote work makes it timely to have more frequent and well-defined interpersonal interactions between scientific-software developers and HPC software developers.

References

1. GitHub for LLVM Project. https://github.com/llvm/llvm-project/releases. Accessed 11-23-2021.
2. LLVM Download Page. https://releases.llvm.org/. Accessed 11-23-2021.
3. Webpage for ECP Annual Meeting. https://ecpannualmeeting.com/. Accessed 11-23-2021.
4. Webpage for ECP training. https://www.exascaleproject.org/training-events/#upcoming. Accessed 11-23-2021
5. Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott 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.

Software sustainability is people, at least for now

Daniel S. Katz, University of Illinois at Urbana-Champaign, d.katz@ieee.org
Michelle Barker, Research Software Alliance, michelle@researchsoft.org

Scientific software, like most other software, cannot be thought of as a single isolated package but is typically either a collection of software packages, and/or has dependencies on various other software (operating systems, compiler, libraries, etc.) Hinsen [1] describes typical layered structures that underlie the software used in research, and explains how this structure encodes dependencies, where the top-level software that is directly used to do research depends on multiple layers of other software and hardware, and compares the fact that elements of these layers change over time to designing building in an environment where the underlying layers (e.g., soil, foundation) can shift.

The fact that software must be maintained to continue working (adapting to discovered bugs, changes in the hardware and software environment, and changes in user needs) leads to a pair of top-level challenges:

1. How much of this can be automated, and how can this best be done? (e.g., how much could introspection and automated testing, or automated code generation, help?) *This is an area for fundamental computer science research, and an area where the impact of the research could make a large difference in the long term, as it has the potential to scale the impact of investments and resources.*
2. For the parts of the work that either cannot be automated or are not automated, how do we encourage the human beings who alone can do this work to actually do it? This is an area where computer science research must be brought together with sociology, ethnography, economics, and other studies of human beings and their culture, and any changes we can make will have immediate impacts, though they are not particularly scalable, as a single person can only directly impact a limited number of software packages and software communities.

In this position paper, we focus on the second question: promoting the key role of people in all stages of software development, maintenance, and use, and discussing associated research challenges. The Research Software Alliance performed a set of interviews and analysis of 28 research software initiatives to examine major issues related to people (or personnel) challenges faced by the research software community, leading to the ReSA People Roadmap [2]. This roadmap considers the activities of these initiatives as part of one or more of five themes, namely:

- **Policy**: activities that aim to influence the policy or actions of other stakeholders, not only one's own organisation
- **Incentives**: ways to improve the establishment, recognition and progression of research software personnel roles
- **Communities**: groupings that enable people with common interests or characteristics to come together to achieve shared goals
- **Skills and training**: projects that support skills and training efforts beyond those for their own staff, and include designing curriculum, training trainers, identifying skills frameworks, and convening training and mentoring events
- **Infrastructure**: the components used to support the role of software for and in research practices, including the schemas and standards that support this

All of these themes are related to the second research question. What **policies** are most effective in creating a system where this human work happens? What **incentives** best encourage the people to do this work? What is the role of **communities** in supporting, encouraging, and enabling this work; and how can their diversity and inclusiveness be increased? What **skills and training** are needed to make the people most able to do the work? And what **infrastructure** is needed to ease the work?

There has been good progress made in many of these themes in activities such as IDEAS and BSSw, US-RSE, URSSI, SSI, ReSA, Software Citation, FAIR4RS, etc., and such work has focused on making changes in software

projects, journals and conferences, repositories and archives, and professional societies and informal organizations. But one area in which little progress has been made is the role of institutions, which we examine in relation to these themes in the remainder of this position paper. Some specific challenges in which research is needed are:

- How can institutional (university, government and private laboratory, and industry) policies and practices around hiring, evaluation, and promotion be changed to
    - Better support research software development and maintenance?
    - Encourage contribution to existing open source projects?
    - Encourage existing members to develop and maintain the skills needed to work in research software, including both computer- and people-oriented skills
    - Support diversity, equitable and accessible practices and communities?
- How can institutions train the larger community to use and contribute to the software developed in the institution?
- What external policy changes (e.g., from governments and funding agencies) and external cultural changes (e.g., from professional societies and publishers) could encourage and support these institutional changes?
- What infrastructure is needed across institutions to support this, such as software repositories to support citation, software registries to facilitate re-use, skills and training registries, etc.?

To move forward, we suggest a workshop that starts with these questions, and then proposes hypotheses along with activities to test them. This could include things that one institution was already doing that another institution could trial, or completely new ideas. Such a workshop would require four types of attendees: research software engineers, research software engineering researchers, organizational practice researchers, and institutional leaders who have the ability to make changes in their own institutional policies. Members of the first three groups will generally be willing to participate, but the participation of the latter group might benefit from direct funding agency (DOE, NSF, NIH) involvement in the workshop organization or at the least, strong formal or informal encouragement.

The intent of this workshop would be to

1. Understand the current state (of institutions and their policies that affect software development)
2. Discuss potential research questions in this area
3. Instigate new research collaborations to answer these questions, and initial plans for such collaborations, including steps, resources, skills, etc.
4. Consider how the results of the research collaborations and finding should be disseminated to create impact

A post-workshop report would include sections on each of the four points above: a landscape and examples of policies and their effects, potential research questions, plans for answering them, and ideas about dissemination of results.

This is timely due to increased general awareness of the importance of research software, new funding programs and policies from public and private research funding agencies, and the beginning of inclusion of software in international principles and national policies around research openness, availability, and reuse.

References

[1] K. Hinsen, "Dealing with Software Collapse," *Computing in Science & Engineering*, v.21(3), pp. 104-108, May-June 2019. https://doi.org/10.1109/MCSE.2019.2900945

[2] M. Barker, "ReSA People Roadmap Report," October 2021. https://doi.org/10.5281/zenodo.5633318

Software as a Scientific Instrument

Kate Keahey, Argonne National Laboratory

Throughout history, scientists have had to rely on scientific instruments to be able to deploy, capture, measure, and analyze the investigated phenomena. Often, it was up to the scientists themselves to design the instrument, build it, test it, and improve it, before eventually using it as part of the process of scientific discovery. Examples include scientists like Antonie van Leeuwenhoek, who invented a microscope enabling many great microbiological discoveries or John Harrison, who invented the marine chronometer, solving the problem of longitude in sea navigation. The building of such scientific instruments has always been held in high regard -- for example, John Harrison received the Copley Medal bestowed by the Royal Society of London from for his contributions (alongside Charles Darwin, Dmitri Mendeleev, and Albert Einstein) -- in recognition of the fact that building such instruments is typically a scientific research venture in itself [1].

With the advent of the digital age, scientific instruments are typically built of a much less tangible -- though more adaptable -- fabric: software. Apart from their digital representation, and the unique challenges the software opportunity entails, scientific software instruments (SSIs) are not substantially different from traditional physical instruments, and their construction continues to be a scientific problem in its own right. However, despite the demonstrated impact they have on many groundbreaking scientific discoveries [2], the design, the evolution, and operation of SSIs are relatively little studied, and the work of their constructions often does not get the credit it deserves. I describe below some of the challenges that arise.

Software scientific instrument (SSI) design, evolution, and maintenance. While teams designing SSIs area often take inspiration from industry in process adoption its dynamics are significantly different since designing a viable instrument is a research task in itself. This means that the requirement discovery tends to be both more challenging, more directed, and less discretionary. Further, the dynamics of the construction effort are strongly influenced by funding structures with more directed effort and significantly lesser proportion of creative destruction. At the same time, experience shows that any scientific development effort will take place with an ecosystem of existing software and will have to adapt to it long term [2]. What is the best methodology (or methodologies) for designing scientific software and infrastructure instruments? How can we adapt existing best practices (or create new ones) to guide the design and development of scientific instruments? Some ideas to implement in this space could be assessment of large projects and/or development efforts, establishing what worked and didn't work, how the software efforts evolved, and formulating recommendations.

Intersection of infrastructure and software. We tend to think of scientific infrastructure as either a defined physical systems (e.g. large scientific instruments such as telescope arrays, tokamaks, etc.) or datacenter investments operated by a narrowly defined range of systems tools. However, with the proliferation of IoT/edge devices increasingly developed and managed by owners, this understanding is evolving to increasingly mean custom built instruments

connected by software. With edge devices going out of the datacenter, scientific infrastructure, like software, will be increasingly designed to solve specific problems rather than provide general services and thus the responsibility of individual research groups. What role will software play as hardware increasingly leaves the datacenter, how will it evolve, and how will that impact its development practices? How can we make it efficient in an era where it is increasingly customized? How can we anticipate developments in this space?

Sharing, publication, and recognition. While designing scientific software/infrastructure instruments is a scientific task, it was not often recognized as such and there is currently an open debate on how to best implement such recognition. For example, on one hand, there are attempts to adapt the traditional scientific publication process, with closed peer reviews (Softwarex), on the other hand, there are attempts to adapt code review process as often carried out in software development (JOSS). What is the best way (or ways -- for different circumstances) to share and/or publish software? How should we mange recognition and credit to software designers?

Software and reproducibility. Software often underpins the reproducibility of digital artifacts but it is not always well understood how. For example, there are challenges related to the timeliness and lifecycle of software products: to reproduce a dataset in domain sciences may require the availability of a solid and maintained software instrument at long timescales, while the repeatability of a systems experiment for the purposes of review, validation, and inserting creative variation may require only a simple image + notebook packaging of a prototype available in a defined but shorter timeframe. What are the guidelines for assessing software suitability for use cases of interest to science? How should we manage their timelines?

Synergies between CS research and production science. Productive interaction between CS research and production science is often critical to progress but very hard to orchestrate: CS scientists often produce prototype quality code to support the publication of results, while scientific instruments require high quality code that does not always represent innovation in the CS field. Articulating scientific requirements is extremely hard and requires much experience, yet the benefits of doing that are often indirect. At the same time, many innovative instruments rely on the “latest and greatest” advances in systems CS, so long as they are expressed in production quality software. What are the best ways of creating an effective “pipeline” between CS research and production science? What are the right mechanisms to use for software support as opposed to innovation?

References

[1] Scientific Software Instruments, Frank Seinstra, David Wallom, Kate Keahey, SoftwareX, Vol 1-2, September 2015, Pages 1-2
[2] Finding the Gravitational Wave: A History of Discovery Written in Software, Couvares, Peter ; Keahey, Kate ; Marion, Frédérique SoftwareX, Vol 15, July 2021, Pages 1-2

High Energy Physics and Scientific Software Development

Jim Kowalkowski (jbk@fnal.gov), Fermi National Accelerator Laboratory
Marc Paterno (paterno@fnal.gov, contact), Fermi National Accelerator Laboratory
Saba Sehrish (ssehrish@fnal.gov), Fermi National Accelerator Laboratory

We are computational scientists with more than 50 years of combined experience in working with high energy physics (HEP) domain scientists. We have worked with experimental collaborations ranging from a few tens of physicists to more than 2000. Along the way we have been involved with the transition of the HEP field from Fortran to C++, including the design and implementation of HEP event processing frameworks (used by CMS, MicroBooNE, Muon g-2, the other Liquid Argon experiments at Fermilab, as well as DØ and CDF). These frameworks are what allowed the contribution of algorithms by tens or hundreds of scientists to a collaborative effort. We have conducted reviews and redesigns of large bodies of physicist-written code, and modified code from traditional grid-oriented processing to HPC-oriented designs able to perform in a day what would traditionally have taken months to accomplish. We are aware of a significant number of specific challenges to be overcome in order to improve the process of software development for the HEP community. In this short note, we describe several of the issues we have confronted, with varying degrees of success, across many software projects with several experiments.

The work of many domain scientists in high energy physics (HEP) includes writing software for all stages of an experiment -- design, planning, data collection, and analysis. Much of this software is developed by graduate students and postdocs, most of whom have no training in programming and almost none of whom have training in software design. The reward structure of the field stresses the importance of the production of physics results in the timescales relevant for the advancement of scientists' careers -- typically a few years. Thus little care is given to the long-term maintainability of the software they produce. HEP experimental collaborations typically do not have formal reviews of design before implementation, or of the implementation that gets done. However, the timescales of experiments are much longer than those of students and postdocs, spanning decades in the design, construction, commissioning, and running, and then often upgrading, of a modern HEP experiment. Thus the large bodies of software (both CMS and ATLAS have about 5 million lines of C++; the software of other LHC experiments are of comparable size, as we expect DUNE to be as well) of modern experiments are very difficult to update to take advantage of new technologies and techniques, and so experiments are slow to adopt new inventions.

Because of the pressure to get something working as soon as possible, little or no time is spent on designing software. Software engineering practices commonly recommended in professional literature are either unknown or ignored. Components are not designed for testability, and thus

testing of critical software components is often done only as acceptance testing (called “physics validation”) for very large programs. This is a task that frequently involves dozens of scientists and months of work. Since the change of an individual component requires the re-validation of a very large ensemble of software, the result is a great reluctance to introduce novel computing technologies.

HEP experiments are very reluctant to share their data, and in some cases their software, with others who are not members of their collaboration. The data are proprietary, at least for a period of time before publication, and the software is the encoding of analysis that might not be mature and ready for publication. The need for discretion before the publication of new results is real, but it can hinder collaboration with non-HEP physicists who are interested in learning about and improving the computing techniques used. Building a relationship of trust between HEP collaborators and computational scientists -- or ASCR institute mathematicians and computer scientists -- takes much time and effort.

It is a challenge for computational scientists who are not members of the experiments to clearly understand the relative priorities of the collaborations, especially since the members of the collaborations are themselves not always in agreement on these priorities. This lack of understanding makes prioritizing the possible improvements to the software difficult.

It is a challenge for experimenters who are not experts in the mathematical and computing domains to understand what may be computationally feasible using new techniques and the vast computational power of DOE computing resources. The senior experimenters who lead experiments and analysis groups within the experiments are often not sufficiently familiar with advances in computational capabilities, and a certain amount of institutional inertia must be overcome to convince an experiment to make use of new techniques. For example, we are still working on getting experimenter approval of a calculational improvement that changes an analysis code from using a calculation that is done on a naive rigid grid to the use of a multivariate functional approximation, which is both faster and removes artifacts introduced by the rigid grid. Because this technique is new to the experimenters, the task of building confidence in it among the experimenters is difficult. This seems to become more difficult as the size of a collaboration increases.

As the participants of this workshop develop the knowledge discovery process that might come from this workshop, to be used as part of this scientific methodology, we believe the collaborative team we have formed could be a valuable partner in the scientific study. We have a team with HEP experimenters and ASCR institute members, which has had some significant successes, and sometimes less success. We believe that the social and cognitive scientists might be able to learn from our experiences what works and how to make the development process better. In addition, our collaboration could benefit from whatever these scientists learn that could help improve how we work.

Sustainability of Scalable Computational Chemistry Software at PNNL

Karol Kowalski,[a] Erdal Mutlu,[b] Ajay Panyala, [b] Sotiris Xantheas [b]
[a] Physical Scienced Division, Pacific Northwest National Laboratory
[b] Advanced Computing, Mathematics and Data Division, Pacific Northwest National Laboratory
karol.kowalski@pnnl.gov, erdal.mutlu@pnnl.gov, ajay.panyala@pnnl.gov, sotiris.xantheas@pnnl.gov

Our position paper capitalizes on PNNL's extensive experience and track record in developing, maintaining, and disseminating scalable scientific software for computational chemistry as well as building a user base among the scientific community through both training and consulting. PNNL's current computational infrastructure supports NWChem,[1] our flagship scalable electronic structure software, and leads/contributes to several of its offspring such as SPEC,[2] Quantum Chemistry (QC), and NWChemEx.[3] These domain software suites integrate novel programming models, applied math algorithms, and theoretical formulations to tackle complex chemical processes and transformations on extreme scale computer architectures at DOE's Leadership Computing Facilities (LCFs). Our position paper emphasizes the role of *co-design* in developing, tuning, and maintaining scientific software in the Chemistry domain.

Challenges: Most of the scalable implementations of the electronic structure methods in NWChem/NWChemEx/SPEC/QC are dominated by contractions of multi-dimensional tensors describing either the Hamiltonian operator or the wave function parameters. Enabling high-accuracy, highly correlated electronic structure models across spatial and temporal spaces is contingent upon the ability to take full advantage of existing extreme- and forthcoming exa-scale resources in order to effectively and efficiently execute these mathematical operations. As a complex process taking place at the nexus of computer science, applied mathematics, and the target domain science, the development/evolution of scientific software consists of several levels associated with the following challenges: (1) the ability of the implemented functionalities (underlying physics) to provide an accurate and comprehensive description of outstanding chemical processes and transformations, (2) the efficient utilization of the continuously evolving hardware at LCFs, (3) the flexibility of the software to adapt to the emerging computing technologies, and (4) its widespread utilization by an engaged user base in the academic, industrial, and the national laboratory ecosystems to deliver cutting-edge science-based applications that cannot be otherwise realized. Additional logistic roadblocks are associated with; limited access to LCFs, the time-consuming process of applying for early access to computational resources at LCFs; the fact that experimentation and performance validation must be performed on oftentimes deficient computational platforms; and the lack of interaction between software developers and early access application users to demonstrate groundbreaking usefulness of resources in addressing domain science problems.

Targeting the efficient implementation of software on LCFs requires taking advantage of the latest technologies available in hardware and memory management. The fact that the design of the software will need to overcome hurdles and bottlenecks in parallelism and communication across distributed nodes of a relatively new architecture may severely affect deliverables and schedule. This can be further amplified by delays in delivering hardware from the vendor or issues with the system's stability and mean-time-to-failure of the individual nodes.

Opportunities: To streamline the process of software development, mechanisms to embrace hardware evolution and advances in computer science, applied mathematics, and domain specific physical models should be in place. With the emergence of exascale computers, factors that will allow the software development effort to fully exploit these resources include the increased scale of parallelism, scientific ML/AI support through greater hardware heterogeneity, complex memory hierarchies, data abstraction, resilience, fault tolerance, checkpointing, and energy efficiency. The computation needs to be expressed as tasks managed by an intelligent runtime that understands the algorithmic and data dependencies, machine resources, memory/communication subsystems, and performance portability to future machines while maintaining scientific productivity. A multi-faceted co-design effort is needed to provide an abstraction layer enabling tensor contractions in a high-level language that automatically generates a parallel and optimized C++ code, providing back-end flexible infrastructure to specify and manipulate data distribution, manage memory, schedule tensor operations, and support a general notion of index spaces, allowing the

development of a much larger class of methods. Some of these principles are already implemented in the TAMM infrastructure.[4] The diversified co-design groups can identify and address additional opportunities stemming from the hardware evolution and adopt advances in domain science while catalyzing productive collaborations and attracting developers and potential users. This can be achieved by ensuring its availability, support, portability, and robustness on the targeted architectures and establishing continuous feedback from the user base regarding its usefulness and additional functionality needed.

While the adoption of best practices (methods, software) is rapidly increasing, the extent of implementation is still limited, mainly due to lack of resources (time, manpower, funding). In general, adopting best practices and software (e.g., workflows, libraries) from other software communities can be productive in the development of scientific software since it reduces the need to research, develop, and maintain such capabilities. However, depending on the context, adopting a specific software or best practice from a different community might not always work out of the box, a case that will require additional investments in time and manpower to adapt it into a new scientific software infrastructure. Adopting practices designed with a different target domain in mind might not directly fit seamlessly into a new scientific software domain. It would be important to recognize these challenges and consider them towards the overall success (efficiency, portability, scaling, robustness) of the scientific software being developed. Finally, the rapid progress in changing the computational paradigms poses many opportunities associated with the optimal utilization of computational resources, data movement, and representation, causing brute force approaches based on a mere increase in the number of computational units to inevitably reach their limits. It also defines the way in which ML and HPC can coexist on exascale platforms providing computational chemistry capabilities that would require beyond-exascale resources if relying only on an "inner computing" approach.

Timeliness: The emergence of the first exascale architecture coupled with the increasing involvement of ML/AI in both domain method development and software design paves the way for a true co-design approach. For the US based computational chemistry software to remain competitive, new computational technologies should be quickly adopted. This effort should include:

- *High-level programming languages* (i.e., Python, Julia etc.), and *libraries with high level abstractions* (i.e., NumPy, TensorFlow etc.) that allow for the rapid development for different domains. While this approach can decrease the development time, it lacks the ability to fully leverage modern HPC architectures, where it is crucial to exploit hardware features by using lower-level programming languages (i.e., C/C++ etc.) and architecture dependent optimizations.
- *Cloud computing* as an alternative to HPC systems. While recent performance of HPC systems has made them *the* solution for most current science problems, their limited availability makes it difficult for domain scientists to rapidly test new theories and obtain timely results. The emergence of ML/DL based solutions has inspired computing providers to heavily invest in improving on all these shortcomings resulting in very competitive comparison to HPC systems.[5]
- *Hybrid conventional/quantum computing* integrating emerging quantum computing technologies with high-performance computing platforms, where numerically intensive elements of calculations are delegated to the quantum hardware, while elements of the simulations that cannot or should not be performed on quantum platforms are performed using conventional computing architectures. This is an emerging area that will continue to benefit from breakthroughs in both quantum computing hardware and algorithmic developments.

References

1. E.Apra, *et al*. "NWChem" Past, present, and future", *Journal of Chemical Physics* **152**, 182102 (2020).
2. https://spec.labworks.org/home
3. K. Kowalski, *et al*. "From NWChem to NWChemEx: Evolving with the computational chemistry landscape", *Chem. Rev*. **121**, 4962 (2021).
4. Erdal Mutlu, *et al.* 2019. Toward generalized tensor algebra for ab initio quantum chemistry methods. In *Proceedings of the 6th ACS SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*
5. Giulia Guidi, *et. al*. 2021. 10 Years Later: Cloud Computing is Closing the Performance Gap. In Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '21).

Capturing the varied complex software lifecycle processes at a scientific user facility

Hari Krishnan (HKrishnan@lbl.gov), Advanced Light Source and Scientific Data Division
Alexander Hexemer, Juliane Reinhardt, Dylan McReynolds, Advanced Light Source Computing Group
Padraic Shafer, Dula Parkinson, Advanced Light Source Photon Science Group
Lavanya Ramakrishnan, Shreyas Cholia, Scientific Data Division
Lawrence Berkeley National Laboratory

The Advanced Light Source (ALS) at Lawrence Berkeley National Laboratory is a BES user facility. It has approximately 40 beamlines with techniques including diffraction, spectroscopy, and imaging, and these instruments serve over 2000 users per year from many scientific disciplines (life, earth, environmental, chemical, physical, and material sciences). Thus, light sources, in general, are not a monolith, but clusters of multi-disciplinary scientific teams with the experimental facility serving as the backbone for enabling experimental science. Traditionally, these teams have diverse strategies for software development typically informed by the constraints of experimental setup, skill sets of the teams, and resources. This ecosystem results in processes and practices that are as varied as the experiments themselves. Additionally, while each of these teams is not large, taken in aggregate, the amount of data produced has been increasing exponentially, and now stands at multiple petabytes per year with projected increases of several orders of magnitude with the ALS upgrade within the next few years.

Challenge: While staff and users have always coordinated and collaborated as significant developers and users of scientific software at their instrumentation, the increasing data rates and complexity of many of the experiments -- with a continuation of the huge diversity -- has meant a critical need for productive and sustainable software development processes. We believe that ensuring a long term sustainable operation of our scientific user facility without sacrificing the quality of experimental science will require a much more coordinated end-to-end strategy for services and resources provided at each layer of the experimental pipeline.

To transition light sources to state-of-the-art data and computational lifecycle processes, it is important to look at how software, data management, and analytical pipelines have been developed in practice. In our experience, there are a few coordinated software development efforts with long term software and data lifecycle considerations, but a majority of the efforts are by one or a small group of individuals whose knowledge and experience in software development is not at the leading edge in the field--they have domain science expertise and just develop software to a minimal level that “works”. There are multiple barriers that keep these developers from engaging with more best practices, or that keep experts in the field from engaging with these developers, including:

- Challenges with communication: The non-expert developers and those who study software development don’t speak the same language, and we believe that overcoming this barrier is a months-long process at minimum, which is a significant investment.

- Local vs global solutions: Because the impact of any one of these software projects is relatively small, the payoff for those working on better software development to engage with any one of these efforts is small. Additionally, the non-expert developers are often reluctant to take the time to engage with software development researchers because they are trying to put minimal effort into something that works rather than learning to develop the best possible software.

Due to the challenges listed above, a prerequisite for eventually having software that follows best practices would be that computer scientists or engineers be more involved in the individual projects. These embedded computer scientists could be the interface to those who study software development and be the ones to implement recommendations for improved practices. However, given the wide array of

demands on scientific user facility budgets, it is not possible to have a large team of computer science staff. Thus it is critical to efficiently use these limited resources, and this collaboration between computer and domain scientists must use sustainable solutions optimized with experimental variability in mind. Any solution will need to scale in the ability to manage and support the scientific software for tens of beamlines (hundreds of beamlines across all DOE BES light sources).

An important consideration in deploying computer scientists across more projects with domain scientists is maintaining proper credit for work from various contributors--there has been frustration on both sides in the past about who can claim credit for developments. Another important consideration is the need for the software to be truly supportable in the 24/7 production environment of a beamline, which either requires computer scientists to be on call 24/7 or for the software to be developed so that domain scientists understand it deeply enough to be able to support it alone.

Opportunity: Because of the large number of software development projects, along with the large number of users who use that software, efforts to improve software at the ALS and other light sources would have an important impact on a large number of researchers. The challenges described above require long-term commitment with a sustainable strategy and vision to overcome the barriers for adoption, only then would these be viewed as useful or welcome--short term “demonstrations” and “pilot projects” are of less interest.

Overcoming barriers: Software with institutional support can be the key difference between one-off and long term manageability for any critical component of a piece of software. Best practice guidelines for non-expert developers, frameworks, and effective communication on software development practices as a critical component is needed to be within the toolbox of the future. Questions on: how do we balance scientific and facility capabilities, transitioning pre existing software and practices, and ensure trust that new methods and practices are as effective as current methods?

Capturing variability: One of the challenges/complexities at the DOE user facilities is also the software that comes from the users that you may or may not have much control or even visibility into that makes software testing and integration significantly more complex. How do we develop software processes and environments that can support this diverse, rich, and complex software ecosystems at user facilities?

Timeliness or maturity: The ALS is in the middle of a 450 million dollar upgrade project which will be finished by around 2026 and will leave the ALS as a much brighter X-ray source, able to produce orders of magnitude more data than it can now. The same is true for light sources across DOE, with NSLS-II and LCLS-II having been recently upgraded with the APS upgrade currently underway.

A unique element of workflows at user facilities is exploratory science. Creating rich and diverse software ecosystems that encapsulate the varied workflows requires balancing variability of scientific workflows and managing data flow from real time to archival. Two unavoidable truths exist at light sources: Data will come in brighter/larger/faster, and the complexity of experiments will grow. User needs for software include sifting/filtering useful signals from background & artifacts, modeling diffraction patterns and other photon-matter interactions, phase retrieval, and many more. Much of the speed and spatial resolution of the newer characterization techniques comes at the cost of not being able to directly interpret the data without an arsenal of modeling, fitting, and other optimizations to make the solutions tractable.

Having software processes that provide effective tools strengthens the urgency and importance of investing in the long term developments that moves the facilities and users towards a brighter future (pun intended).

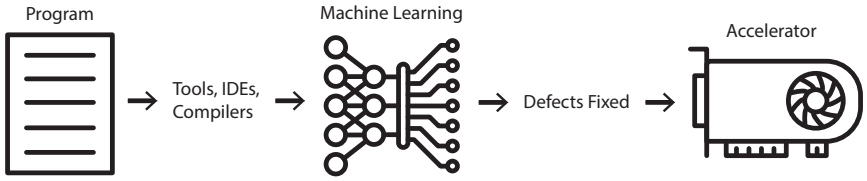
Compiler Analysis for Early Detection of Software Defects: Challenges and Opportunities

Ignacio Laguna (contact), Giorgis Georgakoudis, Harshitha Menon, Konstantinos Parasyris
Contact: ilaguna@llnl.gov, georgakoudis1@llnl.gov, gopalakrishn1@llnl.gov, parasyris1@llnl.gov

Keywords: static analysis, dynamic analysis, machine learning, clang/LLVM, tools

Challenge

As the software stack complexity increases, it becomes imperative to help programmers detect and fix software defects as early as possible. Software is continuously ported to multiple platforms (many of them heterogenous), which significantly complicates diagnosing defects. In addition, many issues only manifest during production runs [1], which could be detected earlier. There have been significant improvements in techniques and tools to analyze code statically and dynamically to fix performance and correctness defects. However, those approaches are applied usually at the late stages of programmers' workflows. The challenging question we seek to address is: ***Can new static and dynamic analysis methods be more helpful to programmers if they are designed to be applied at earlier stages of the development process?*** We describe possible opportunities in three areas: (1) advanced programmer guidance; (2) compiler assistance through AI; (3) better IDE and CI support.



Opportunities

Advanced Programmer Guidance

Performance defects can be tricky to detect, and most importantly, to fix. While there is a plethora of runtime performance analysis tools, fixing performance regressions is still a challenge in scientific codes. Many tools provide a significant amount of performance data, such as code hotspots and hardware counters; however, they provide little guidance to programmers on fixing such defects. Correctness defects in HPC are even more challenging to diagnose than performance defects since few tools exist for parallel code (robust correctness tools are mostly designed for non-HPC programming models). More importantly, when these tools are applied, they are activated at a late stage in development workflows, which means that sometimes fixing defects may require re-designing the whole application, a productivity drain.

There is significant opportunity to help programmers develop more robust and efficient code, if static and dynamic analysis tools are re-designed to be useful at early stages in development. For example, consider the problem of enabling compiler optimizations on code; previous work [2] has shown that changes in the source code can unlock further compiler optimization, improving performance up to 2x. Modern compilers, such as `Clang/LLVM`, include facilities (see `clangd` [3]) for on-the-fly compilation and program analysis and report information on compiler optimization and analysis (for example see LLVM remarks [4]). Further on performance, past work [5] has shown that piecewise extraction and benchmarking of code regions provides accurate metrics of performance. Regarding correctness, previous work [6] demonstrated that compiler static analysis is a powerful tool for detecting data races. We propose that these methods for detecting performance and correctness defects are available to programmers early in the development process, while the application design takes place, to guide programmers.

Compiler Assistance Through AI

As machine learning (ML) and artificial intelligence (AI) become more advanced, there is an opportunity to involve ML/AI in the development process. ML/AI have the potential to make programmers more productive and help them increase confidence in the correctness of their codes. Existing approaches for classification could be adapted to encode and solve static analysis problems, such as the following:

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-829147).

1. **Identification of erroneous patterns:** classification or regression models can be trained to detect patterns in the code that are known to be erroneous or problematic. Some of the challenges in this area involve encoding programs in a representation so that existing advanced classification models are reused, e.g., those used in image classification or natural language processing. These representations should not only capture programming semantics, such as loops and functions, but also encode information regarding the algorithm, for example, representing in abstract forms the specification or intention of an algorithm, such as denoting sorting algorithms or parallel reduction techniques. Another challenge is the generation of large amounts of training data—auto-generation of large numbers of small programs (which can be automatically labeled) could help.
2. **Natural Language Processing (NLP) for pattern extraction:** NLP could be used to extract or even create "best practices" from existing programs to help (via static or compiler analysis) programmers. In the former case, NLP techniques can be used to analyze already existing best practices manuscripts and guide the developer to perform code changes to follow the latest advances. On the latter case, NLP techniques can analyze code, written by domain experts, and independently learn best practices on a domain. Some of the challenges of applying NLP is labeling the data and requiring advanced feature reduction techniques as the mapping space from programs to patterns can be huge.
3. **Computation model detection:** can a compiler reason about the implementation of a mathematical model or computation? For example, can static analysis detect accurately that a given code region is performing a matrix multiplication and suggest to the programmer more optimal or correct ways to write such a program?

We believe these questions can be solved by integrating ML/AI techniques early in the development process.

Better IDE and CI support

We believe that static and dynamic analysis methods could become more useful (and potentially mainstream) if they are better integrated into development tools and frameworks, such as IDEs and CI (continuous integration) frameworks. Focusing efforts on defect detection in IDEs—possibly as programmers type code—enables fixing issues at an early stage. With porting being common in scientific codes, a significant challenge question is: ***How to enable IDE technologies for testing and defect detection on different platforms?***

We need to enable building and testing of HPC software quickly and reliably across different platforms, across codebases, and for multiple users. For the build and test system to be user-friendly, solutions need to provide a uniform, simple, and extensible language to specify the build process, and for it to be fast and reliable, it needs to interface with the package management system used in HPC, such as `Spack`, to enable dependency analysis, advanced caching, and reproducible builds. It will also be necessary to provide interfaces to job schedulers to enable the users to build and test on clusters with different architectures. To truly be a part of the development process, this needs to be integrated with commonly used IDEs and with a single command or a click of a button. The goal is to provide the user with a build/test system that seamlessly integrates with the current development process and enables the users to easily and reliably build and test on large inputs and different architectures.

Timeliness

Addressing the above challenges is now more feasible than before and the solutions would have significant impact to DOE scientific codes. First, the proliferation of open-source compilers and well-specified intermediate representations (e.g., `LLVM`) allow now sophisticated static analysis of software. Second, there are significant advances in the machine learning field that could be applied to detect defects statically and dynamically in scientific codes early. Third, the broad adoption of continuous integration (CI) and continuous delivery (CD) brings opportunities to automatically apply such methods as applications evolve.

References

[1] Laguna, Ignacio, Dong H. Ahn, Bronis R. De Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi et al. "Debugging high-performance computing applications at massive scales." Communications of the ACM 58, no. 9 (2015): 72-81.

[2] Georgakoudis, Giorgis, Johannes Doerfert, Ignacio Laguna, and Thomas RW Scogland. "FAROS: A Framework to Analyze OpenMP Compilation Through Benchmarking and Compiler Optimization Analysis." In International Workshop on OpenMP, 2020.

[3] https://clangd.llvm.org/

[4] https://llvm.org/docs/Remarks.html

[5] Castro, Pablo De Oliveira, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. "CERE: LLVM-based codelet extractor and replayer for piecewise benchmarking and optimization." *ACM Transactions on Architecture and Code Optimization (TACO)* 12, no. 1 (2015): 1-24.

[6] Swain, Bradley, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. "OMPRACER: A scalable and precise static race detector for OpenMP programs." In *SC20*, 2020.

Reproducibility in the Era of Heterogeneous Computing: Problems and Solutions

Ignacio Laguna
Lawrence Livermore National Laboratory
Contact: ilaguna@llnl.gov
Keywords: reproducibility, accelerators, floating-point, numerical consistency, compilers

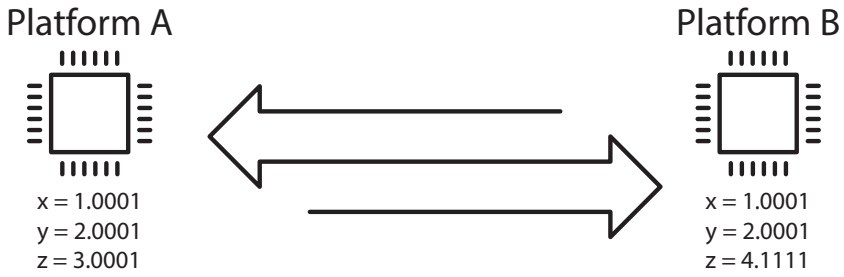
Introduction

DOE HPC systems are becoming highly heterogeneous, resulting in a variety of different computing platforms across DOE facilities. As a result, one of the biggest challenges for scientific computing programmers is **portability**, i.e., the ability to port scientific applications to different HPC platforms to execute scientific simulations efficiently and correctly in all of them. ***But what is portability without reproducibility?*** Even if a simulation runs faster in a new GPU system, we haven't achieved portability if we cannot obtain numerical consistency and ensure numerical reproducibility among systems. Thus, reproducibility is perhaps one of the biggest challenges that programmers face as DOE acquires a large variety of heterogeneous platforms.

Challenges

The state of the practice for reproducibility is to get "*partial reproducibility*", i.e., as long as the numerical results are correct within an error bound on multiple systems, we say that the results are reproducible in those platforms. It is known that bit-by-bit reproducibility is almost unattainable—most applications use floating-point (which introduces error at all layers) and limiting the asynchrony and concurrency of accelerators (which could provide high reproducibility) would hurt performance.

However, even when partial reproducibility has been the practice for many years, it is not uncommon to see numerical inconsistency and variability cases when applications are ported to new systems. For example, an interesting case is documented in [1] where the statistical consistency tests of the Community Earth System Model (CESM) failed when the code was ported to a new Argonne system—this case required months of investigation. Another example is documented in [2], where the energy calculations of a new application varied significantly when code was ported to a GPU system at LLNL. Some of these reproducibility nightmares are documented, but unfortunately, many are not.



Opportunities: What can we do about it?

While guaranteeing bit-by-bit reproducibility continues to be difficult to achieve (and sometimes it's unneeded), there is an opportunity to alleviate the reproducibility crisis in two ways: (1) provide **better tools** to help programmers identify the source of numerical inconsistencies; (2) document **best practices** to avoid such issues in the development of scientific software. In the rest of the paper, we elaborate on these two solutions.

Better Tools

Investments are needed to develop tools that help programmers identify the root cause of potential numerical reproducibility problems. Our research found that floating-point rounding error or concurrency are just the tip

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-829148).

of the iceberg and are not the most crucial sources of numerical inconsistencies. Various other factors play a significant role in this problem. These factors require significantly more research and tools:

1. **Compilers and optimizations**: Not all compilers are created equal—commercial compilers have a different philosophy than open-source compilers to generate code. Often optimization levels are not interchangeable (-O3 in `clang` doesn't mean the same than -O3 in `nvcc` or `xlc`), and they can alter the semantics of the code, which can introduce unwanted numerical answers. In addition, code running on the CPU is compiled with one compiler (e.g., `clang`/`gcc`), and code running on a GPU is compiled with a different compiler (e.g., `nvcc`). Tools to understand when compilers and optimizations induce reproducibility issues are needed. Tools like FLiT [3] and pLiner [2] are helpful, but much more research is needed to solve this problem.
2. **Hardware differences: CPU ≠ GPU**: While floating-point is a standard (IEEE 754), some accelerators implement only a portion of the standard or deviate from it. For example, NVIDIA GPUs do not provide mechanisms to detect exceptional quantities, such as infinity or NaN, and their floating-point units do not fully implement the standard. The FPChecker tool [4] is handy because it can detect such exceptional quantities and show the code line that originated it; however, more research is needed to provide tools for different classes of accelerators and to isolate problems earlier (e.g., via static analysis).
3. **Language semantics**: While floating-point is usually underspecified in programing languages, floating-point semantics can vary significantly among languages commonly used in scientific computing. For example, FORTRAN was designed with mathematical formulas in mind, but that was not the case for C and C++. In FORTRAN, for example, the compiler is allowed to transform A/B/C to A/(B*C), which is not the case for C/C++ (unless flags such as -ffast-math are used). Most programmers don't understand how these nuances affect numerical results. Static analysis tools can help programmers understand or control such language-related variations.

Learn and Document Best Practices

Many of the above numerical reproducibility issues (and others) can be mitigated if programmers follow best practices. Unfortunately, understanding all the aspects of the floating-point standard, compiler optimizations, and hardware implementations is too challenging. We can, however, take two actions. First, document and provide best practices of existing issues; this will help programmers fix many problems that are already known to occur when porting code. Second, new best practices and coding patterns must be learned and documented. Learning new best numerical coding practices is crucial as different classes of accelerators are adopted with varying levels of precision and perhaps more exotic domain-specific units. Advances in machine learning (ML) and artificial intelligence (AI) can help here.

Timeliness

Several new computer science trends suggest that solving the above challenges is possible today. Some of these trends include: (a) the proliferation of open-source compilers and standardized intermediate representations (e.g., LLVM), which allow a sophisticated analysis of software, including the behavior under different floating-point constraints, inputs, and environments; (b) advances in the ML and AI fields that could be applied to speed up learning potential sources of reproducibility issues and best coding patterns. Investing in solving the above problems would have a significant impact on DOE scientific applications running in future systems.

References

[1] Ahn, Dong H., Allison H. Baker, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dorit M. Hammerling, Ignacio Laguna, Gregory L. Lee, Daniel J. Milroy, and Mariana Vertenstein. "Keeping science on keel when software moves." Communications of the ACM 64, no. 2 (2021): 66-74.

[2] Guo, Hui, Ignacio Laguna, and Cindy Rubio-González. "pLiner: isolating lines of floating-point code for compiler-induced variability." In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-14. IEEE, 2020.

[3] Bentley, Michael, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. "Multi-level analysis of compiler-induced variability and performance tradeoffs." In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, pp. 61-72. 2019.

[4] Laguna, Ignacio. "FPChecker: a dynamic analysis tool to detect floating-point errors in HPC applications.", 202. https://fpchecker.org/

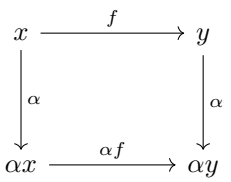
Toward a Science of Abstraction Design in Software

Kirk Landin (SNL) ktlandi@sandia.gov

Abstractions are the core building block of software. Good abstractions will allow software to quickly grow and evolve to meet the ever-changing needs of the research community. Bad abstractions will cause software to collapse under its own weight, as it becomes brittle, inflexible, and riddled with bugs. The overwhelming majority of literature on software quality focuses on improving the software engineering process and not the actual design of the software. Process is important in software engineering, but is only part of the picture. Poor engineering process can successfully be retooled at any point in a software lifecycle, superficial design issues such as non-uniform indentation can be fixed with simple text-processing tools, however software with a poor abstraction design is much harder to fix and may have to be re-built from the ground up to overcome its problems and limitations. Because of this, the structural design of the software and its abstractions is one of the **Most Important** facets of its construction, yet most of the software engineering community never mentions it as an issue. This needs to change!

What is an abstraction, Really?

What is an abstraction, really? Ask five different computational scientists and you will likely get five different answers, because abstraction has never been taught in a principled manner. An abstraction is a logical theory that accurately describes the structure of a computation to some level of precision. At its core, an abstraction is a *Commuting Diagram*, as shown here.



At the top of the figure is the *Concrete Domain*, with data values x, y and operation f. The bottom of the diagram contains the corresponding *Abstract Domain*, with data values, αx, αy, and operation αf. The *Abstraction Operation*, α, maps every concrete value/operation to some abstract one. It essentially "projects" away most of the unneeded details about the concrete domain, leaving a "distilled" interface that contains only the details necessary for the required abstract reasoning.

The abstraction operation, α, is *Sound* if this diagram commutes for all possible concrete values, x. This means that **All Possible Behavior** of the concrete implementation conforms to its abstract model. Soundness is an incredibly useful property when designing systems, especially ones that have many layers of abstraction.

The two foundational mathematical formalisms for specifying abstractions are *Functions* and *(Co)Algebraic Theories*. *Functions* (pure functions, in the mathematical sense) let us describe self-contained operations, and *Algebraic Theories* (along with their dual *CoAlgebraic Theories*) let us describe inter-dependent operations. When developing (Co)Algebraic Theories, one needs to specify behavioral invariants, which cannot be done in any commonly-used programming language. Because of this, developers should leverage modeling tools that enable them to specify/model/check invariants. *Alloy* is a very mature tool, developed at MIT, that allows very efficient specification and verification of abstractions and invariants. It is especially suited for structural modeling of software (Jackson 2012). *TLA+* is a suite of tools, also very mature, for modeling abstractions, and it is especially suited for modeling of state machines and state transition systems (Lamport 2002).

Abstraction Design as a Scientific Process

The general process of scientific inquiry is that one formulates a hypothesis and then runs experiments to test that hypothesis. If the experiments disprove the hypothesis, the hypothesis is modified to incorporate the additional knowledge gleaned from the experiments. This process is repeated until the hypothesis is sufficiently precise and general to adequately model the phenomenon of interest. This process is the same in Software Abstraction design. An initial abstraction is posed to model a component, "experiments" are run by developing both concrete implementations and client code for the abstraction. The hypothesis is then tested by asking the two questions, "Do the concrete implementations model the abstraction?" and "Does the abstraction provide the necessary information to the clients?" This process continues as new concrete implementations and clients of the abstraction are created. At any point in this process, one of the new components may disprove the hypothesis/break the abstraction. In this case a new, more correct, hypothesis/abstraction will be developed. At some point, the hypothesis/abstraction will become sufficiently useful and

the hypothesis testing loop will stop.

In addition to experimental data, abstraction designers leverage their theoretical understanding of software semantics to make informed decisions about what hypotheses/abstractions they should propose. They use this knowledge to craft elegant abstractions that have desirable properties (compositionality, soundness, simplicity, generality, etc.) Just as we expect a Physicist to adequately understand the underlying theory and propose hypotheses that are adequately informed by this understanding, we should expect our Computational Scientists to do the same thing. However, very few Computational Scientists have any knowledge about Programming Language Theory, Type Theory, Formal Semantics, etc. Because of this lack of knowledge, many design decisions are ad-hoc, can be very sub-optimal, and may be highly damaging to the overall software system.

Research Challenges

Leveraging the Full Power of Functional Abstractions: *Functional Programming* has been a buzzword for at least a decade now, and for good reason too, as functional abstractions are some of the simplest, most composeable, and easiest to verify abstractions in software. Our current software stacks woefully under-use functional abstractions and suffer a lot because of it. How should we re-architect scientific computations and libraries to maximize the use of functional abstraction? How do we decompose our computations in a way that is amenable to functional abstractions, and what should the functional abstractions look like for these computations? Which of our computations' correctness properties work well to encode in a language's type-system, are verifiable by standard type-inference algorithms, and make for a pleasant user experience? How do we effectively educate scientific software developers on the structure and use of these abstractions?

Designing Good Non-Functional Abstractions: In most code-bases there will be a non-trivial amount of functionality whose structure does not fit within the confines of functional abstractions. For these portions of common scientific computations, what appropriate (Co)Algebraic Theories describe the behavior at the correct level of precision, are elegant, easy to reason about, and are composeable with other abstractions? What sort of tooling and workflow will help our developers be efficient and productive in developing and testing non-functional abstractions?

Automatic Verification via Property-Based Testing: For correctness properties other than those which can be encoded into the type-system, we need an additional step to verify that concrete components actually model their abstractions. One of the most effective tools for doing this is *Randomized Property-Based Testing*, in which invariants are reified as unit tests that randomly generate large sets of inputs, testing that the given invariant holds for each input. If none of the test cases disprove the invariant, then there is good (although imperfect) assurance that the concrete implementation actually models its abstraction. How do we define our invariants so that they can generate effective property-based tests? How do we automate this test generation process as much as possible? How do we randomly generate test data for our different components in a way that ensures proper domain coverage by these randomized tests?

Concrete Steps to Move Forward

We need to start treating abstraction design as a real science, and not as some lesser, throwaway, activity as it has been treated in the past. We need to start using the scientific method to build abstractions. Everyone in development needs to understand the basic mathematical formalisms of abstraction and understand how syntactic mechanisms in mainstream languages, such as classes, inheritance, and generics fit into these formalisms. We need a culture that encourages developers to be precise about their abstractions when needed, actually specify invariants, and use modeling tools such as Alloy or TLA+. We need the Programming Languages community to start teaching the Scientific Computing community about the mathematical foundations of abstraction design. One of the most effective tools for teaching semantic reasoning to programmers is for them to study a programming language whose structure is closely tied to the semantic foundations of software. Haskell is the best language for this, because it forces programmers to wrestle with the semantic structure of their code much more than other languages. This causes a programmer to develop an intuitive understanding of semantic structure that can be applied to software systems written in any language. A year of working in Haskell yields huge benefits for a programmer's ability to design things in Python and C++.

References

Jackson, Daniel. 2012. *Software Abstractions: Logic, Language, and Analysis*. MIT press.

Lamport, Leslie. 2002. *Specifying Systems*. Vol. 388. Addison-Wesley Boston.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2021-14541 C

Community Building Blocks for Workflow Systems: The ExaWorks Experience

Dong H. Ahn[2], Kyle Chard[3,4*], James Corbett[2], Mihael Hategan[3,4], Shantenu Jha[5,1*], Daniel Laney[2*], Andre Merzky[5], Todd Munson[3], Rafael de Silva[6], Mikhail Titov[5], Matteo Turilli[1,5], Justin M. Wozniak[3]

[1] *Rutgers, the State University of New Jersey, Piscataway, NJ 08854, USA*
[2] *Lawrence Livermore National Laboratory, Livermore, CA 94550, USA*
[3] *Argonne National Laboratory, Lemont, IL 60439, USA*
[4] *The University of Chicago, Chicago, IL 60637, USA*
[5] *Brookhaven National Laboratory, Upton, NY 11973, USA*
[6] *Oak Ridge National Laboratory, TN, USA*
Contact Authors: chard@anl.gov, shantenu@bnl.gov, laney1@llnl.gov

I. Challenge

Workflows are increasingly necessary for scientific discovery, and represent a fast growing and important class of applications [1] that will use ExaScale systems. Workflow-based applications are diverse, increasing in number, and highly complex. This trend is driven by the coupling of traditional High Performance Computing (HPC) with new simulation, analysis, and data science approaches. Several Exascale Computing Project (ECP) [2] workflows exemplify this new reality [3], [4]: a heterogeneous combination of applications, Machine Learning (ML) models, and "glue" code, running on heterogeneous computers, orchestrated by a scalable workflow system.

These software combinations and integrations, however, are difficult to achieve due to challenges of coordination and deployment of heterogeneous software components on diverse and massive platforms. We observe that teams often develop ad hoc approaches which evolve into complex, hard to maintain software infrastructures. We are seeing an explosion in the number of workflow applications and engineering approaches to manage this new software complexity. Without intervention, a large number of custom workflow systems will be developed, and a great many domain-expert hours will be spent porting and developing duplicative capabilities across the community. Eventually, as project funding ends, these systems will fall into disuse as the next project re-creates their own capability.

Developing workflow systems that address the requirements of scale, performance and diversity is a grand challenge problem, and at least as challenging as developing the workflow applications they enable. Some challenges in building workflow systems include:

- What programming models and patterns are most suitable (productive, intuitive, performant) for workflow developers and application science?
- How do you build software that is performant and scalable yet is stable across a wide variety of environments?
- How do you find the balance between complexity (functionality) and performance?
- How can the workflows community move towards reuse rather than reinvention?
- What software best practices and methodologies can be developed for reducing duplication and re-implementation of systems?

An explosion in the number of independent solutions is making development and support of workflows increasingly unwieldy, expensive and unsustainable. Ultimately, we must find a way to manage the proliferation of tools and systems, and enable domain scientists to manage the complexity of new coupled, large & diverse workflow applications. This white paper is focused on how the computational science and workflows communities can develop software technologies and best practices to manage software complexity and reduce duplication and re-implementation of systems.

Several technical and non-technical challenges impede the creation of portable, repeatable, and performant workflows. On the technical side, workflow management systems (WMS) and complex workflows are difficult to port and maintain which hinders usability, portability and ultimately adoption. On the socio-technical side, myriad WMS exist which often try to provide complete and end-to-end capabilities, resulting in duplicated general capabilities but missing specialized functionality. The lack of coordination, high-quality specialized and broadly usable common components, has resulted in a disjoint workflows community that tends towards building ad hoc solutions rather than adopting and extending existing solutions. The complex workflows landscape demands an open community-based approach to address these challenges.

The ECP ExaWorks project, which we represent, was created in response to some of these challenges and is working to address some of these challenges, but there is a substantial need in the long term for focused effort on the problem of managing the software complexity that will surely come to the applications and science communities that are beyond the scope of ExaWorks' effort to build a software development kit of workflow components.

II. ExaWorks Experience

ExaWorks is leading a co-design process to create a workflow Software Development Toolkit (SDK) consisting of a wide range of workflow management tools that can be composed and inter-operate through common interfaces. The goal is to make SDK-supported tools easier to apply to complex science challenges by domain scientists. Although the ExaWorks design has well defined core, it is designed to be inclusive and extensible. The ExaWorks project is also working with the workflows community, large computing facilities, and HPC platform vendors to sustainably address the requirements of workflows at the exascale. The ExaWorks experience in developing community building blocks for workflow systems and developing workflow applications motivates the following socio-technical challenges:

- How can we develop community engagements develop methodologies for reducing system duplication and re-implementation, while promoting new ideas?
- How do application teams find and adopt workflow systems? How can workflow systems be structured to prevent "lock-in," accelerating adoption while allowing applications to use different systems for different problems and different users?
- What patterns and features in workflow systems are good at "staying out of the way," enabling discovery-driven science to advance most rapidly?

As hardware becomes more specialized, the disparity between the needs of scientific computing and the computing industry as a whole and specifically in workflows is likely to increase. It is also likely, however, that the increasing adoption of community-based AI methods and libraries will continue to connect scientific computing to the general community; thus, opportunities to appropriately integrate workflow tools with AI frameworks will become increasingly important.

III. Opportunity

More robust and reliable workflow systems can drive several aspects of changes in the scientific-computing community to improve software development practices. First, collaboration, compatibility, and information sharing could be enhanced among workflow systems to promote the best use of one or more systems for a given application. Second, well-tested software maintenance and deployment mechanisms are needed to ease the use of the workflow system features. Third, a robust feedback loop between scientific-computing developers and workflow researchers should be allowed to drive new features and rapidly address issues.

As the workflow environment becomes stronger, there are multiple opportunities to study and improve both the development of workflows and also workflow systems.

From the workflows perspective, there are inherently different usability challenges to consider, ranging from the way workflows are assembled (i.e., by scripting, adapting APIs, and integrating diverse components) through to the way they are executed in heterogeneous environments. Workflows research has focused primarily on performance and scalability, and relatively little focus has been given to studying composition, ease-of-use, user adoption, and ability to represent workflow patterns, for example.

From the workflow *systems* perspective, the ExaWorks SDK provides a starting point for consolidating the muddied technology landscape and studying the ability to package, test, and integrate components from different systems. However, addressing packaging challenges is only scraping the surface of the problem. A scientific approach should be taken to understand the combinations that can be applied as well as their interactions with the full spectrum of applications and physical environments. Modular, component-based approaches to software development have been successfully studied in many domains, and validation of the efficacy and benefits of this approach must be considered in the workflows space.

IV. Timeliness

Timeliness or maturity: Why now? What breakthrough or change makes progress possible now where it wasn't possible previously? What will be the impact of success?

Applications, hardware, and users. Increasing hardware heterogeneity, new computational workloads (e.g., ML/AI), and diverse user community necessitates new approaches to computation. Workflow systems provide a productive, robust, and scalable approach to democratizing computation.

Technology adoption. Widespread adoption of high level programming languages and a movement towards composing applications from disparate components (e.g., as driven by ML and workflow systems), combined with support for containerization and new packaging technologies, make it now easy to develop, share, and deploy complex workflows on different systems. This newly open-minded approach to a diverse range of programming modes provides a basis for these efforts.

Community readiness. The workflows community has long been segmented; however, recent efforts such as ExaWorks, WorkflowsRI, and ELIXIR have demonstrated the community's interest to collaboratively tackle the most challenging workflows problems.

Impact. The momentum of the present moment makes studying workflow programming a high value proposition. Better workflow programming practices will enhance the value of the workflow systems, the applications that run in them, and the underlying infrastructure itself.

References

[1] T. Ben-Nun, T. Gamblin, D. S. Hollman *et al.*, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 57–69.

[2] P. Messina, "The Exascale Computing Project," *Computing in Science Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

[3] L. Ward, G. Sivaraman, G. Pauloski *et al.*, "Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," in *Workshop on Machine Learning in HPC Environments*, 2021.

[4] H. Bhatia, F. D. Natale, J. Y. Moon *et al.*, "Generalizable coordination of large multiscale workflow: Challenges and learnings at scale," in *to Appear in the Proceedings of Supercomputing '19: The International Conference for High Performance Computing*, ser. SC '21.

Using the Science of Team Science for Better Science Software and Better Science

W. Christopher Lenhardt
Research Scientist,
Earth Data Science
Renaissance Computing Institute (RENCI) The University of North Carolina at Chapel Hill
clenhardt@renci.org

Stephen M. Fiore, Ph.D.
Pres., International Network for the Science of Team Science
Professor, Cognitive Sciences, Department of Philosophy
Director, Cognitive Sciences Laboratory, Institute for Simulation & Training
University of Central Florida
sfiore@ist.ucf.edu

Shannon McKeen ▪
Adjunct Faculty ▪
UNC's Kenan-Flagler Business School
Executive Director ▪ National Consortium for Data Science ▪
Deputy Director ▪ South Big Data Hub, RENCI, UNC-CH
mckeen@renci.org

In this paper we argue incorporating approaches, insights and lessons learned from the science of team science (SciTS), into the development and use of scientific software can contribute to qualitative improvements, and reciprocally these efforts can also be fertile grounds for SciTS studies. The Renaissance Computing Institute (RENCI) at the University of North Carolina, Chapel Hill is helping to create a science of team science activity as part of its NSF-funded South Big Data Hub (SBDH), which it co-leads with Georgia Tech. The goal of the SBDH SciTS track is to raise awareness about the relevance of team science to scientific research, to provide practical materials applying team science concepts for use in the context of research, and eventually to offer training or similar types of educational activities. This paper will provide an overview of team science, its relevance to science, and offer suggestions for applying team science to science software development.

Both the scientific enterprise and the development of scientific software are collective efforts. The myth of the lone scientist, or hero coder, are just that, memetic ways to represent activities that are complex and filled with tacit, expert knowledge. The elements of the scientific enterprise including, technology, tools, methods, and data, are increasingly complex as represented by examples of large scientific instruments and a range of other technologies and approaches (e.g., genomic technologies, nanotechnologies, and ubiquitous sensing). Additionally, across science there is recognition of the need to bring together disparate disciplines to address complex science questions as expressed by concepts such as transdisciplinary or convergent science. The digitalization of science technology and products adds yet another wide set of domains that need to be incorporated into the science enterprise in order to leverage these technologies in research. Funders and the National Academy have recognized these challenges and worked to identify the salient elements and identify ways to facilitate these types of activities for example recommending proposals with large teams include collaboration plans (*Convergence*, 2014; Lauer, 2018; National Research Council, 2015).

The complexities of pursuing research in these circumstances creates practical challenges such as developing shared understandings of project deliverables, defining decision-making roles across the project, and coordinating among geographically dispersed participants. This highlights the importance of foundational team concepts such as shared cognition, group dynamics, and coordination. Science software development is, itself, a complex interdisciplinary endeavor. It requires reconciling formalized software engineering processes, feature and design elicitation methodologies, IT and data security requirements with the overall research process. Tension emerges because, relative to software development, the scientific method is less structured and not as easily coordinated, even by adaptive methods such as agile development. Compounding this is the fact that most scientists are not familiar with software development methodologies and, therefore, lack the kind of common ground needed to coordinate diverse knowledge and expertise in an interdisciplinary team. Furthermore, in many cases, because of their limited understanding, they may be reluctant to shift limited project resources from research to other areas (Blanton & Lenhardt, 2014). Finally, addressing these

challenges is critical to the evolution of the scientific ecosystem given that science software is becoming an object of scholarly reuse similar to data. Funders see science software as an investment where improving the sustainability of the software increases the potential return and avoids duplication of effort. Sustainability in this context includes reusability and elements of reproducibility and replicability.

SciTS can help to address the aforementioned problems given that it focuses on knowledge building and synthesis of data and information from diverse team members. In particular, SciTS researchers study these complexities by drawing on a variety of social science domains that help us understand collaboration across domains. These domains come from the cognitive and computational sciences as well as the social and organizational sciences. Broadly termed, much of the SciTS field is ideal for examining complex sociotechnical systems such as science software development. Bringing the array of methods used by SciTS, such as ethnographic analysis, structured interviews, and questionnaires, to analyze science software development, could yield new ideas for improvements. Even though the SciTS field is relatively young, SciTS findings provide evidence of the potential to address science software development challenges. For example, SciTS research identifies a distinction within the work that a team does between tasks that relate to helping the team to work effectively from project tasks that directly further project goals. Team science has also highlighted the need to incorporate ways to facilitate shared understanding across diverse team members in order to facilitate moving a project forward (Fiore & Wiltshire, 2016; Hall et al., 2018). The SciTS also examines elements of teams such as decision-making, team dynamics and interpersonal relations and to understand how those elements influence outcomes. SciTS also works to understand how to translate SciTS findings into better science and for improving team function (Stokols et al., 2008).

This paper has provided an overview of the elements and challenges presented by the team aspects of scientific software development and the potential to leverage insights from SciTS. The potential relevance of SciTS insights to developing science software, could contribute to more efficient software development, software that more closely aligns with requirements, potentially improve reusability, and science outcomes. Reciprocally, SciTS researchers can benefit from the opportunity to study these types of scientific efforts. To this end the SBDH track in this area may become a resource on which to draw in future efforts.

References

Blanton, B., & Lenhardt, C. (2014). A Scientist's Perspective on Sustainable Scientific Software. *Journal of Open Research Software*, *2*(1). http://openresearchsoftware.metajnl.com/article/view/jors.ba

Convergence: Facilitating Transdisciplinary Integration of Life Sciences, Physical Sciences, Engineering, and Beyond. (2014). National Academies Press. https://doi.org/10.17226/18722

Fiore, S. M., & Wiltshire, T. J. (2016). Technology as Teammate: Examining the Role of External Cognition in Support of Team Cognitive Processes. *Frontiers in Psychology*, *7*. https://doi.org/10.3389/fpsyg.2016.01531

Hall, K. L., Vogel, A. L., Huang, G. C., Serrano, K. J., Rice, E. L., Tsakraklides, S. P., & Fiore, S. M. (2018). The science of team science: A review of the empirical evidence and research gaps on collaboration in science. *American Psychologist*, *73*(4), 532–548. https://doi.org/10.1037/amp0000319

Lauer, M. (2018, April 4). Impact of Teams Receiving NIH Funding. *NIH Extramural Nexus*. https://nexus.od.nih.gov/all/2018/04/04/theres-no-i-in-team-assessing-impact-of-teams-receiving-nih-funding/

National Research Council. (2015). *Enhancing the Effectiveness of Team Science*. The National Academies Press. https://doi.org/10.17226/19007

Stokols, D., Hall, K. L., Taylor, B. K., & Moser, R. P. (2008). The Science of Team Science. *American Journal of Preventive Medicine*, *35*(2), S77–S89. https://doi.org/10.1016/j.amepre.2008.05.002

Transformative Profiles in Research Software Teams: DEI for Collaborative Innovation

Authors: Mary Ann Leung, Sustainable Horizons Institute, and Connie L. McNeely, George Mason University

Computational science and engineering (CSE) was established as one of the pillars of scientific discovery many decades ago and the field has seen a great deal of growth since its inception. The coupling of high-performance computing with CSE has led to even more growth and the field has become an increasingly important paradigm for advancing scientific knowledge and contributing to the economy. Research software sophistication and complexity have followed this growth leading to larger and larger multidisciplinary teams. While formal software engineering methods are well studied and documented in other disciplines and especially in the private sector, related formal practices have not been widely used in research software development.

Challenge

According to a Department of Energy (DOE) Advanced Scientific Advisory Committee Workforce Subcommittee Letter in 2014 (1) and more recently in a report on four decades of DOE Advanced Scientific Computing research (2), DOE laboratories are facing workforce challenges, including increasing competition for qualified workers and the need to develop more effective techniques to attract and sustain a full spectrum of relevant talent. The letter also calls for new approaches to recruiting and retaining computing sciences staff, especially women and minorities. While research indicates that social diversity results in greater innovation (3), research software itself, as a growing multidisciplinary field spanning the globe and representing a diverse spectrum of technical domains, remains largely homogeneous.

Opportunity

A recent article (4) posits employing the scientific method to understand and advance how research software is developed — i.e., a science of research software. We suggest a similar approach to the study of the social aspects of research software development through the lens of diversity, equity, and inclusion (DEI), applying a science of broadening participation (SoBP) approach (5) (6)to research software development. One might expect research software communities to welcome social diversity, given the technical breadth of the field. An SoBP approach could reveal various aspects of research software as a field that might facilitate or inhibit broader participation from underrepresented groups,[1] document the current status of those groups, and identify successful strategies and practices, as well as common inhibitors or detractors. Also, SoBP offers a systemic perspective on diversity as both an innovation and opportunity for transforming the profile and capacity of research software teams (7), raising a variety

[1] Including Black or African American, Hispanic/Latinx, American Indian, Alaska Native, Native Hawaiian, Pacific Islanders, women, persons with disabilities, first generation scholars, and students from smaller colleges and universities.

of questions for further exploration. For example, what aspects of the research software enterprise impact DEI, and vice versa? To what extent does the inherent technical diversity and the evolution of the field impact DEI strategies? What can be learned from analyses (e.g., longitudinal or structural) of efforts to diversify the research software workforce and related impacts? To what extent are research software communities open to members of underrepresented groups, and how might a broader and inclusive collaborative base be developed to encourage and support innovation in research software?

Timeliness

Recent events have highlighted longstanding disparities resulting in increased awareness, renewed interest in DEI, larger and larger numbers of allies. Through profiles of research software teams, their formation and make up, growth and attrition, and interactions the science of research software could identify how aspects of diversity, equity, and inclusion can facilitate transformative capacities and collaborative innovation. Applying the science of broadening participation for a more in-depth understanding of research software communities could illuminate pathways to related advances and more effective scientific discovery.

References

1. **Chapman, Barbara, et al.** *DOE Advanced Scientific Advisory Committee (ASCAC): Workforce Subcommittee Letter.* United States : s.n., 2014. Web. doi:10.2172/1222711.

2. **Hendrickson, Bruce, et al.** *ASCR@40: Highlights and Impacts of ASCR's Programs.* s.l. : ASCAC Subcommittee on the 40-year history of ASCR for the, 2020.

3. **Gompers, Paul and Kovvali, Silpa.** The Other Diversity Dividend. *Harvard Business Review.* July-August 2018.

4. **Heroux, Mike.** Research Software Science: A Scientific Approach to Understanding and Improving How We Develop and Use Software for Research. *Better Scientific Software.* [Online] Sept 25, 2019. [Cited: Nov 22, 2021.] https://bssw.io/blog_posts/research-software-science-a-scientific-approach-to-understanding-and-improving-how-we-develop-and-use-software-for-research?utm_medium=email&utm_source=govdelivery.

5. *Moving the Needle, Raising Consciousness: The Scienec and Practice of Broadening Participation.* **McNeely, Connie L. and Husbands Fealing, Kay.** 5, s.l. : American Behavioral Scientist, 2018, Vol. 62.

6. *Developing Sustainable Methods for Broadening Participation: Transforming Mainstream Science and Technologies through Normalization of Inclusion.* **Leung, Mary Ann.** 5, s.l. : American Behavioral Scientist, 2018, Vol. 62.

7. **McNeely, Connie L.** Diversity as Innovation and Opportunity: Transformative Affect in S&T Workforce Participation. [book auth.] U Hilbert. *Diversities of Innovation.* London : Routledge, 2019.

Automated, Extensible Correctness Diagnostics for Scientific Computing

Chunhua Liao[1] and Peter Pirkelbauer

Lawrence Livermore National Laboratory

Challenge

Developing scientific simulation codes involve multiple iterations of designing, implementation and refactoring in order to exploit everchanging hardware platforms to answer bigger scientific questions. Any changes to the simulation codes may potentially introduce bugs. Currently, developers often manually insert ad-hoc diagnostic statements into an application to log relevant variable values, at critical steps within an execution. The resulting log information is then used for manual investigation to ensure that the simulation runs as expected. Obviously, this common practice is time-consuming, non reusable, and error-prone.

Opportunity

There is an opportunity to develop extensible and automated techniques to support universal correctness diagnostic capability suitable for reuse across many simulation codes. Ideally, a computational scientist would be able to identify a set of key properties of interests and use standard language constructs, source code annotations or directives to express diagnostic semantics. The HPC programming systems, including compilers and runtime systems, would then take over and automatically implement the corresponding code instrumentation, data logging, and even problem diagnosing and reporting.

In order to achieve this vision, we need to invest in the following research areas:
1. Systematically collecting application semantics which are commonly used by developers to support correctness diagnostics of simulation codes. Example diagnostic semantics can be as simple as which stage a program is currently executing or a range of allowed values for a given variable. Others can be highly domain-specific information such as the number of segments executed in a particle transport program, or convergence rates of iterative methods.

2. Initiating development of standard ways to express such diagnostic semantics and communicate them to a compiler/runtime system, which in turn can automatically instrument code at the right time and trigger the right checking for the relevant properties. Directive-based programming models, such as OpenMP, could be an ideal vehicle to experiment with new annotations for such purposes.

3. It may take years to have matured language and compiler support for automated correctness diagnostics. Library-based approaches should be investigated as early as possible so

[1] Corresponding author: liao6@llnl.gov This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-ABS-829256

experienced developers can leverage libraries to improve diagnostic support. The libraries often form the basis for runtime library support of compilers. There are existing libraries for memory access patterns (e.g. XPlacer[1]) and performance introspection (e.g. Caliper[2]) of HPC software. They could be extended to support correctness diagnostics.

4. Given the diverse and rapid-changing scientific computing needs, it is impossible to anticipate all semantics and properties which are needed in the field. We need to study easy interfaces for users to add customized extensions. Callback functions and user-defined operations (similar to user-defined reduction in OpenMP) may be possible approaches to realize such extensions.

5. Side effects and overhead are major concerns of any code instrumentation, which may disrupt original performance characteristics, impede compiler optimization, and increase time and space complexity of applications. Further research is needed to minimize such negative impact.

In the end, wide collaboration among domain scientists, applied mathematicians, computer scientists, and software engineers is needed to reach the goal of extensible and automated correctness diagnostics of scientific applications.

Maturity

Compiler-based code instrumentation is a mature technique to help inspect behaviors of programs. The resulting information can be used for correctness checking (including data race detection), memory access pattern analysis, and so on. On the other hand, directive-based programming models, such as OpenMP and OpenACC, are commonly used to standardize annotations related to parallelism. Prior studies also successfully explored other types of semantics, including fault tolerance[3]. It is a promising new research direction to incorporate semantics related to correctness diagnostics so users can leverage compilers and runtime to automatically conduct in-depth and standard correctness investigation of scientific computing.

References

1. Peter Pirkelbauer, Pei-Hung Lin, Tristan Vanderbruggen and Chunhua Liao, XPlacer: Automatic Analysis of CPU/GPU Access Patterns, 34th IEEE International Parallel & Distributed Processing Symposium, May 18-22, 2020.
2. David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In Supercomputing 2016 (SC16), Salt Lake City, Utah, November 13-18, 2016.
3. Jacob Lidman, Daniel J. Quinlan, Chunhua Liao and Sally A. McKee, ROSE::FTTransform A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research, Fault-Tolerance for HPC at Extreme Scale (FTXS 2012), Boston, June 25-28, 2012.

Reinventing the Ecosystem to Improve HPC Software

Chunhua Liao[1]

Lawrence Livermore National Laboratory

Challenge

High Performance Computing is becoming increasingly difficult for users to use due to the ever-growing complexity of heterogeneous hardware and software components. Users often have to spend tremendous time on each step of HPC workflows, including learning how to use a system, adapting their codes to novel hardware features, debugging the execution, conducting performance analysis, applying various optimizations, and so on. This process typically needs to repeat multiple times before users can sufficiently utilize the supercomputers to obtain satisfactory results.

On the other hand, there are numerous software packages being developed to help users in various steps of their workflows to make the process more productive. However, the very existence of such software packages (including tools for performance, modernization and correctness) is not easily known to users. Many promising software packages have not received timely support to be robust and sustainable. This trend has to be reversed or only few users would enjoy the HPC systems.

Opportunity

There is an opportunity to reinvent the HPC ecosystem to have HPC of the users, by the users , and for the users. A well-designed ecosystem would speedup both hardware advances and software development based on timely user feedback. In particular, the HPC community should benefit from research and development of the following elements in the ecosystem:

1. Holistic metrics of success: The mainstream success metrics of HPC are machine-oriented, focusing on machine performance and efficiency. Human factors are not officially considered when ranking the best supercomputers as of today. A set of new holistic success metrics is needed to consider the total time, cost, and quality of results when evaluating HPC systems. The total time should include not just machine cycles, but also human cycles spent on training, thinking steps, number of keystrokes, mouse clicks or even cursor travel distances. With the right metrics, we can drive the HPC software development in the right directions.
2. A centralized marketplace. Currently, HPC Software packages are scattered all over the place, released through ad-hoc channels and formats. It is challenging for users to find the right software tools for their problems. The best practice of providing and finding software is through a common software marketplace. Prominent examples include Apple‘s app store and Google Play store. The HPC community needs to set up a similar marketplace for HPC software packages so providers can register software information while users can easily

[1] Contact: liao6@llnl.gov Prepared under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-ABS-829260

search for what they want. Such a marketplace can also accept user reviews to give feedback to providers while facilitating communications among users.

3. A user-directed rewarding system. The reality of software development is that financial support is essential. Traditional funding systems use a top-down approach with many benefits. User-directed, bottom-up funding systems may provide additional support for software developers who are overlooked by higher management in typical funding cycles. The user-directed rewarding system can be implemented as peer-to-peer bonuses. Early experiments could use virtual currency in the community. People could earn virtual currency if they develop good software tools for the community. Or they could send earned virtual currency to other providers if they are happy with the software provided. This rewarding system can be made part of the software marketplace.
4. Standard benchmarks, competitions and state-of-the-art dashboards. If you can't measure it, you can't improve it. Performance-oriented benchmarks have been instrumental in HPC to drive the advance of supercomputers. We need similar investment in correctness and productivity benchmarks that can help reduce time-to-solution. Such benchmarks can be used to generate a dashboard comparing the state-of-the-art tools for a given problem, fostering fair competition and facilitating tool selections.
5. Software tools as composable services. Users often find that no single software tool is perfect in every aspect. Assembling multiple tools is a natural solution. Traditionally, tool researchers and developers are incentivized to focus on their own individual tools, without too much desire to work with other tools. To foster software tool development as composable microservices, we should invest in better interoperability of software tools, by promoting common APIs and result exchange formats for each type of software tools.

Why Now

“The best time to plant a tree was 20 years ago. The second best time is now.” We expect that most people would agree with the ideas of holistic success metrics, a common software marketplace, and complementary user-directed rewarding systems. There are also growing interests in benchmark development focusing on other than performance in the community. DataRaceBench[1] is an example benchmark focusing on correctness of data race detection tools, with a dashboard showing apple-to-apple comparison of existing tools to facilitate tool selections. A recent study[2] also shows that composing multiple data race detection tools can deliver better results, once common APIs and data exchange formats are defined.

References

1. Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan and Ian Karlin, DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools, The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, Nov. 12-17, 2017
2. Yaying Shi, Anjia Wang, Yonghong Yan, Chunhua Liao, RDS: A Cloud-Based Metaservice for Detecting Data Races in Parallel Programs, 14th IEEE/ACM International Conference on Utility and Cloud Computing, University of Leicester, Leicester, UK, December 6-9, 2021

Bridging Software Developers with Dynamic Program Analysis in IDE

Xu Liu
Department of Computer Science
North Carolina State University
Oak Ridge National Laboratory
xliu88@ncsu.edu

1. Challenge

Dynamic program analysis (also known as profiling) is well-known for its powerful capabilities of identifying performance inefficiencies in modern software packages. Although a large number of dynamic program analysis techniques are developed in academia and industry, very few of them are widely used by scientific software developers. The main reason is that mastering various tools to interpret their analysis results requires substantial efforts; moreover, many tools have their own design of graphic user interfaces (GUI) for data presentation, which deepens the learning curves.

The DoE supported profilers, such as HPCToolkit and TAU spend substantial efforts to organize tutorials to flatten the learning curve. However, there are still various challenges for scientific application developers to intensively use these tools in their daily software developing activities. **Challenge I**: profilers are isolated from each other; software developers typically use one profiler they are familiar with consistently, which limits the insights they can obtain from other profilers with different capabilities. **Challenge II**: these profilers are disjoint from the coding environments such as IDEs or editors; frequently switching focus between profilers and IDEs/ editors significantly complicates the entire cycle of software development. **Challenge III:** most tools focus on data collection instead of the analysis. Comparing analysis results and deriving new metrics across different executions or even different profilers are not straightforward.

Thus, we raise a research question here: how to better integrate existing profilers into the daily development activities for scientific software developers.

2. Opportunity

To achieve this goal, we foresee three opportunities to address the aforementioned challenges.

2.1. Generic Data Format for HPC Profilers

With the generic format, we can combine the analysis from different profilers and perform insightful analysis. Google's Protocol Buffer [1] can express new data format, which can automatically generate a set of low-level APIs of different language bindings to produce the data in the format. Designing a generic format that can support common features of mainstream HPC profilers (e.g., HPCToolkit [3], TAU [4], ScoreP [4], Caliper [5]), which we identify as (1) p*rofiling contexts:* functions, loops, basic blocks, or individual instructions of interest, (2) m*etrics:* cycles, memory consumption, cache misses, lock contention, and many others, (3) c*all paths:* a series of frames on the call stack at any given profiling contexts, and (4) source c*ode mapping:* associating analysis results with programs at the binary or source code levels. Moreover, the format can be extended to support unique features from individual profilers, such as various tracing data.

The generic data format can unify different profilers to address the Challenge I because the generic data format enables the possibility of breaking the abstraction wall among different profilers, which can help the adoptions of various tools.

2.2. Profiling Visualization in IDEs/Editors

Microsoft's Visual Studio Code (VSCode) [2] is a text editor with powerful IDE-like features. As VSCode becomes more and more popular for scientific developers, integrating the profiling visualization in VSCode can help address the Challenge II.

Using VSCode plugin, we can visualize profiles produced by existing profilers in either flame graphs or tree tables. All the views provide three variances with inclusive and exclusive metrics: top-down view, bottom-up view, and flat view, which are similar to the views in HPCToolkit. As an advanced feature, one can click on any context to associate it with source code and can directly modify the source code according to the profiles with no need to switch back and forth between profilers and IDEs.

2.3. Data Analysis Interfaces

Most tools focus on data collection and attribution with little support for analysis with existing data mining and machine learning techniques. Thus, software developers need significant expertise to understand performance issues and take appropriate actions for optimization. The generic data format and integration with VSCode open a new direction to let software developers analyze performance data in an easy way. With a simple interface, one can analyze one or multiple profiles from one or multiple profilers in one or multiple runs, such as merging or differentiating profiles, deriving metrics, and correlating contexts involved in performance inefficiencies. It is also possible to integrate some ML algorithms to automatically pinpoint performance issues and provide optimization suggestions. This can address the Challenge III.

3. Timeliness or Maturity

This is a timely research because as the DoE supported profilers become mature in data collection and attribution, easing their usage will become more important. Moreover, the stable infrastructures such as VSCode and Protocol Buffer can make this research feasible. Another challenge in the research is to handle a large amount of data produced by the profilers and process them in a reasonable time in the front-end GUI. The evolution of JavaScript and WebAssembly today with huge performance improvement makes the data analysis in the visualization interface possible. If this research succeeds, the software developers will actively use existing profilers in their daily development activities.

References

1. Google Inc., Protocol Buffers, "https://developers.google.com/protocol-buffers".
2. Microsoft Inc., Visual Studio Code, "https://code.visualstudio.com".
3. Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.
4. S. Shende and A. D. Malony. TAU: The TAU Parallel Performance System, International Journal of High Performance Computing Applications, Volume 20 Number 2 Summer 2006. Pages 287-311.
5. Score-P, "https://www.vi-hps.org/projects/score-p/"
6. David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *Supercomputing 2016 (SC16)*, Salt Lake City, Utah, November 13-18, 2016.

Enhancing Software Performance Portability and Reproducibility: Perspectives from Automatic Performance Tuning

Younghyun Cho * James W. Demmel * Xiaoye S. Li † Yang Liu † Hengrui Luo †

1 Introduction

The Exascale Computing Project (ECP) community has been investing a lot of effort into the development of High-Performance Computing (HPC) software to meet the exascale performance requirements. ECP software packages such as SuperLU_DIST [5], SLATE [4], and many others, typically contain various tunable parameters that are decisive in the performance. Automatic performance tuning plays an important role to achieve performance portability, but it still remains a challenge. First, the optimal parameter values vary from platform to platform. Second, state-of-the-art autotuners [8, 6, 7] still require many performance data samples to train the surrogate performance model. The high cost of autotuning makes it difficult to achieve performance portability of HPC software frameworks. In this context, research communities started to consider sharing performance data samples collected from various software and hardware platforms among different users [2, 1]. However, conventional HPC software frameworks do not have sufficient reproducibility, making it difficult to reuse the collected performance data samples obtained from different platforms (e.g., transfer learning). In what follows, we identify the challenges in performance portability and reproducibility, and discuss future directions for HPC software development, with a focus on reducing the costs of performance tuning.

2 Enhancing Performance Portability and Reproducibility

2.1 Performance Portability

Challenges. Software and hardware platforms continue to evolve as new technologies are invented. Consequently, software developers need to constantly maintain their software packages to provide the best possible performance across different hardware platforms. As mentioned in Section 1, HPC applications rely on many tuning parameters and have a large tuning search space. To attain the best performance across different platforms, the current practice is to use autotuning techniques [8, 6, 7]. However, since the software performance needs to be tuned for every new platform, software developers or users need to run an expensive autotuning task for each platform. Another difficulty is estimating the performance bound of an application on a new hardware architecture, which can be used to understand if the performance is sufficiently optimized. Although there are existing techniques such as analytical roofline analysis [11], estimating performance bounds is becoming more difficult on upcoming hardware architectures which have non-uniform memory accesses and special hardware intrinsics.

Opportunities and suggestions. Performance tuning and optimization is an expensive task, especially for HPC applications. To achieve performance portability, transfer learning using historical performance data (trained on different hardware) is a promising approach [8, 1]. We argue that transfer learning for performance tuning would be the key to achieving software performance portability in the exascale era. Transfer learning has originally been researched in to train machine learning (ML) models efficiently [9]. There is not much work demonstrating the validity of historical data for performance tuning, hence it requires continued research from the community for transfer learning-based performance tuning. More specifically, to determine the reusability of available performance data, transfer learning will require new advanced techniques to estimate or quantify performance variability across different platforms or upon some code changes. It is also important to model and evaluate performance bounds on new hardware architectures. These efforts should be considered a future research direction for HPC.

2.2 Performance Reproducibility

Challenges. Conventional software packaging does not provide sufficient information to reproduce the same performance or result. Software packages are becoming more complex and require an increasing number of other dependent software packages. For reproducibility, therefore, the dependent software packages and their versions need to be recorded properly. However, conventional software packages usually require manual installation steps

*University of California at Berkeley (younghyun@berkeley.edu, demmel@cs.berkeley.edu)
†Lawrence Berkeley National Laboratory (xsli@lbl.gov, liuyangzhuan@lbl.gov, hrluo@lbl.gov)

and are distributed in various ways (e.g., Github, Bitbucket, specific websites, etc.) with no common agreement of version control. Manually tracking such reproducibility information is becoming infeasible. In addition, today's resource managers often do not permit collecting the necessary machine information that affects software performance. Hence, it is also necessary to monitor and record the machine environment information where the code is evaluated (e.g., allocated core and node resources, interference with other shared users). Although some workload managers (e.g., Slurm) permit collecting the information about the allocated nodes and their processor/memory architectures, monitoring or understanding interference of shared resources (e.g., shared disks) is still a challenge.

Opportunities and suggestions. For reproducibility, software packages should use workflow management and/or automatic installation tools without relying on manual installation. There are recent community efforts to provide a portable and reproducible software environment, such as CK [2] and Spack [3]. CK is a tool for reproducible and automated workflows and has successfully been used in many research artifacts in artifact evaluation processes of some computer science conferences (e.g., SC, PLDI, PPoPP, etc.). Users can write a meta-description of their workflow using CK's syntax and use a command line interface for automatically installing and running the workflow. Spack is an automatic package manager for HPC. Spack already supports many software packages (about 6,000 packages) and can track software dependency information. If a software package supports these tools, users can easily parse the software dependencies and provide the necessary information to reproduce the performance. xSDK [12] is a good example of using Spack. xSDK is a collection of related and complementary ECP software packages that support automatic installation using Spack [3] along with an autotuner called GPTune [10, 6]. Applying such an automation tool to complex software frameworks requires lots of efforts by software developers to consider the evolving software and hardware platforms. Therefore, continuous investment is required for the sustainability of the approach. In addition, to further enhance the software performance reproducibility, resource and workload managers can provide more machine statistics information (e.g., load of shared resources).

3 Conclusion

As we approach the exascale era, performance data collected on large-scale machines is an expensive and valuable resource, and it is gaining importance to collect and share performance data between different users. Building surrogate performance models using shared performance data (i.e., transfer learning-based autotuning) becomes imminently important but have not yet been considered widely due to the difficulty to track and maintain the complex software and hardware dependency of performance data. We observe that recent community efforts for reproducible and portable software environments such as CK [2], Spack [3], and xSDK [12], with a support of an autotuner database [1] can help maintain a tractable and easy-to-use ECP software infrastructure with performance portability and reproducibility. In conclusion, we highly suggest the community to consider using these tools in software development and provide an easier interface to use autotuning capabilities such as transfer learning.

References

[1] Y. Cho, J. W. Demmel, X. S. Li, Y. Liu, and H. Luo. Enhancing Autotuning Capability with a History Database. *The IEEE 14th International Symposium on Embedded Multicore/Manycore SoCs (MCSoC-2021)*, 2021.

[2] G. Fursin. Collective Knowledge: organizing research projects as a database of reusable components and portable workflows with common APIs. *arXiv preprint arXiv:2011.01149*, 2020.

[3] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[4] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[5] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

[6] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 234–246, New York, NY, USA, 2021. Association for Computing Machinery.

[7] H. Luo, J. W. Demmel, Y. Cho, X. S. Li, and Y. Liu. Non-smooth Bayesian optimization in tuning problems. *arXiv preprint arXiv:2109.07563*, 2021.

[8] H. Menon, A. Bhatele, and T. Gamblin. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 831–840, 2020.

[9] S. J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[10] W. M. Sid-Lakhdar, Y. Cho, J. W. Demmel, H. Luo, X. S. Li, Y. Liu, and O. Marques. GPTune User Guide. `https://gptune.lbl.gov/documentation/gptune-user-guide`, 2021.

[11] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[12] xSDK4ECP. xSDK: Extreme-scale Scientific Software Development Kit. `https://xsdk.info`, 2021.

The Opportunity and Challenge of Serverless Computing for Large Scale Computational Science

Sandia National Laboratories, Jay Lofstead (gflofst@sandia.gov), Matthew Glickman (mrglick@sandia.gov)

Challenge: Large scale science simulations, such as the Community Earth Systems Model (CESM) and Consortium for Advanced Simulation of Lightweight Reactors (CASL), are composed of independent subsystems coupled to offer a better computational view of the physics being studied. CESM's development was coordinated to provide for the execution of independent sub-models as well as the composite whole. The various third-party libraries were kept largely consistent to ensure compatibility and enable easier builds. Despite these efforts, the system still has to deal with physical and time scale differences when coupling making the composite application more complex.

CASL is a more loosely coupled system that has relied on advanced tools to coordinate communication between components. Notoriously, CASL had a period, if not a current state, where different subsystems each relied on a different, incompatible versions of the same third-party libraries making assembling the complete application exceedingly difficult.

Both of these examples are far from isolated. When Sandia's CTH shock physics simulation was used for an in-memory data coupling project, it took two months of two failed builds per day (each taking about 8 hours to complete) before the discovery that the documentation was definitely wrong and that a special configuration known only to the CTH developers was necessary to realize a successful build.

All of these examples demonstrate the need for better means of resolving the incompatibilities and other challenges that arise when composing large-scale, coupled applications.

While the need exists for building and deploying these complex, composite applications, how to achieve that goal is still an unsolved problem. Existing approaches are ad hoc and try to use existing, more traditional infrastructure. For example, CESM is designed to use MPI multi-applications with a coupling component to integrate the various components. This integrator is responsible for understanding how to do time and space conversions and how to route data to the appropriate paired component(s) is hard coded. Using the coupler for a different composite application is not possible.

For CASL, the incompatible library versions forced develoeprs to decompose the application and use tools to provide for communication between independent parallel applications. In the traditional HPC space, this kind of communication is not a natural function. For example, MPI Intercommunicators are difficult to use and require intricate configurations that are platform specific to set them up (e.g., how the nodes are named and how to connect to them depends on the interconnect).

While deploying multiple applications using a single job allocation using MPI is straightforward with the standard multi-application run as is done for CESM, the partitioning is fixed. The composite would have to exit to re-allocate in a different configuration.

Opportunity: A recent trend in the enterprise and Internet space is serverless or function-as-a-service architectures. These closely related ideas boil down to decomposing an application into small, independent parts with well defined interfaces enabling independent scaling and software versioning for each component. Initial HPC experiments for serverless have been mixed [1]. But the idea may offer two advantages.

First, sharing data using computational fabrics like CXL [3] (more broadly supported than the HPE-led Gen-Z) are reducing the performance barrier to distributing components and still achieving high memory access performance. The idea of distributing memory on a very fast fabric enables coupling applications via direct memory sharing with considerably less performance loss. It still does not address the space and time mismatch problems though.

Second, with service orchestration systems like Kubernetes and OpenStack achieving a dominant market position, extensive support for this computational model is available with continuous improvements and maintenance. The compute model used in this environment is much more service than batch job oriented. The goal is to deploy and scale services according to demand rather than making a fixed partition allocated to a single user for a period of time. Some services may be common and shared among all machine users. The advantage to this approach is modular construction enabling a mix-and-match construction using well defined interfaces and proven implementations.

With standard math systems like Trilinos and PetSc offering vast computational infrastructure on which the physics is programmed, these systems could be offered as a collection of services reducing the code load for an application. In essence, truly shared libraries can be deployed with multiple applications all using the same services, but scaling the number of service processes on an on-demand basis. With the strict

interfaces already provided by these systems, new versions can be deployed as needed or slowly rolled out by reconfiguring what version of the service a component will use. Note that this would directly address the problems CASL encountered with incompatible different third-party library versions because each component would be deployed with its own manifest of what it depended on allowing the system to partition and spin up different service versions, as needed.

Adapting computational science applications into this kind of serverless model deployed on an orchestration framework like OpenStack or Kubernetes may enable wider independent development of different subsystems while maintaining nearly the same performance as the fragile, tightly coupled composite application structures used today. The challenges that have already been identified show that while the serverless infrastructure is mature, we do not understand how to deploy science applications using it to achieve the advantages we need for the future software generations.

The research opportunities that can match the functionality gains are numerous. For example, coupling components requires a coupling engine that has typically been a custom effort. Work like SuperGlue [4] and SmartBlock [2] demonstrate the potential for assembling conversion routines via confiuguration rather than writing code. This approach could enable this approach to be more feasible.

Another example of a research opportunity is simply performance engineering. Currently, the orchestration services have non-trivial overheads exceeding those of traditional batch scheduling systems. Part of this overhead is due to the ability to delay spinning up services that may not be needed or were underspecified shortly after they are provisioned. Other factors include deploying components to various compute nodes and the data access and later persistence operations. Experiments to determine overhead sources and how to mitigate them are ongoing in industry as it affects the financial returns on the infrastructure. Ensuring these developments meet the needs of HPC applications will be left to the science community.

Timeliness or Maturity: This idea is both extremely timely and the maturity outside of the computational science space is high. Specifically, as composite applications continue to increase complexity with increasing scale due to incorporating new physics as well as in situ analytics, finding a way to enable the complexity to grow while reducing the fragility and inter-component conflicts will enable higher quality science continuing to take advantage of the extreme scales new platforms are achieving.

The maturity of the serverless model and orchestration frameworks is demonstrated through all of the Internet scale services enjoyed by the planet today. These applications are composed of small, independently deployed and scaled components that work in concert to achieve some function. The pervasive and diverse use of these platforms offers a vibrant open source community to support the underlying software infrastructure while the computational science community can solely focus on the science. The analogy is similar to the relationship with MPI today. MPI developers ensure a solid, performant foundation exists and the computational scientists use this foundation to enable their science. Through the community embracing research focused on how to enable these environments more fully for science applications, we will be able to leverage the extensive for profit efforts to support non-profit science and engineering efforts with greater fidelity and scale than ever before.

While the architecture does not address all of the issues with forming composite application, the other cited efforts offer an approach to addressing the coupling data issues with shared, robust components.

References

[1] Babuji, Chard, and Foster. Workshop on high performance serverless computing, hpdc 2011. https://highperformanceserverless.github.io/, 2021. online; accessed 9 November 2021.

[2] A. Champsaur, J. Lofstead, J. Dayal, M. Wolf, G. Eisenhauer, P. Widener, and A. Gavrilovska. Smartblock: An approach to standardizing in situ workflow components. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1301–1308. IEEE, 2017.

[3] CXL consortium. Compute express link. https://www.computeexpresslink.org/, 2021. online; accessed 9 November 2021.

[4] J. Lofstead, A. Champsaur, J. Dayal, M. Wolf, and G. Eisenhauer. Superglue: Standardizing glue components for hpc workflows. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 170–171. IEEE, 2016.

The New Science of Scientific Software Development

Addi Malviya Thakur
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Email: malviyaa@ornl.gov

Audris Mockus
Electrical Engineering and Computer Science
University of Knoxville, Tennessee
Email: audris@utk.edu

I. Challenge

The next scientific software era will involve developing software for Exascale architectures and employing Artificial Intelligence (AI) in scientific domains as part of a broader software development portfolio. The success of these efforts is likely to hinge on the ability and willingness of diverse domain scientists to contribute and adopt these specific technologies and other open-source innovations. The next challenge in scientific software development (SSD) comes from the knowledge generation and development of understanding related to the proper usage and application of rigorous, science-based methodologies to improve its development and adoption. With the rise of open-source software, virtually all software relies on the sustainability of the underlying software ecosystem, and it is critical that the science-based methodologies [5] focus on the qualitative, quantitative, and sustainability aspects towards realizing successful software development and use.

The challenges and opportunities discussed in this position paper target the next-generation science of scientific-software development and use. The diversity of perspectives, breadth of domain contribution, expertise from varying disciplines, applied methodologies, and experiences will be needed to expand the horizon of scientific software development. The past challenges [1], [3]–[7], [9]–[12] can be roughly grouped into lack of general software engineering training, coordination and collaboration essential for sustainability, and navigating technical landscapes of tightly interconnected software components. Next, we illustrate these challenges in more detail.

A. Software Engineering (SE) Training

Software sustainability and reproducibility have taken center stage in scientific software development. To achieve these goals, scientific software should be well-grounded in software engineering principles. Thus, a software development process should adhere to software engineering tents, architecture, and complex software artifacts. In the past, scientific software developers who represented multiple science domains (such as materials, nuclear, etc.) struggled with formal SE training resulting in monolithic and difficult to port code. A gradual sharing of such codes and packages across the community only resulted in further exacerbating the problem [6]. In addition, at times, inadequately following the formal software development methodology created reproducibility problems for critical scientific discoveries that uses such software programs [1], [9], [12] . AI is still evolving, and the need to utilize it in mainstream scientific software development would require more work. There is a need for AI-Driven best practices in scientific software development including the formalization of introducing AI/ML in the development cycle.

B. Coordination and Collaboration

As science becomes ever-complex and requires multi-domain contribution, SSD has become more challenging. There is a greater need than ever to coordinate and collaborate the development activities for the greater good. Multi-year scientific software development occurs in a competitive funding environment. It sometimes evolves as discoveries are made, resulting in complex team composition and dynamics that may impede scientific software development. Also, the culturally diverse and varying background of domain scientists and developers could result in difficulties in clearly communicating the requirements and responsibilities.

C. Navigating Technical Landscape

General software development is on an upward trajectory, with the release of more benchmarks, libraries, programming languages, message passing routines, etc. Traditionally, the scientific development environment struggled with the adoption of code scalability and portability across varying execution environments: As the scientific computing platforms evolve, the software development performed on a previous hardware platform may or may not be compatible with the latest one. This creates a significant issue in the long-term development cycles and investment. In addition, a perceived need for compatibility in the libraries and build environment standardization would create challenges in the next-generation development of software systems. Scientific software developers often need proper requirements specifications. The science of the development of scientific software aligns with adhoc invention trajectories, thereby making it really difficult to assess the requirement from the beginning [10], [11]. As discussed earlier, scientific reproducibility is a critical component today; validating and verifying the development could be another challenge in the current scenario where the computation could involve several thousand cores. Scientific software development happens as a function of scientific discoveries, which jeopardizes planning and engineering. Most scientific software developments start

with prototyping the complete domain problem. Such an approach makes it extremely challenging to scale up for a larger use case or broader applicability. In several cases, the development has to begin with a clean slate [3], [4], [7].

II. Opportunity

Open Source Software (OSS) provides not just the source code but also the incredibly detailed process data in the public issue tracking and version control systems. Over the last two decades, powerful empirical methods and big data infrastructure [8] to use such data for studies of practices that are essential for building vibrant and sustainable software ecosystems have been developed. Our key position is to discover these practices in general and scientific software in particular and develop training and tools to support wider adoption of such practices within the scientific software community.

Software Engineering for all fields and, especially, for science has become highly technically integrated and collaborative with scientific research and infrastructure software rarely built from scratch, but instead, on top of existing open-source libraries, packages, and source code. Thus greater coordination on technical activities would be timely at this time. The opportunity to complement the computing power of forthcoming exascale systems with recently developed complex Artificial Intelligence frameworks will usher transformative inventions and discoveries. The massive size of open source software and the lack of systematic approaches to discovering, evaluating, and reusing existing software pose several challenges, especially for scientists. Specifically, current practices of finding and reusing software assets are based on highly subjective prior experiences, individual professional network use, and online search engines. Furthermore, some assets are employed as external libraries while others are copied and maintained with the target project. These approaches do not take into account potential problems with reused assets or with assets they depend on and may result in problematic or not reproducible science as well as reduced ability to sustain such software long-term if the upstream projects are no longer maintained and/or contain defects, thus, endanger sustainability of all downstream efforts. Similarly, upstream projects need to maintain transparent objectives and predictable activities to avoid creating problems and disengage downstream projects.

III. Timeliness and maturity

We believe that SSD is at the cusp of transformative changes with the arrival of exascale computing platforms, AI/ML inclusion, incredibly rapid growth in the amount and importance of open source software, and data-driven decision-making across various scientific domains. To sum up, there couldn't be a better time than now to make breakthroughs for the next generation of scientific software development [2].

Empirical software development: Data-driven technologies are maturing to allow a finer analysis of the practice of scientific software development. This would enable the focus on scientific artifact generation rather than the development of software. In addition, a new focus on improving the SE practices and catalog and learning practices will be achieved.

Education and outreach: Designing revised curriculum to interactive programming environments will reduce learning curve and be productive in scientific software development.

Reusability: Greater emphasis on the reusability of software libraries and packages would accelerate the development cycle.

Scientific-software systems are growing larger and more complex. We're finally starting to see the interaction between computational science and SE communities, but more dialogue and studies are necessary.

The infrastructure and techniques related to measuring/modeling public source code supply chains have matured. As a result, it is now possible to study them exhaustively.

Academic-Industry-National lab partnerships have resulted in several novel discoveries, and coordination among these entities would allow the amalgamation of cohesive studies that would be timely and relevant across the board.

IV. ACKNOWLEDGMENT*

This manuscript has been authored by UT-Battelle, LLC under Contract No. DEAC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a nonexclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access.

It was also supported by NSF grants 1901102 and 2120429.

References

[1] Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. Scientific software development at a research facility. *IEEE Softw.*, 25(4):44–51, jul 2008.

[2] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.

[3] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *ICSE*, pages 550–559, 2007.

[4] Steve M. Easterbrook and Timothy C. Johns. Engineering the software for understanding climate change. *Computing in Science Eng.*, 2009.

[5] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science. *Inf. Softw. Technol.*, 67(C), 2015.

[6] Konrad Hinsen. Software development for reproducible research. *Computing in Science & Engineering*, 15(04):60–63, 2013.

[7] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report.

[8] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data. *International Journal of Empirical Software Engineering*, 2020.

[9] J. Yates Monteith, John D. McGregor, and John E. Ingram. Scientific research software ecosystems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, 2014.

[10] Rebecca Sanders and Diane Kelly. Dealing with risk in scientific software development. *IEEE Software*, 25(4):21–28, 2008.

[11] Judith Segal. A field study of the early stages of development of software for a scientific community. *CSC Work*, 2009.

[12] Judith Segal. Some challenges facing software engineers developing software for scientists. In *ICSE*, 2009.

Infrastructure-as-Code for Scientific Software and its Importance Towards Federated Infrastructure

Marshall McDonnell, Elton Cranfill, Jason Kincl, Addi Malviya-Thakur

I. Challenge

Development and Operations (DevOps) is a methodology that emphasizes meeting the goal of successfully developing and maintaining reliable software. Software is typically deployed and run on infrastructure (i.e., physical servers for scientific instruments, high-performance computing, physical or virtual networks, data storage). "Infrastructure-as-Code" (IaC) is a core DevOps practice that has emerged to capture the requirements and automate the creation of this infrastructure instead of a team manually documenting, managing, and configuring these resources.

A current challenge in software across industry is the reliability, reproducibility, and scalability of the deployment; this is the same in the case of large-scale scientific software. However, scientific software has additional complexity due to the heterogeneity of the software stacks (i.e., a mix of containerized and non-containerized applications), diversity of the hardware requirements (i.e., GPUs, FPGAs, low-latency networks), and the need to run at a specific physical location (i.e., neutron facilities, high-energy physics experiment, supercomputer)

Thus, the proposed challenge is that currently, to enable running large-scale workflows of heterogeneous software components and infrastructure across a federated group of either national or international facilities, investment in IaC tooling must be realized to reach that level of reliability, reproducibility, and scalability of the deployment.

II. Opportunity

A. How has Infrastructure-as-Code helped Industry?

A challenge in industry that IaC has addressed is bringing visibility to the state and configuration of an organization's infrastructure. IaC aims to take the confusion and error-prone aspect of a manual process out of creating infrastructure by automation and reduce complexity. The measurable benfits associated with the of adoption IaC are [1] 1) reduction of cost from automating deployments, 2) increased speed of deploying, and 3) reduction of errors and consistency with deployments. Case studies exemplifying the IaC benefits include Ambit Energy deploying x1,200 faster[2], TMX Group increasing infrastructure provisioning time by 75% [3], and Snowflake reducing cluster provisioning from a week to less than a day. [4].

Sompe popular general purpose IaC tools are Terraform, Pulumi, CloudFormation, Chef, Puppet, and Ansible. Docker and Kubernetes are similar IaC tools for container technologies.

B. Current Infrastructure-as-Code in Scientific Software

HPC capabilities via cloud services [5], [6], [7] and even solutions tailored specifically to scientific software applications [8] are becoming more prevalent. With this is the growing use of IaC in HPC. To support COVID-19 research, Google published open-source IaC scripts to provision infrastructure on Google Cloud Platform to run Folding@Home [9]. The Aristotle Federated Cloud project has developed IaC scripts for provisioning Kubernetes clusters with OpenMPI [10]. The work by Ivanova et al. [11] developed IaC for secure, fast, and powerful cloud computing environment for medical imaging analytics.

IaC can help address another critical aspect of bringing scientific software into production: security. Security best practices can be codified into IaC. Static analysis can be performed on IaC scripts to identifying security defects in the scripts [12] or even idnetify issues found in specific hardware[13], [14] . Continuously addressing these vulnerabilities iteratively improves the IaC modules for secure deployment of the scientific software.

C. Examples and the importance of Infrastructure-as-Code for Scientific Software

1) Early stage development for distributed scientific software: As popularity rises for microservice architectures, heterogeneity and complexity of scientific software deployment will increase. Automation of setting up environments and continuous deployments is crucial to run large, complex applications in a reliable, reproducible, and scalable manner.

IaC provides a unique capability for the early stages of startup development for distributed scientific applications related to intermittent funding. With IaC, just as easy as it is to spin up infrastructure for the application, one can also spin the infrastructure down. Even if all funding ceases for a period of time, the entire infrastructure can be de-commissioned / destroyed without concern. The entire project infrastructure can be re-created to a known working state with possibly one simple command execution. This emphasizes early adoption of IaC on a project for immediate benefit with the long term goal of refining and expanding the IaC capabilities.

2) Creating federated infrastructure for running scientific workflows: Another example is running scientific workflows across pooled infrastructure owned by many different institutions, or a "federated infrastructure". Adding an institution's infrastructure to the federation would be enabled by standing up a "deployment endpoint" to deploy and run workflow steps. This deployment endpoint would be something like

Kubernetes, Docker Swarm, or Nomad. The deployment endpoint could publish metadata that provides the capabilities and hardware to map workflow steps to deployment endpoints. This idea was similarly described by Bayser et al. [15] as *"software product lines"*.

An example of a workflow running on federated infrastructure would be 1) the workflow runs a step on an x-ray scattering instrument deployment endpoint, 2) submits a data transfer of the x-ray data to the next deployment endpoint at an HPC center, and 3) runs HPC data analytics on the x-ray data or supplemental simulations using the x-ray data as input.

Institutions joining the federation would require a low entry barrier, and the security, reproducibility, and reliability of the deployment endpoints be high. To facilitate this goal, deployment endpoint setups would be automated with IaC. Examples could include setting up Kubernetes deployment endpoints on either an HPC resource, such as Slate [16] for the Summit supercomputer [17], or a Rucio data lake containing high-energy physics experimental data.[18].

3) Infrastructure discovery for a given workflow on a federated infrastructure: To extend the previous example, for scientific workflows that contain specialized hardware requirements for each different step of the workflow, it may not be possible to create common deployment endpoints in the federated infrastructure or hard to even find if and where each step of the workflow can be run.

Yet, with the addition of having data centers in the federation setup with "composable infrastructure" that house a large variety of compute and storage resources, one can envision being able to run "infrastructure discovery" against the federation. Composable infrastructure is a category of data center infrastructure that has the ability to connect compute (i.e. CPUs, GPUs, FPGAs, etc.), storage, and networking into pools of resources that are then provisioned through software APIs that can be available for on-demand allocation (i.e., "composable") [19]. One could envision a federated composable infrastructure being able to serve a query whether all these resources existed in the federation for a given scientific workflow, where each step of the workflow can run or if can even be accommodated.. Then, using the software APIs to provision infrastructure, IaC could be dynamically generated from the workflow definition and used to request the composable infrastructure for the workflow from the federation.

4) Reproducibility: Reproducibility is a key component and a growing concern across all of science. [20] IaC necessarily provides artifacts for scientific software deployments that can be packaged as part of the scientific software for future work. Community movements inside of IaC such as GitOps espouse principles of declarable and immutable state[21].

III. Timeliness

IaC is already being adopted in scientific software. With larger programs like the Exascale Computing Project ending soon, there is a focus on sustainability now from the Advanced Scientific Computer Research program. IaC has proven to address reliability, reproducibility, and scalability, which all will help achieve sustainability of scientific software.

Emphasizing the need for IaC practices and guidance on standardization for adoption in scientific software development will help facilities convergence towards larger goals like a federated infrastructure to run distributed, inter-facility scientific workflows. It will take several years to reach full federated infrastructure. Yet, iterative gains can be made with individual growth of the federation on a yearly basis.

References

[1] Red Hat, "What is infrastructure as code (iac)?" 2020. [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

[2] Puppet Labs, "Ambit Energy's customer story for deploying 1,200 times faster due to IaC with Puppet," 2021. [Online]. Available: https://puppet.com/resources/customer-story/ambit-energy

[3] R. M. Mark Salam, "Up and to the right," 2021. [Online]. Available: https://www.hashicorp.com/case-studies/tmx-group

[4] Pulumi, "Pulumi's case study for snowflake, a data platform." [Online]. Available: https://www.pulumi.com/case-studies/snowflake/

[5] Amazon Web Services, 2021. [Online]. Available: https://aws.amazon.com/hpc/

[6] Azure, 2021. [Online]. Available: https://azure.microsoft.com/en-us/solutions/high-performance-computing/

[7] Google Cloud Platform, 2021. [Online]. Available: https://cloud.google.com/solutions/hpc/

[8] Penguin Computing, "Penguin Computing HPC Cloud Scientific Applications," 2021. [Online]. Available: https://www.penguincomputing.com/pod-hpc-cloud/pod-applications/

[9] Google Cloud Platform, "IaC for provisioning Folding@Home on GCP," 2021. [Online]. Available: https://github.com/GoogleCloudPlatform/terraform-folding-at-home

[10] Aristotle Cloud Federation, "IaC for provisioning Kubernetes clusters with OpenMPI on GCP," 2021. [Online]. Available: https://github.com/federatedcloud/kubernetes-mpi-clusters

[11] D. Ivanova, P. Borovska, and S. Zahov, "Development of PaaS using AWS and Terraform for medical imaging analytics," in *PROCEEDINGS OF THE 44TH INTERNATIONAL CONFERENCE "APPLICATIONS OF MATHEMATICS IN ENGINEERING AND ECONOMICS"*, ser. AIP Conference Proceedings, Pasheva, V and Popivanov, N and Venkov, G, Ed., vol. 2048, 2018.

[12] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *2019 IEEE/ACM 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE 2019)*, ser. International Conference on Software Engineering, 2019, pp. 164–175.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[14] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[15] M. de Bayser, L. G. Azevedo, and R. Cerqueira, "Researchops: The case for devops in scientific applications," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 1398–1404.

[16] Slate, "Slate: Kubernetes cluster with access to Summit," 2021. [Online]. Available: https://docs.olcf.ornl.gov/services_and_applications/slate/overview.html#what-is-slate

[17] Summit, "Summit: Oak Ridge National Laboratory's 200 petaflop supercomputer," 2021. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[18] Rucio. [Online]. Available: https://rucio.cern.ch/

[19] Jeff Nicholson, Mark Miquelon, "Open Composable API v1.0," 2021. [Online]. Available: https://www.opencompute.org/documents/open-composable-api-for-ocp-2019-06-24-pdf

[20] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, pp. 452–454, 2016.

[21] G. W. Group, 2020. [Online]. Available: https://opengitops.dev/

Microproductivity in the Lab: Towards a New Future of Human-Centered Scientific Software Development

Jacob McLemore
jmclemo6@vols.utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Alex C. Williams
acw@utk.edu
University of Tennessee
Knoxville, Tennessee, USA

1 CHALLENGE

Advances in modern science have come to rely heavily on the maintenance and development of scientific software and software developers. Among the many challenges that permeate scientific software teams, the most pressing is a "knowledge problem". Specifically, what must be done to allow scientific software development groups to productively harness their members' knowledge that is required to build scientific software?

The ability to answer these questions relies on the individual and their relative context. Often, a new scientific software developer is required to hold advanced degrees and have multiple years of professional experience at a minimum. Once this new developer has joined a scientific software development team, they will quickly find themselves in an environment where traditional software methodologies aren't of use due to the highly unique knowledge and skills of team members. Workflows that are incapable of addressing this can inhibit productivity due to a team not being able to harness their collective knowledge to achieve their goals.

Addressing this *"knowledge transfer"* challenge is crucial to individuals and teams alike that rely on the development and maintenance of scientific software. Here, we argue for a new future of human-centered scientific software development that leverages machine learning methods in conjunction with recent advances in interactivity to create new software development experiences on the basis of accessibility, moility, and intelligence.

2 OPPORTUNITY

The software industry recognizes developer productivity as an important area for research and innovation. In 2014, a report from the United States Department of Energy noted productivity as one of the top 10 challenges in exascale computing[7]. A later U.S. DOE report from 2018 highlighted that a future where scientific software productivity remains the same as in the past will result in unsustainable levels of software maintenance being required [10]. A U.S. DOE report restated this challenge in 2020 [3]. Here, we argue for a new frontier of research in scientific software development centered around "Microproductivity", a process in which larger macrotasks are algorithmically sub-divided into small, context-free microtasks that require little time and effort to complete.

2.1 Microproductivity

Microproductivity has been well-studied in information workplaces Teevan, Iqbal, and Veh studied microproductivity in the context of colloborative writing tasks and found that microproductivity-focused writing helped participants produce useful text in a short amount of time [9]. Williams et. al. investigated mobile microproductivity workflows for computer programming tasks with Merury

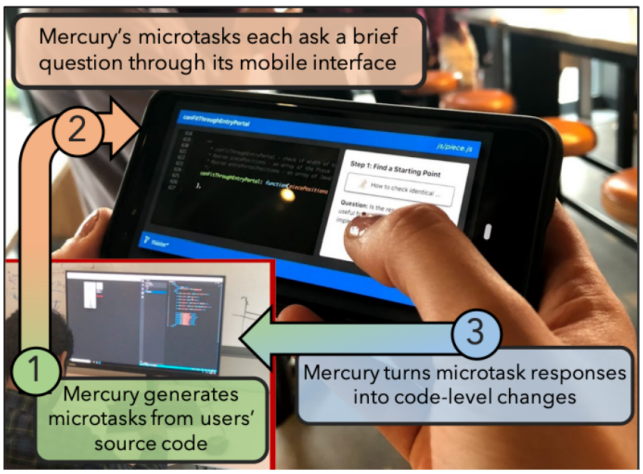


Figure 1: *Mercury*'s microproductivity workflow. [12]

(see Figure 1). Participants in this study reported that a microproductivity workflow was enjoyable, helped them find a "starting point", and helped them resume work faster after being interrupted [12]. Iqbal and Teevan et. al. studied multitasking during periods of divided attention and found that participants greatly preferred and were more productive when using microproductivity workflows [4]. Overall, these are only a small selection of studies that have shown microproductivity workflows can improve productivity and workers' enjoyment of work.

2.2 Intelligent Microproductivity Workflows

An important aspect of microproductivity systems is that they often utilize new algorithms for identifying or generating tasks in larger macrotasks. For example, Mercury automatically generates two types of microtasks from source code: (1) Grounded Microtasks (e.g. assisted unit-testing) and (2) Exploratory Microtasks (e.g., assisted search) [12]. More generally, Zhang et. al. studied systems for decomposing larger tasks into microtasks suitable for microproductivity workflows [13]. More intelligently, Fritz and Ou et. al. developed a model that could determine how familiar someone was with source code [1] and this model could be used to assign generated microtasks to those most capable of performing them. We believe that these works suggest that the management of time and effort can be orchestrated and assisted with AI-powered tools and microproductivity workflows.

These advances alone show great promise, but they do not solve the "knowledge problem" that has been previously mentioned. However, by combining these advances in artificial intelligence and

microproductivity with work from the fields of cognitive science and teamwork, we believe that we can attempt to do so. This is possible by harnessing the power of transactive memory systems. Transactive memory systems are defined as emergent states where team members work towards developing a shared ontology of their expertise and roles and manage information in dedicated knowledge repositories [11]. Research has shown that groups with a TMS are better performing [5], are more creative [2], and learn more than groups without a TMS [6] [8]. A microproductivity workflow could be part of and a complement to a group's TMS. Future research must be done on how microproductivity workflows can be adopted by a scientific software engineering group and if those workflows can assist those groups in developing more robust transactive memory. By making it easier for members of a scientific software development group to do their work and develop a TMS to share their expert knowledge with one another, the group as a whole could see increased productivity.

2.3 Visions: Toward a Lab of the Future

Research in human-computer interaction have historically viewed the "information workplace" as a setting that is rich with discovery. Scholars have, for example, used the workplace to learn how people manage interruptions, how people might use new technologies to improve their wellbeing, and even how people use microproductivity systems. Within our broader argument for introducing microproductivity into the laboratory context, we also argue that the laboratory itself can and should be recognized as an *academic* information workplace that has generally gone unnoticed.

Our proposed vision for introducing microproductivity approaches into national laboratories aims to resolve knowledge transfer. By better supporting knowledge transfer with microproductivity systems, we subtly introduce a wealth of opportunities that run parallel to our broader goal. For example, microproductivity systems enable:

- **Limited-Attention Scientific Software Development**: RSDEs and SDEs can use microproductivity interfaces that simplify or orchestrate desktop tasks when working with limited attention, e.g. while on a Zoom call.
- **Mobile Opportunities Scientific Software Development**: RSDEs and SDEs can contribute to software development from their mobile devices with new microproductivity interfaces. Interfaces can include standard GUI-based mobile applications to voice-powered conversational assistants.
- **Citizen-Powered Scientific Software Development**: Members of the general public can engage with the maintenance and development of scientific software by completing tasks that require no context and no expertise in computing.

Importantly, each of these opportunities introduces new challenges related to understanding who a task should be distributed to, when it should be distributed to them, and how it is distributed. Here, we argue that machine learning models, or a human operator with tacit knowledge, can be leveraged to resolve these challenges.

3 MATURITY

Productivity within scientific software communities has never been more important. Meanwhile, advances in our understanding of microproductivity and task interactivity have introduced new ways of "getting things done". Similar recent advances in artificial intelligence and machine learning can be combined with microproductivity to create new AI-powered interactions that are designed to alleviate the stresses that come with managing and completing scientific software work at national laboratories. By combining these new advances with more mature cognitive science research of transactive memory systems, a new future of human-centered scientific software work could be upon us. Only further research can give us a definitive answer.

REFERENCES

[1] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 385–394, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Francesca Gino, Linda Argote, Ella Miron-Spektor, and Gergana Todorova. First, get your feet wet: The effects of learning from direct and indirect experience on team creativity. *Organizational behavior and human decision processes*, 111(2):102–115, 2010.

[3] Michael A. Heroux, Lois McInnes, David E. Bernholdt, Anshu Dubey, Elsa Gonsiorowski, Osni Marques, J. David Moulton, Boyana Norris, Elaine Raybourn, Satish Balay, Roscoe A. Bartlett, Lisa Childers, Todd Gamblin, Patricia Grubel, Rinku Gupta, Rebecca Hartman-Baker, Judith C. Hill, Stephen Hudson, Christoph Junghans, Alicia Klinvex, Reed Milewicz, Mark Miller, Hai Ah Nam, Jared O'Neal, Katherine Riley, Ben Sims, Jean Schuler, Barry F. Smith, Louis Vernon, Gregory R. Watson, James Willenbring, and Paul Wolfenbarger. Advancing scientific productivity through better scientific software: Developer productivity and software sustainability report. 1 2020.

[4] Shamsi T. Iqbal, Jaime Teevan, Dan Liebling, and Anne Loomis Thompson. Multitasking with play write, a mobile microproductivity writing tool. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 411–422, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Kyle Lewis. Measuring transactive memory systems in the field: scale development and validation. *Journal of applied psychology*, 88(4):587, 2003.

[6] Kyle Lewis, Donald Lange, and Lynette Gillis. Learning transfer. 2005.

[7] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, Al Geist, Rud Haring, Jeffrey Hittinger, Adolfy Hoisie, Dean Micron Klein, Peter Kogge, Richard Lethin, Vivek Sarkar, Robert Schreiber, John Shalf, Thomas Sterling, Rick Stevens, Jon Bashor, Ron Brightwell, Paul Coteus, Erik Debenedictus, Jon Hiller, K. H. Kim, Harper Langston, Richard Micron Murphy, Clayton Webster, Stefan Wild, Gary Grider, Rob Ross, Sven Leyffer, and James Laros III. Doe advanced scientific computing advisory subcommittee (ascac) report: Top ten exascale research challenges. 2 2014.

[8] Jacob McLemore Senjuti Dutta Fatema Akbar Ted Grover Thomas Breideband Judith Borghouts Yuwen Lu Gloria Mark Austin Z. Henley Rhema Linder, Chase Hunter and Alex C. Williams. Characterizing work-life for information work on mars: A design fiction for the new future of work on earth. *Proceedings of the ACM on Human-Computer Interaction*, 6(GROUP):1–27, 2022.

[9] Jaime Teevan, Shamsi Iqbal, and Curtis von Veh. Supporting collaborative writing with microtasks. In *CHI 2016*. ACM - Association for Computing Machinery, May 2016.

[10] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Anshu Dubey, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. Extreme heterogeneity 2018: Productive computational science in the era of extreme heterogeneity report for doe ascr basic research needs workshop on extreme heterogeneity january 23–25, 2018. 1 2018.

[11] Daniel M Wegner. Transactive memory: A contemporary analysis of the group mind. In *Theories of group behavior*, pages 185–208. Springer, 1987.

[12] Alex C. Williams, Harmanpreet Kaur, Shamsi Iqbal, Ryen W. White, Jaime Teevan, and Adam Fourney. Mercury: Empowering programmers' mobile work practices with microproductivity. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, page 81–94, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Yi Zhang, Sujay Kumar Jauhar, Julia Kiseleva, Ryen White, and Dan Roth. Learning to decompose and organize complex tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2726–2735, Online, June 2021. Association for Computational Linguistics.

Considering Activity-Centered Design in Scientific Software Development

1st Annika Meinecke
German Aerospace Centre (DLR)
Institute for Software Technology
Cologne, Germany
annika.meinecke@dlr.de

2nd David Heidrich
German Aerospace Centre (DLR)
Institute for Software Technology
Weßling, Germany
david.heidrich@dlr.de

3rd Katharina Dworatzyk
German Aerospace Centre (DLR)
Institute for Software Technology
Cologne, Germany
katharina.dworatzyk@dlr.de

4th Annika Wohlan
German Aerospace Centre (DLR)
Institute for Software Technology
Cologne, Germany
annika.wohlan@dlr.de

I. Challenge

Well designed software products base their requirements on the particular needs of their user groups. In industries, companies that aim at designing commercially successful software products have long since recognized the importance of user feedback and have started to involve their users early on in the design process. The motivation for this is evident: Having a well defined user group and a profound understanding of their needs results in great benefits for both users and developers. The user, on the one hand, benefits from products that allow for a more efficient, more effective or safer use, while the developer, on the other hand, benefits from the need for fewer re-designs [1].

Approaches such as user-centered design, activity-centered design, or design thinking conceptualize the development process as a series of different interdependent steps, at all of which user feedback can make a substantial contribution to the final product. The basic idea is that identifying the needs of potential users, considering their perspective of use, and defining the activities users wish to accomplish by using this specific software, will ultimately lead to better software products. Following an approach where user feedback is constantly integrated in the development process might lead to a better acceptance, while an approach that focuses more on characterizing the underlying activity might even enhances its re-usability. Both features, that is acceptance by a large community and re-usability, appear to be essential for scientific software as well.

In scientific software development, however, we see that potential user groups are only rarely considered. This observation might be partly explained by the fact that scientific software is often developed by small research groups or even single researchers who, at the same time, are the main user group. In our experience, scientific software is mostly developed by researchers by and for themselves. This comes with a few limitations. First, domain scientists often lack the time or proper education to face the complex challenges of software engineering. Secondly, research software is not regularly shared across the borders of working groups, which further suggests that feedback, for instance in form of regular code reviews, is not often provided or considered. While peer-review of manuscripts is part of the process when publishing research results, it seems that yet too little emphasis is put on peer-review of research software. This lack of exchange between research groups, institutes, fields or even domains becomes even more significant when considering how often different software projects are being worked on in parallel to solve essentially similar problems.

Most scientific software is developed to fulfill specific tasks a single researcher or a small group of researchers face in their daily work [2], [3]. Considering that other researchers, as they face similar problems in their research, might profit from using already existing scientific software or contributing to it, is often neglected in the process. Focusing on requirements of few or even a single individual might, however, result in tools that are no longer usable or accepted by the majority as Norman argues in his critique of human-centered design [4]. This could indeed serve as the key explanation of one of the major challenges of research software. We suggest that activity-centered design provides a suitable lens to understand this specific challenge of scientific software development, namely the too narrow focus on one's own research activities, and to provide guidance to see *the bigger picture*.

II. Opportunity

Adopting approaches from human-centered design and integrating them in the process of research software development, as it is done by companies developing software products with the intention to gain profit, will not always be feasible. Investigating potential user groups and their needs through extensive interviews or large-scale surveys requires a profound understanding of the methods at hand as well as a varying amount of time and financial resources; in particular, if one

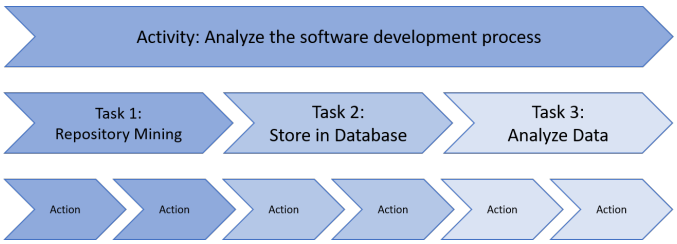


Fig. 1. Analysing the software development process

only has a vague idea of who the users might be. Nonetheless, we believe that everybody developing software should ask themselves:

- Who, besides myself, could be interested in using my software?
- What do I intend to achieve with my software?
- How can I make it easier for others to use my software?

We are convinced that research software can benefit a lot from using an activity-centered approach when designing software. According to Norman [4], there are different levels to describe the use of a tool. Activities in this context form the highest level and are composed of different tasks that are necessary to successfully perform an activity. Tasks themselves are composed of different actions, which in turn are composed of different operations.

To give an example from our own research, we describe the analysis of the software development process (Figure 1), which is our main activity. The tasks we need to do to accomplish this activity include:

1) Collect data from the development process by doing repository mining. We use, for instance, GitLab2Prov [5], [6] to extract retrospective provenance data of the development process through the GitLab[1] API.
2) Store the data in a database. To store the previously extracted data in a Neo4J[2] graph data base, we use Prov2Neo [6], [7].
3) Finally, to analyze the data through visual analysis, we use different visualization techniques [8].

While each of these tasks are necessary to accomplish the activity as a whole, the individual tasks can stand on their own and can be conceptualized as independent from each other. Accordingly in the development process, software solutions can be designed for each step separately. Other researchers can then profit from such designed software, as they might be interested in gathering data or storing it in a database, but not in visualizing the results. For this step they would be developing their own analysis tools. As a result, tools are not limited to the specific needs of a certain research group and the user group of can easily expand since the intended use of each of the tools is clearly defined.

[1]https://about.gitlab.com/features/
[2]https://neo4j.com/product/neo4j-graph-database/

In general, if the reasoning behind structures and designs are clear, others can adapt to it more easily and even complex systems can be learned and applied by other users [4].

III. Timeliness or maturity

Providing and gaining access to research software has never been easier. Code hosting platforms like GitHub or GitLab allow us to share code with fellow researchers as well as to find published software which might be useful for our own research. In addition, we can build upon the tools written by others and adapt and improve them to fit our individual purposes. However, the large number of public, yet unused or unmaintained research software also makes us aware of fundamental problems in the design of such software.

Activity-centered design can help re-think one's own activity as being composed of different tasks which can be supported by different software solutions. The realization that involving users of software in its development process has lead to better designed software and improved the user experience in many cases. Scientific software should not be considered a tool necessary to conduct ones own research rather than something that could be used, re-used and adapted by people beyond the own research group.

References

[1] C. Abras, D. Maloney-Krichmar, J. Preece *et al.*, "User-centered design," *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, vol. 37, no. 4, pp. 445–456, 2004.
[2] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. Ieee, 2009, pp. 1–8.
[3] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.
[4] D. A. Norman, "Human-centered design considered harmful," *interactions*, vol. 12, no. 4, pp. 14–19, 2005.
[5] C. de Boer and A. Schreiber, "Dlr-sc/gitlab2prov: Gitlab2prov 0.5," Jun. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5009043
[6] A. Schreiber, C. de Boer, and L. von Kurnatowski, "Gitlab2prov—provenance of software projects hosted on gitlab," in *13th International Workshop on Theory and Practice of Provenance (TaPP 2021)*. USENIX Association, Jul. 2021. [Online]. Available: https://www.usenix.org/conference/tapp2021/presentation/schreiber
[7] A. Schreiber, L. von Kurnatowski, and C. de Boer, "Analyzing software engineering processes with provenance-based knowledge graphs," in *2021 IEEE Aerospace Conference (50100)*. IEEE, 2021, pp. 1–11.
[8] A. Schreiber, L. Kurnatowski, A. Meinecke, and C. Boer, "An interactive dashboard for visualizing the provenance of software development processes," in *2021 Working Conference on Software Visualization (VISSOFT)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2021, pp. 100–104. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/VISSOFT52517.2021.00019

Developing a Sustainable Software Ecosystem for Tools, Runtime Libraries, and Applications

John Mellor-Crummey (johnmc@rice.edu) and Xiaozhu Meng (xm13@rice.edu)
Rice University

Challenge

Today, it is impossible to build sustainable software tools for monitoring the performance of sophisticated parallel applications for existing and emerging GPU-accelerated supercomputers. There are no rules that govern what applications and runtime libraries may do or how they interact with tools, other libraries, or the operating system. As a result, tool developers need to observe runtime and tool libraries to determine what they do and infer constraints on how such libraries may be observed or controlled. Based on the behaviors of libraries used by applications or libraries used to develop the tools themselves, tools must be engineered to avoid any problems that arise as a result of observed behaviors. This approach does not lead to sustainable software for tools because tools can be broken at any time by changes to one or more of the libraries whose actions they must observe and/or control.

To make these challenges a bit more concrete, below we list a few examples of the many issues encountered when attempting to develop a performance tool to monitor sophisticated GPU-accelerated applications.

- HPE's MPI runtime creates threads using a function pointer to pthread_create obtained from the pthread library using dlsym. Such creation of threads cannot be monitored by pre-loading a wrapper for pthread_create - the typical mechanism used by tools.
- IBM's Spectrum MPI wraps dlsym to intercept and monitor calls to cudaMalloc to compensate for the fact that NVIDIA refused to provide a proper monitoring API. Spectrum MPI's use of dlsym causes deadlock when used with any tool that wraps dlopen to track loading of dynamic libraries.
- NVIDIA's CUDA Performance Tool Interface, known as CUPTI, creates threads. These threads should not be measured because they only exist to support performance tools.
- AMD's implementation of support for the OpenMP tools API (OMPT) in LLVM's libomptarget (a device-independent runtime for OpenMP offloading) uses threads to flush OMPT trace records. Monitoring such threads and having a tool try to flush GPU operations initiated by these threads caused deadlock.

Coping with such issues in an ad-hoc fashion leads to a brittle software ecosystem in which tools are difficult to develop and prone to fail if any library details change over time.

Opportunity

To make software tools for heterogeneous supercomputers sustainable, the DOE software community needs to recognize the fragility of the current approach and develop a set of software mechanisms and policies that

- make it possible to compose an application with one or more sophisticated libraries,
- enable the libraries to coordinate their actions as needed,
- provide tools with necessary observability and control, and
- ensure that applications and libraries must meet their obligations to the software ecosystem.

Development of mechanisms and policies that make it possible to build reliable and sustainable tools is a challenge at several levels.

First, it will require identification of classes of problematic interactions that must be avoided. Second, it will require development of suitable set of mechanisms in both Linux and libraries that support introspection, observability, and control that will enable libraries to safely compose their operations and tools to monitor and control interactions in a safe and reliable fashion.[1] Third, it will require development of policies and best practices that libraries and tools should employ to sustainably support composition of sophisticated software components.
Fourth, it would be useful to explore mechanisms for enforcing that libraries fulfill their obligations.

Timeliness

GPU-accelerated supercomputing has become the foundation for computation-intensive modeling, simulation, and data analysis.

The multiplicity, sophistication, and complexity of communication libraries, runtime libraries, and tool libraries, and tools for programming GPU-accelerated systems creates an urgent need for developing a standard and mechanisms for coordinating the actions of interacting components in the software stack. Tools developed for Aurora, Frontier, and Perlmutter are unlikely to be readily sustainable without significant changes to the software ecosystem.

References

1. John Mellor-Crummey, Xiaozhu Meng, Jonathon Anderson, Mark Krentel. Requirements for Glibc LD_AUDIT on Leadership Computing Platforms. Initial version: May 13, 2021. Last updated: Nov. 17, 2021. Available as https://bit.ly/glibc-ldaudit.

[1] Glibc's LD_AUDIT facility promises many of the necessary capabilities. Unfortunately, we have identified seven issues that cause failure, incorrect operation, or a performance degradation of up to 10x performance degradation of calls to short dynamic library routines. Mechanisms in Linux clearly need work to provide the level of support necessary.

Deep Learning Assisted Performance Analysis and Diagnosis

Xiaozhu Meng (xm13@rice.edu) and John Mellor-Crummey
Rice University

Challenge

Today, most of the top 10 supercomputers are equipped with GPU accelerators to boost performance [5]. This trend of heterogeneous computing leads to increasingly complex and diverse programming models, runtime libraries, and architectural idiosyncrasies. These factors make it difficult for programmers to achieve high performance with applications.

Performance tools such as HPCToolkit monitor program execution, attribute performance measurements to source code, and provide an interactive user interface for users to navigate through performance results. However, the process of performance tuning is manual and laborious. There are often three main steps through a successful tuning journey: (1) identifying the performance problem, (2) diagnosing the root cause of the problem (where HPCToolkit observes a performance problem is not necessarily the root cause of the problem), and (3) fixing the problem. All three steps are mostly done in a manual fashion and it often requires deep knowledge of the tool and applications to even finish these steps.

As team members of the HPCToolkit project, we envision the need for automated performance analysis for identifying performance problems, diagnosing root causes, and recommending potential fixes.

Opportunity

We envision the amount of performance data to continue to grow as developers run code at larger scales than ever and applications become more sophisticated with increasing numbers of software components. Deep learning has been successful in ingesting large data for many application domains, such as image processing and natural language processing. We expect that the use of deep learning to automate performance analysis will reduce manual efforts and increase tuning productivity.

The success of this idea will depend on several factors, including whether we can design deep learning models to learn performance characteristics from code, how to capture heterogeneous computing characteristics in learning models, and how to incorporate domain knowledge. We expect that it will be essential to develop new deep learning technologies tailored for this domain.

Timeliness or maturity

The emergence of “Big Code” shows the feasibility of learning program structure and functionality characteristics through deep learning techniques [1]. A natural next step is to learn performance characteristics from code. Traditional program analysis relies on graph representation of code, such as control flow and data flow graphs. Recent studies have shown that Graph Neural Networks (GNN) can take graph representations of code as input to learn code properties to perform tasks such as procedure name completion [2] and data type inference for machine code [3]. The use of GNN will be different from traditional machine learning approaches where the models are based on pre-computed code features, and thus has the advantage that GNN may learn more flexible representation correlated to performance.

A specific example showing the promise of this direction is CompilerGym [4], which is a project led by Facebook that uses various learning approaches to determine the best compiler optimization flags for binary size and runtime. This project shows the feasibility of learning performance characteristics from a dataset of code.

Success of this research area will lead to simplified and more effective performance tuning experience. We envision the following capabilities from a machine learning component of a performance tool: (1) highlight performance problems in an application and avoid the need for a user to sift through measurement data to find performance problems; (2) identify root causes of problems rather than relying on users users to identify root causes; (3) recommend potential suggestions for fixing performance problems.

Reference

[1] Learning from “Big Code”, http://learnbigcode.github.io/
[2] Jeremy Lacomis et al. “DIRE: a neural approach to decompiled identifier naming”. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19).
[3] Miltiadis Allamanis et al. “Typilus: neural type hints”. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).
[4] Chris Cummins et al. "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research." arXiv preprint arXiv:2109.08267 (2021).
[5] Top 500. The List. November, 2021. https://www.top500.org/lists/top500/2021/11.

When Can We Trust Your Models?

Tim Menzies, NC State, Computer Science, timm@ieee.org

CHALLENGE: We seek to support a "model economy" where scientific models are available for use across the web (perhaps for some fee for service). How can we share models across the internet where model owners can maintain control over their product, while others can assess the validity and fairness of those models?

There are valid privacy, pragmatic and proprietary reasons why model owners may not want to share too much information about their scientific models. For example, model owners want to hide details, least competitors reverse engineer their contents. Also, the data used to build a model may reveal sensitive information about individuals. Furthermore, owners of the models want to keep those models details in-house, at least until they recoup their development cost via renting out their model. Lastly, for models used in some safety-critical context, certification bodies might want to retain control of the official certified model, lest unprincipled (or sloppy) developers generate misleading or unsafe copies of that model.

At the same time, while there are many good reasons for keeping model details private, there are also equally compelling reasons for making all aspects of a model publicly accessible. Sadly, there are too many recent examples of models with lamentable and discriminatory properties. For example,

- Amazon had to scrap an automated recruiting tool as it was found to be biased against women[1].
- Also, a recidivism assessment models used by the criminal justice system was found to be more likely to falsely label black defendants as future criminals (twice as often as white defendants)[2] .

Currently AI explanation tools can succinctly study complex model, but those same methods allow malicious actors to hide certain discriminatory properties of their model. .Slack et al.[3] have recently shown that explanation algorithms are vulnerable to malicious human operators trying to hide discriminatory models. In their work, they discuss explanations generated via stochastic methods. For example, as shown in Figure 1a, the model-agnostic LIME explanation algorithm[4] generates random instances near to some example. After building some linear models from that generated data, LIME can explain what factors can change the classification of a particular example.

The problem here is shown in Figure 1b. Here, Slack et al. show the raw data (in **blue**) and data synthesized by LIME (in **red**). While some **blue** dots are near the **red** dots, manya re not, Slack et al. showed that automatic classifiers they are distinguishable via automatic classifiers. That is. the model can distinguish between (a) queries-about-explanation from (b) normal-queries seen in day-to-day operation. This means that ***malicious actors can lie about discriminatory properties of their model by offering model1 to queries-about-explanation and model2 to normal-queries from day-to-day operation.***

Figure 1: About LIME.

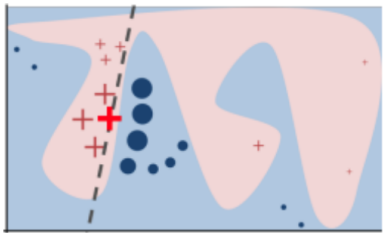


Figure 1a: *LIME samples around the red cross to learn delta between blue,red classes.*

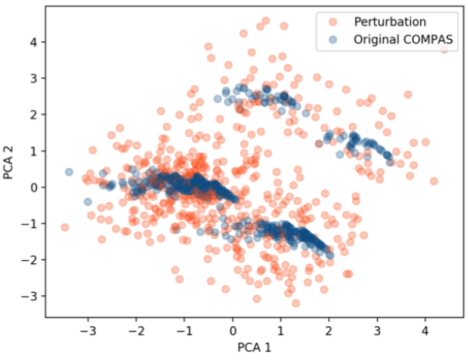


Figure 1b: *Red/blue = original/invented data. From [36].*

---

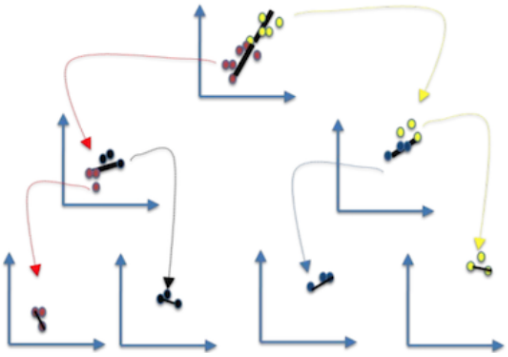
[1] "Amazon scraps secret ai recruiting tool that showed bias against women," Oct 2018. [Online]. Available: https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G

[2] Machine bias," www.propublica.org, May 2016. [Online]. Available: https://www.propublica.org/article/ machine- bias- risk-assessments- in- criminal- sentencing

[3] D. Slack, S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju, "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods," in *3rd AAAI/ACM Conference on AI, Ethics, and Society*, 2020.

[4] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.293977

OPPORTUNITY: One response to the issues raised by Slack et al. is to make LIME's artificial examples indistinguishable from the real data (i.e. to ensure that all the **blue** dots fall at the same location as the **red** dots). To that end, Ji et al.[5] applied extensive and elaborate Monte Carlo sampling to learn everything they can about the distributions in the data. But their sampling methods made numerous normality assumptions about their domain, and many domains do not correspond to those assumptions. Also, their methods only detected unfair models-- they did not go to the next step of offering mitigation methods. Further, despite their best efforts, their methods still made enough queries to the model such that the Figure 1b results might hold; i.e. a model could distinguish the special queries (the **red** dots) being made by Ji et al. This, in turn, means that that model could then offer misinformation to the Ji et al. queries.



Rather than patch the Ji et al. methods, we take a different approach. If the problem is that the model in Figure 1b can distinguish the fairness queries, then make fewer queries. Semi-supervised learning[6] methods learn the shape of data, then only samples in different parts of that shape. For example, the clustering method shown at right (a) finds two far points; then (b) separates the data according to its distance to the far points, then (c) recurses on each half (formally, this is the FASTMAP procedure that implements a non-parametric analog to principal component analysis[7]). Then, to find an approximation for the data, we need only generate a few **red** dots for each leaf cluster.

TIMELINESS or MATURITY: Stepping back from our specific solution, we say that now is the time to explore issues of trust and accountability. Recently, the requirements for ethical software have become quite complex. The European Union, Microsoft, and the IEEE have all released white papers discussing software[8] [9] [10], While these documents differ in the details, they all agree that ethical AI must be "FAT"; i.e. fair, accountable and transparent. It is no longer enough to merely validate software. Now we need to ship software with auditing tools built into their very structure. Communities of model users need tools that let them trust that they can trust each other.

Semi-supervised learning is the "quiet revolution". This kind of reasoning has recently matured to the point where complex systems can be approximated by a few points. This technology has the potential to enable trusted model sharing. But to prove that potential, tools like those discussed here have to be generalized to many applications. Also, we need to understand how good are the approximations generated in this manner.

---

[5] D. Ji, P. Smyth, and M. Steyvers, "Can i trust my fairness metric? assessing fairness with unlabeled data and bayesian inference," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*,

[6] O. Chapelle, B. Scholkopf, and A. Zien, Eds., Semi-Supervised Learning. The MIT Press, 2006. [Online]. Available: http://dblp.uni-trier.de/db/books/collections/CSZ2006.html

[7] Platt, J. (2005). FastMap, MetricMap, and Landmark MDS are all Nystrom Algorithms. *AISTATS*.

[8] "Ethically-aligned design: A vision for prioritizing human well-begin with autonomous and intelligence systems." 2019.

[9] "Ethics guidelines for trustworthy artificial intelligence." 2018. [Online]. Available: https://ec.europa.eu/digital-single-market/en/news/ethics-guidelines-trustworthy-ai

[10] "Microsoft AI principles," 2019. [Online]. Available: http://tiny.cc/ Microsoft

Building Bridges: Establishing a Dialogue Between Software Engineering Research and Computational Science

Reed Milewicz
Department of Software Engineering and Research
Sandia National Laboratories
Albuquerque, NM
rmilewi@sandia.gov

Miranda Mundt
Department of Software Engineering and Research
Sandia National Laboratories
Albuquerque, NM
mmundt@sandia.gov

***Abstract*—There has been growing interest within the computational science and engineering (CSE) community in engaging with software engineering research – the systematic study of software systems and their development, operation, and maintenance – to solve challenges in scientific software development. Historically, there has been little interaction between scientific computing and the field, which has held back progress. With the ranks of scientific software teams expanding to include software engineering researchers and practitioners, we can work to build bridges to software science and reap the rewards of evidence-based practice in software development.**

***Index Terms*—scientific software development, computational science and engineering, software engineering research, software science**

I. Challenge

Software engineering research (or SE research) is the systematic study of software systems and their development, operation, and maintenance [1]. A branch of computer science, software engineering research builds a bridge between the theory of computer programming and its practice as software engineering. It is likewise methodologically diverse, placing formal and empirical software analysis alongside human factors; it applies a sociotechnical lens to help model, understand, and predict the factors that lead to high-quality software systems. The field emerged in tandem with software engineering, with the first academic conferences and journals appearing in the mid-1970s. Since the beginning, there has been a symbiotic relationship between software engineering researchers and practitioners [2]; as the science of software development, SE research provides a foundation for evidence-based practice in software engineering, and researchers work closely with industry to pioneer innovations in tools, techniques, and methodologies. Of importance to our work, there has been growing interest within the computational science and engineering (CSE) community in engaging with SE research to help address challenges in scientific software development [3].

There has been little interaction between US DOE laboratories and the SE research community up until very recently, which has stymied the flow of innovation. Whereas the national labs have had a presence in HPC-related conferences and journals since the very beginning, we are virtually unknown in the realm of software engineering. Case in point, searching IEEE Xplore for publications by lab-affiliated authors at the ACM/IEEE International Conference on Software Engineering (ICSE) turns up just 7 articles since the conference was established in 1975 – all of them in recent years. Likewise, IEEE Transactions on Software Engineering, one of the premier journals for SE research, has accepted exactly three articles from national lab authors in its history: one in 1984 and two in 1987. Current numbers of CSE participation in SE research are not readily available, but a 2002 review of the publication records of six prominent SE journals found that only 1% of articles concerned computing in science and engineering [4].

Meanwhile, because the CSE community has kept its distance, we are not yet conversant in software engineering literature and thus have yet to reap the rewards of evidence-based practice in software development. By evidence-based practice, we mean integrating current best evidence from research with practical experience and human values to improve decision-making related to software development and maintenance [5]. The lack of familiarity with the science has had a direct impact on productivity and quality in CSE software development. As noted by the authors in a previous article [6], SE researchers have made great progress in understanding the factors that influence developer productivity with one recent meta-analysis identifying thirty five such factors [7]. Each of those factors has been the subject of systematic, scientific investigation by SE researchers, and insights from those studies could inform CSE development practices. There may be situations where the applicability of studies is limited insofar as scientific software developers are underrepresented in the literature, but this is once again the result of a lack of engagement – something which the CSE community can and should remedy.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DENA0003525. SAND2021-14807 C.

II. Opportunity

We believe that SE research has tremendous potential to benefit scientific computing, and we have identified three key ways in which the CSE community could leverage that potential: (1) by translating existing SE research techniques and findings to the CSE domain, (2) by partnering with SE researchers to conduct novel research in established topic areas, and (3) by taking a leading role in defining SE research in emerging domains of interest to the DOE.

First, SE research provides a rich toolkit of approaches for studying both software (*e.g.,* program analysis and repository data mining) and those who create and use that software (*e.g.,* case studies and controlled experiments). These techniques have been successfully applied to many of the notional topics in the ASCR workshop call including the impact of static and dynamic tools on software development, models of developer communities, and strategies for addressing software reliability. A promising course of action would be to build upon existing SE literature to address the same concerns in scientific software development. As we mentioned previously, the relative scarcity of published studies targeting scientific software teams could limit the transferability of findings. We see this as an opportunity to conduct translational research, such as replication studies, to fill that gap. Doing so would promote a deeper understanding of the best available evidence on the science of software development, spur closer collaboration with academia, and raise the visibility of the CSE community in SE research.

Second, there are areas in SE research which are well-established but where we know scientific software developers face issues that do not neatly align with those in conventional software development. As noted by Carver et al., scientific software teams frequently encounter issues in areas such as verification and validation, performance engineering, and the use of common development tools (like IDEs) [8]. Likewise, we see broader needs that SE research could help satisfy, such as in developing strategies for managing scientific software evolution, development methodologies tailored to scientific software development, models for scientific software security, and data-driven approaches to scientific software ecosystem sustainability. Through partnership with SE researchers – giving them the opportunity to apply their knowledge to a new domain – we could real progress in these fronts.

Finally, looking towards the horizon of scientific computing, we anticipate increasing heterogeneity and diversity in computing architectures (*e.g.,* neuromorphic, optical, quantum, etc.) as well as novel applications and use cases for HPC scientific computing (*e.g.,* machine learning, hybrid edge computing). The SE community has interests in all of these subjects, but research is still nascent, such as in SE for machine learning [9] and quantum computing [10]. The national labs are well-positioned to take a leadership role in defining evidence-based best practices through SE research in these emerging areas.

III. Why Now?

The growing diversity of computational science teams has drawn software engineering researchers and practitioners into the fold. This is of immediate practical benefit as these professionals are bringing with them much-needed insights from other software domains to scientific computing. Case in point, at Sandia National Laboratories we have made some strides towards bringing software science into the conversation with the establishment of the Department of Software Engineering and Research within the Center for Computing Research in late 2018 [11], [12]. As we turn our attention to the future, we see an opportunity to re-imagine the relationship between computational science and software engineering research; by expanding the workforce to include more software scientists, we gain new, native connections to the field. It is our sincere hope that this marks the beginning of an enduring partnership, one in which software engineering empowers computational science.

References

[1] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–51, 2018.

[2] L. J. Osterweil, C. Ghezzi, J. Kramer, and A. L. Wolf, "Determining the impact of software engineering research on practice," *Computer*, vol. 41, no. 3, pp. 39–49, 2008.

[3] M. Heroux, "Research software science: A scientific approach to understanding and improving how we develop and use software for research," published 2019-09-25.

[4] R. L. Glass, I. Vessey, and V. Ramesh, "Research in software engineering: an analysis of the literature," *Information and Software technology*, vol. 44, no. 8, pp. 491–506, 2002.

[5] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings. 26th International Conference on Software Engineering*. IEEE, 2004, pp. 273–281.

[6] R. Milewicz, "Towards evidence-based practice in scientific software development," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[7] E. Oliveira, T. Conte, M. Cristo, and N. Valentim, "Influence factors in software productivity—a tertiary literature review," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 11n12, pp. 1795–1810, 2018.

[8] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *29th International Conference on Software Engineering (ICSE'07)*. Ieee, 2007, pp. 550–559.

[9] F. Khomh, B. Adams, J. Cheng, M. Fokaefs, and G. Antoniol, "Software engineering for machine-learning applications: The road ahead," *IEEE Software*, vol. 35, no. 5, pp. 81–84, 2018.

[10] M. Piattini, G. Peterssen, and R. Pérez-Castillo, "Quantum computing: A new software engineering golden age," *ACM SIGSOFT Software Engineering Notes*, vol. 45, no. 3, pp. 12–14, 2021.

[11] R. Milewicz, J. Willenbring, and D. Vigil, "Research, develop, deploy: Building a full spectrum software engineering and research department," *Research Software Engineers in HPC (RSE-HPC-2020)*, 2020.

[12] ——, "Moving forward together: How a software engineering department can impact developer productivity in a research organization," *The 2020 Collegeville Workshop on Scientific Software*, 2020.

Emphasis on Software Maintenance for Scientific Software

Ross Miller (rgmiller@ornl.gov), Joshua Brown (brownjs@ornl.gov), Sarp Oral (oralhs@ornl.gov), Olga Kuchar (kucharoa@ornl.gov), and Mallikarjun Shankar (shankarm@ornl.gov)

In this paper, we, as the staff at the Oak Ridge Leadership Computing Facility (OLCF), argue that lack of sustained and proper maintenance of most scientific software impedes the scientific discovery process across the DOE supercomputing, especially given the on-going Cambrian explosion of hardware and software technologies, such as accelerators and AI. We propose a multi-pronged solution, including technical and policy changes.

We have a combined experience of close to three decades of developing scientific production software on large-scale supercomputers and associated systems. We have observed that the lack of proper and continuing maintenance of scientific software is a major problem, especially on the large-scale computational systems. There are multiple well-known applications that run on the supercomputers that are clearly being maintained (e.g., MPI, Lustre, HPSS), while there is also an enormous amount of software that is poorly maintained.

Scientific software development is characterized by continual transition. In academia, the transitionary nature of the key development staff (e.g., graduate students or postdocs) is the norm. The problem also remains a challenge for career software developers at DOE laboratories as they must frequently juggle multiple projects in parallel. Loss of one individual from a small development team can introduce large knowledge gaps in a project. Also, the technical gap is far larger in scientific software compared to commercial software. Commercial software has the financial ability and incentive to maintain quality software once it becomes a revenue generating product. In contrast, the financial strategy behind most scientific software is feature-based. Once the scientific software package has been created and a paper has been published, the funding runs out unless a new feature is proposed.

As a technical solution we argue that it should be easier for developers to maintain their software. One opportunity we suggest is to catalog well-maintained scientific software. This catalog should be extensive and include middleware, support libraries, and development and debugging tools. Ideally, the catalog would capture software packages from DOE laboratories and would include metadata such as maintainers, origins, targets, and expected lifetime of a software artifact.

The Spack project [2] is a good start at such a catalog. It is predominantly aimed at HPC and used widely across multiple DOE laboratories. Where it falls short is in the maintenance of individual packages. Most packages in Spack list no maintainer and there's no good way to know how much maintenance any given package receives. Spack also keeps no information about a package's expected lifetime.

A similar idea is embodied in the OLCF's Center for Accelerated Application Readiness (CAAR) [3]. CAAR is only for a few leadership-class applications, but the important thing is that it is proactive: developers begin working on the applications just as soon as the architecture for a new supercomputer is chosen. We propose expanding this kind of proactive work to include more applications, with richer metadata captured, as explained above.

We also argue there is opportunity to improve software testing and maintenance tools. The main problem here is the difficulty of using the existing tools. Continuous integration (CI) and regression testing provide a good illustration of this problem. The OLCF maintains GitLab instances for its developers and has several OpenShift clusters. It is possible to configure a repo in GitLab to automatically run tests in an OpenShift pod whenever a change is committed. We have tested this path and observed that configuring the repo and OpenShift pods took a full day of work and required the assistance of the OpenShift admins. If we want developers to use tools like this, this kind of setup hassle needs to be drastically reduced. What needs to be done to make various tools easier to use is probably specific to each tool. This may be a good topic for discussion at the upcoming workshop.

As a policy solution, we argue there needs to be a sustained commitment to scientific software maintenance. This commitment needs to be organizational and needs to be in the form of both providing sustained funds to work on software maintenance tasks and incentives to do so. We note that there are a variety of awards and other forms of recognition for producing new and/or innovative software, but not much for on-going maintenance. Even at the level of individual annual performance reviews, producing new software is valued more than regular tasks like fixing bugs or validating the software on a new platform. We believe this is the sort of organizational attitude that needs to change.

Funding sources also need to value more the importance of sustained investment in maintaining the existing scientific software. Studies examining code replication and dependency deprecation are needed to help quantify the direct and indirect cost and motivate investment. One realization of an effective policy change would require proposals of any significant development effort to include a section on sustained code maintenance and distribution, as proposed by R. Lehe *et al.* [1]. What constitutes as a significant development effort can be a topic for debate at the workshop.

In summary, our proposed multi-pronged solution consists of:

- Maintain a catalog of known well-maintained software
- Expand ORNL's program for proactively porting software to upcoming hardware
- Make it easier to use existing software test & maintenance tools
- Change the attitudes of the funding sources towards software maintenance

References

[1] R. Lehe et al., "Embracing modern software tools and user-friendly practices, when distributing scientific codes," pp. 9–11.
[2] "Spack - Spack." https://spack.io/ (accessed Nov. 22, 2021).
[3] "Center for Accelerated Application Readiness (CAAR) – Oak Ridge Leadership Computing Facility." https://www.olcf.ornl.gov/caar/ (accessed Nov. 22, 2021).

Working in Harmony: Towards Integrating RSEs into Multi-Disciplinary CSE Teams

Miranda Mundt
Department of Software Engineering and Research
Sandia National Laboratories
Albuquerque, NM
mmundt@sandia.gov

Reed Milewicz
Department of Software Engineering and Research
Sandia National Laboratories
Albuquerque, NM
rmilewi@sandia.gov

***Abstract*—Within the rapidly diversifying field of computational science and engineering (CSE), research software engineers (RSEs) represent a shift towards the adoption of mainstream software engineering tools and practices into scientific software development. An unresolved challenge is the need to effectively integrate RSEs and their expertise into multi-disciplinary scientific software teams. There has been a long-standing "chasm" between the domains of CSE and software engineering, and the emergence of RSEs as a professional identity within CSE presents an opportunity to finally bridge that divide. For this reason, we argue there is an urgent need for systematic investigation into multi-disciplinary teaming strategies which could promote a more productive relationship between the two fields.**

***Index Terms*—scientific software development, software engineering, research software engineers, multi-disciplinary teaming**

I. Challenge

In the past decade, the computational science and engineering (CSE) community has grown to include more software engineering (SE) practitioners and researchers. This is most visibly evident on the practice front with the emergence of the research software engineer (RSE) as a professional designation [1]. These RSEs bring with them valuable expertise in the science and engineering of software. Software engineering is the "disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software" [2]. Every engineering discipline has counterparts in science, and for software engineering, that is software science (also known as software engineering research): the systematic study of software systems and their development, operation, and maintenance [3]. These twin fields, which first emerged as a response to the "software crisis" of the late 1960s, are united in the pursuit of evidence-based best practices for the development of software.

In this effort, the SE community enjoyed considerable success: many of the tools, techniques, and processes that are employed by developers today are the product of a diligent synthesis of research and practice. We believe that similar progress can be made in scientific software development [4]. What is not yet clear, however, is how to integrate SE expertise into multi-disciplinary computational science teams. Situating RSEs inside projects does not, by itself, guarantee effective teaming or the adoption of SE best practices; if we truly mean to include and enfranchise RSEs within the CSE community, we must address this challenge.

There has long been a divide – which notable commentators have described as a "chasm" [5]–[7] – between CSE and software engineering. CSE researcher-developers have historically maintained an independent community of practice outside of mainstream software engineering [8] – one with its own terms, techniques, values, and norms. We have seen how these differences can spur conflict: researchers feeling that software engineers fail to appreciate the domain-specific nuances of scientific computing, and software engineers feeling that researchers only value quality and craftsmanship insofar as it advances the science.

This is a common problem in multi-disciplinary team contexts. To quote one scholar on multi-disciplinary teaming, "a multi-disciplinary team without differences is a contradiction in terms" [9]. Within multi-disciplinary software development specifically, Burnell et al. has noted how culture differences, conflicts in working patterns, and a lack of understanding of other disciplines can impede communication and effective teaming [10]. While it is imperative that the CSE community adopt and adapt SE best practices to meet the ever-growing demand for scientific computing, this can only succeed if we resolve those communication barriers. There is an urgent need for systematic research in this space to demonstrate effective teaming strategies that can bridge the divide.

II. Why now?

As we mentioned previously, research institutions in recent years have moved to bring RSEs into the fold. The creation of both RSE groups at national labs and universities (*e.g.,* [11], [12]) as well as national associations promoting the interests of those RSEs [13]–[16] have been promising developments. Whereas before SE was viewed as something separate from and outside of CSE, we can now point to RSEs working on the frontlines of scientific software development. Having

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DENA0003525. SAND2021-14806 C.

researchers and RSEs in the same room has been a necessary first step towards reconciliation.

Likewise, we have seen a revaluation of SE concepts and practices in science and mathematics. As noted in a 2018 SIAM report by Rude et al., software itself is the foundation for collaboration and discovery in CSE, and this makes software engineering "central to any effort to increase CSE software productivity" [17]. According to the report, not only must scientists and mathematicians become more conversant in software engineering concepts and techniques, CSE education and training should emphasize cross-disciplinary communication and collaboration so that researchers can work more effectively with software engineers and others.

Software has become an invaluable tool in the pursuit of scientific research, and we are finally at a stage where we can have a productive dialogue on the roles of software engineering and software engineers in the development of that software.

III. Opportunity

Incorporating SE into CSE through better teaming is, at its core, a research challenge. How do we create robust research teams that leverage RSEs and their expertise? How should professionals from different computing backgrounds interact? How do we evaluate the quality of their teaming? Can we provide empirical evidence for the effectiveness of certain multi-disciplinary teaming strategies? There is a clear need for experience reports and case studies addressing these questions to provide an evidentiary basis for further meta-analyses. Within the Department of Software Engineering and Research at Sandia National Laboratories, we have found that embedding RSEs into research efforts has resulted in better teaming, productivity, and quality in software [18], and this dovetails with similar reports elsewhere (*e.g.,* [19]).

At the present, a chief concern for the authors is the lack of a well-defined framework for recreating these successes. Reproducibility is the foundation of all science, and validated models for how multi-disciplinary scientific software teams can work together would be integral to new teams replicating the successes of others. Research in this space could provide a better understanding of team composition and consensus-building strategies between researchers and RSEs. This would build upon the existing body of scholarship into cross-domain collaborative software development teams, such as in the realm of video game development [20], and we hope to see similar results in the CSE domain.

Moreover, we believe that addressing this RSE teaming challenge supports the broader investigation into the science of scientific software development. There is a panoply of as-yet unresolved questions in software science that have been held back by limited access to scientific software teams. Maintaining a strong SE presence within CSE will facilitate software engineering research into tools, techniques, and methodologies.

References

[1] R. Baxter, N. C. Hong, D. Gorissen, J. Hetherington, and I. Todorov, "The research software engineer," in *Digital Research Conference, Oxford*, 2012, pp. 1–3.

[2] W. S. Humphrey, "The software engineering process: definition and scope," in *Proceedings of the 4th international software process workshop on Representing and enacting the software process*, 1988, pp. 82–83.

[3] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Transactions on Software Engineering*, no. 2, pp. 155–165, 1983.

[4] R. Milewicz, "Towards evidence-based practice in scientific software development." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[5] D. F. Kelly, "A software chasm: Software engineering and scientific computing," *IEEE software*, vol. 24, no. 6, pp. 120–119, 2007.

[6] S. Faulk, E. Loh, M. L. Van De Vanter, S. Squires, and L. G. Votta, "Scientific computing's productivity gridlock: How software engineering can help," *Computing in science & engineering*, vol. 11, no. 6, pp. 30–39, 2009.

[7] T. Storer, "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–32, 2017.

[8] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. Ieee, 2009, pp. 1–8.

[9] J. Øvretveit, "Team decision-making," *Journal of Interprofessional care*, vol. 9, no. 1, pp. 41–51, 1995.

[10] L. J. Burnell, J. W. Priest, and J. Durrett, "Teaching distributed multi-disciplinary software development," *IEEE software*, vol. 19, no. 5, pp. 86–93, 2002.

[11] R. Milewicz, J. Willenbring, and D. Vigil, "Moving forward together: How a software engineering department can impact developer productivity in a research organization," *The 2020 Collegeville Workshop on Scientific Software*, 2020.

[12] ——, "Research, develop, deploy: Building a full spectrum software engineering and research department," *Research Software Engineers in HPC (RSE-HPC-2020)*, 2020.

[13] "Society of research software engineering," https://society-rse.org/, accessed: 2021-07-29.

[14] "Us-rse," https://us-rse.org/), accessed: 2021-07-29.

[15] "De-rse," https://www.de-rse.org, accessed: 2021-07-29.

[16] "Nl-rse," https://nl-rse.org/, accessed: 2021-07-29.

[17] U. Rude, K. Willcox, L. C. McInnes, and H. D. Sterck, "Research and education in computational science and engineering," *Siam Review*, vol. 60, no. 3, pp. 707–754, 2018.

[18] M. R. Mundt, A. L. Levine, and J. D. Siirola, "Overcoming productivity plateaus: A story of automation tools and developer productivity." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[19] J. Cohen, D. S. Katz, M. Barker, N. P. C. Hong, R. Haines, and C. Jay, "The four pillars of research software engineering," *IEEE Software*, 2020.

[20] G. Freeman and N. J. McNeese, "Exploring indie game development: Team practices and social experiences in a creativity-centric technology community," *Computer Supported Cooperative Work (CSCW)*, vol. 28, no. 3, pp. 723–748, 2019.

Introducing RSE As a Practice in CS Courses

PRATIK NAYAK, TERRY COJEAN, ADITYA KASHI, and TOBIAS RIBIZEL, Karlsruhe Institute of Technology, Germany
HARTWIG ANZT, Karlsruhe Institute of Technology, Germany and Innovative Computing Laboratory, UTK, USA

Production-level research software has become essential to fulfill the goals of sustainability, productivity and (performance) portability. It is therefore necessary to develop a culture of Research Software Engineering (RSE) in academic courses. To tackle the skill discrepancy on this front, we suggest including key RSE practices such as code version control, automated testing, continuous integration, and collaborative peer reviews as a part of Computer Science (CS) and HPC course curriculum. We believe that this can greatly benefit the students' careers and skills, enhance their interests in CS and HPC courses, increase RSE outreach into the broad community, and streamline the CS teaching process.

Additional Key Words and Phrases: RSE outreach, sustainable software practice, productive software engineering

1 INTRODUCTION

The practice of software development has progressively followed a trend of relying more on tools and techniques for improving software sustainability and development productivity in order to achieve new scientific breakthroughs and better cope with the rising complexity of reusable, portable, exascale-capable software. While the use of these practices pays off over the lifetime of a project, the initial cost of training or "onboarding" new researchers becomes larger. Thus, there is a need for computational research scientists who are not only experts in their domain, but also familiar with modern software development practices and tools.

We believe that instead of theoretically introducing students to sustainable software practices and techniques, our community should strive to integrate these concerns naturally into most coursework, at least on a graduate level. We also present in this paper a concrete example of a graduate-level Numerical Linear Algebra course that adopts these techniques [Nayak et al. 2021].

2 TOOLS AND TECHNIQUES FOR SUSTAINABLE AND PRODUCTIVE RSE

The following set of tools, techniques, and skills have proven unavoidable for sustainable and productive research software engineering.

Version Control Systems are a popular way to manage code bases. The basic idea of these tools is to create code snapshots at different times, which is useful to help with the development process by providing the developers with a complete history of the codebase. This allows manipulating the code-base in new ways such as rolling back in time.

Automated Testing is the practice of utilizing a framework to ease writing of tests, such that the code functionality is systematically compared against reference results, including convenient ways to automatically change parameter settings. These testing frameworks are more applicable at the unit or integration level, whereas end-to-end testing often requires human interaction.

Containers are a way to encapsulate all software dependencies needed to accomplish a task, such as building and running application software. While traditionally not used on HPC platforms, they have seen growing adoption due to the ease of configuration and use, user-level security options, and the ability to run in GPU environments. Hence, they are becoming essential for testing complex research software inside a controlled, reproducible environment.

Continuous Integration is a natural evolution of version control systems and automated testing. This is the process of running the testing frameworks at every new developer commit, under many environments in a continuous and automatic fashion. Therefore, writing CI test scripts is writing the process of pulling, compiling, installing, and testing the code under different environmental and architectural settings.

Collaborative Tools support the version control systems by including extra tools for collaboration. This includes the concepts of pull/merge request, which allows a developer/user to request to add their code into the main code base, allowing for their peers to provide feedback on the quality, sustainability, and need for this addition.

Contributing to a Complex Codebase is an important skill that students should spend more time learning rather than building their own code from scratch for every course [Spinellis 2021]. Developers often learn more from contributing to an existing code, understanding its design, architecture, features, and interacting with its existing community.

3 LEVERAGING THESE CONCEPTS IN COURSE WORK

Integrating research software engineering, better software engineering, and productivity into courses that are not specifically a tutorial on these topics may seem like a daunting task at first. It requires re-designing all or a part of the course structure, handout, and the grading system to integrate these strategies. There is also an initial setup cost, although a more widespread adoption could help reduce this cost thanks to reusing parts of the infrastructure software and material. In this section, we show that it is beneficial to do so, along with the different ways to leverage these skills depending on the type of course, and showcase the benefits to both the students and the instructors.

3.1 Why and How

We claim that despite the initial setup cost, bringing automation, testing, a framework, and other techniques allows to reduce efforts and ease the sustainability of the course in the long term. By using standard collaborative techniques and practices, students are able

Authors' addresses: Pratik Nayak, pratik.nayak@kit.edu; Terry Cojean, terry.cojean@kit.edu; Aditya Kashi, aditya.kashi@kit.edu; Tobias Ribizel, ribizel@kit.edu, Karlsruhe Institute of Technology, Germany; Hartwig Anzt, Karlsruhe Institute of Technology, Germany and Innovative Computing Laboratory, UTK, Knoxville, USA, hanzt@icl.utk.edu.

to naturally engage more with the course and learn applied collaborative programming. The use of these concepts in courses allows students to focus on the important aspects of the course. Finally, this would make both scientific computing and good software practice more widespread and attractive while creating and propagating a reputation of quality to scientific computing.

In detail, we only showcase a few examples of how to integrate these practices into courses. (1) Systematically checking input/output conformity of student algorithmic implementations is a strategy that can be used with most applied courses and is already in use in some computer science departments. To push this further, it can be combined with (2) an exercise skeleton or a framework that supports unit tests and possibly leverages CI [Nayak et al. 2021], (3) a benchmarking and visualization framework enhance understanding and ease analysis [Lasserre et al. 2021], and (4) collaborative peer review in the exercise process so that students can learn from one another. More advanced techniques can be (5) relying on git workflows to track code development or for experimental design and reproducibility [Stanisic et al. 2015], (6) allow HPC cluster access to students by relying on user-level container environments, and (7) making the coursework take place as a part of, or by reusing pieces of, a more complex software.

3.2 In Practice: The NLA4HPC Course

At the Department of Computer Science at Karlsruhe Institute of Technology, we offer a course on Numerical Linear Algebra for High-Performance Computing (NLA4HPC) which teaches numerical linear algebra (NLA) and parallel programming concepts on different architectures (GPUs, multi-core CPUs and multi-node clusters). The course consists of lectures complemented with practical coding exercises of NLA algorithms.

Our main objectives were to:

- Provide the students with a common framework, allowing them to focus on the algorithm and kernel implementation rather than build and compilation issues.
- Familiarize the students with established unit testing and benchmarking frameworks and encourage their usage.
- Leverage the available git forges such as GitHub and GitLab to encourage the students to peer review each others' code.
- Introduce students to the Ginkgo library [Anzt et al. 2020] and encourage them to contribute to research software.

The framework we provide consists of a complete build system using CMake integrated with unit tests to be written in googletest, optionally allowing students to benchmark their codes with google benchmark. Additionally, we provided a continuous integration workflow to automatically run the students' codes for each homework on our machines based on Gitlab CI. A similar public example of the framework is available on GitHub [Nayak 2021].

In Table 1, we showcase the results of the anonymous survey we conducted after the completion of the lecture. We posed the students with statements and asked them to rate how much they agreed/disagreed with the statements and explain their reasoning.

All students said that they would like to see this type of framework in other courses. Many students found the code review process time-consuming but agreed that the review process helped them improve their coding skills. We also observed a marked improvement in the code quality as the exercises progressed, with students incorporating suggestions from previous reviews in their new assignments.

Table 1. Student feedback (1-Strongly Agree to 5-Strongly Disagree)

Statement	**Avg Rating (1–5)**
It was easy to use the framework	1.8
The exercises instructions and documentation was clear	1.5
It was easy to compile and run the code as provided	2.6
The code review from my peer was useful	1.6
The reviewing process was easy	3.3
I would like to see this type of framework adopted by other courses	1

4 CHALLENGES

We mention some challenges or disadvantages of this approach here, though we maintain that the benefits described above far outweigh these. The initial setup cost needs to be paid (including familiarizing the instructors with the tools), the students need to be taught the use of these tools in addition to the subject matter, and additional computing resources are needed (e.g. compute time on the cloud/computing cluster especially when working with new architectures). When using them, peer reviews need to be integrated into the grading process as well.

5 CONCLUSION

In this paper, we explored the need to incorporate software sustainability, productivity concepts, and collaboration into coursework. We elaborated on some of the techniques needed to implement this and discussed some of the practical aspects involved. Finally, we showcased a concrete example through our NLA4HPC course and elaborated on the challenges we faced in adopting this process.

REFERENCES

Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. 2020. Ginkgo: A High Performance Numerical Linear Algebra Library. *Journal of Open Source Software* 5, 52 (Aug. 2020), 2260. https://doi.org/10.21105/joss.02260

Alice Lasserre, Raymond Namyst, and Pierre-André Wacrenier. 2021. EasyPAP: A framework for learning parallel programming. *J. Parallel and Distrib. Comput.* 158 (2021), 94–114. https://doi.org/10.1016/j.jpdc.2021.07.018

Pratik Nayak. 2021. NLA4HPC Exercises Framework. https://github.com/pratikvn/nla4hpc-Exercises-Framework

Pratik Nayak, Fritz Göbel, and Hartwig Anzt. 2021. A Collaborative Peer Review Process for Grading Coding Assignments. In *Computational Science – ICCS 2021*, Maciej Paszynski, Dieter Kranzlmüller, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer International Publishing, Cham, 654–660.

Diomidis Spinellis. 2021. Why Computing Students Should Contribute to Open Source Software Projects. *Commun. ACM* 64, 7 (jun 2021), 36–38. https://doi.org/10.1145/3437254

Luka Stanisic, Arnaud Legrand, and Vincent Danjean. 2015. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 61–70. https://doi.org/10.1145/2723872.2723881

Joint Analysis of Development Processes and Code Quality for Better Software

Boyana Norris, University of Oregon, norris@cs.uoregon.edu

1 Challenge

Software development is a high-expertise, high-effort activity that occurs in industry, science, engineering, and many other areas. While many practices and tools that increase both developer productivity and software quality are widely available, many HPC projects have been hesitant or slow to adopt them. There are very good reasons for this, for example, tensions between the primary objective (producing a scientific result) and the additional effort required to develop high-quality software. Moreover, very little hard evidence is available to justify spending the time and effort to adopt a new "best practice" that has not yet been shown to be effective in an HPC setting.

2 Opportunity

We believe that defining a rich set of metrics quantifying both development processes and characteristics of the software itself will enable lead to positive change in HPC software development, especially when accompanied by a number of compelling use cases illustrating their application in real HPC projects.

First, the near-universal use of revision control systems and associated services (e.g., issue trackers) enables the collection of rich metadata that can be used to study and improve development processes. To that end, we are building a comprehensive analysis framework that enables data mining of development metadata (git metadata, issues, mailing list messages). The goal is to provide efficient and easy-to-use tools along with a large number of case studies to enable developers to mine their own data.

Second, in the last 30 years, a vast amount of knowledge has been amassed that quantifies software quality. However, apart from the light use of basic linters, the HPC community has not adopted quality measurement or goals as part of software development processes. At the same time, significant automation of code quality checks is now much more achievable than it has ever been, thanks to open-source frameworks, such as LLVM.

Finally, this is not just an opportunity to automate repository data mining and code quality checks, but to define and use much more comprehensive analyses that track development over time and can be used to guide development decisions (e.g., suggest refactoring based on anomaly detection in key code components) or directly assist in development (e.g., we are working on a pull request assistant that identifies opportunities for improving testing and documentation).

Our recent IDEAS Best Practices webinar goes into more detail on these opportunities and current approach [3]; we also show some initial code quality metrics automation in [4].

3 Timeliness or maturity

Plurasight released an e-book describing 20 development patterns based on observations of thousands of developers in industry. This is meant to be a field guide that can be used "to get a better feel for how the team works, and to recognize achievement, spot bottlenecks, and debug your development process with data." [1] We have started implementing automated detection of these patterns using git metadata and source code analysis and believe that monitoring trends over time can help make informed decisions on different aspects of development, including better human resource allocation, detecting undesirable trends, identifying gaps in documentation and testing, guiding refactoring decisions, and informing developer's implementation choices.

The ISO/IEC Information Technology Task Force (ITTF) recently released the ISO/IEC 5055:2021 Standard "Information technology — Software measurement — Software quality measurement — Automated source code quality measures" in March of 2021 [2]. It includes many well-defined software quality metrics that, in principle, can be collected automatically. To quote the introduction, "Quantitative evaluation of software product quality is key to ensuring correct performance. This assurance requires rigorous measures of software product quality characteristics." To our knowledge, almost none of the metrics included in this standard are regularly tracked in HPC software, even though many of them are just as relevant to HPC development as any other type of software.

Most software quality metrics require compiler support. In recent years, the LLVM framework has grown and is now firmly established as an open-source, comprehensive compiler suite that supports many high-level languages (including Fortran). More important, it provides a growing number of intermediate language-based tools. Finally, through the MLIR it enables more flexible language extensions or domain-specific language definitions. Given all this, it is now possible to develop robust source analysis tools with a fraction of the effort (and expertise) that used to be required less than a decade ago.

In conclusion, at this workshop, we hope to discuss the needs and use cases for data-driven methodologies for the combined analysis of development metadata (for understanding development patterns) and software quality over time. We hope to share some of our findings and ideas to prime the discussion.

References

[1] 20 patterns to watch for in your team. https://www.pluralsight.com/content/dam/pluralsight2/landing-pages/offers/flow/pdf/Pluralsight_20Patterns_ebook.pdf, 2019.

[2] ISO/IEC 5055:2021 Information technology — Software measurement — Software quality measurement — Automated source code quality measures. https://www.iso.org/standard/80623.html, 2021.

[3] Mining development data to understand and improve software engineering processes in HPC projects. http://ideas-productivity.org/events/hpc-best-practices-webinars/#2021-webinars, 2021.

[4] Bosco Ndemeye, Shahid Hussain, and Boyana Norris. Threshold-based analysis of the code quality of high performance computing software packages. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021. To appear; preprint: https://hpcl.cs.uoregon.edu/pubs/QRS21.pdf.

Improving Software Sustainability Through Adoption and Democratization of User Research Within Scientific Organizations

Drew Paine, Lavanya Ramakrishnan, Dan Gunter, Sarah Poon
Scientific Data Division, Lawrence Berkeley National Laboratory
[pained, lramakrishnan, dkgunter, sspoon]@lbl.gov

Software is integral to scientific research, but requires ongoing usability and user experience (UX) improvements in order to be a reliable, sustainable resource for user communities. There is a substantial focus on improving the usability and reliability of commercial software through user experience efforts. User experience research, in particular, helps ground product designs in realistic user needs through qualitative methods such as interviews, observations, and usability evaluations [10]. In recent years these approaches have seen some success in scientific contexts [5,7]. *Even still, for DOE communities to successfully produce more usable, sustainable software products there remains an urgent need to more widely adopt and adapt user experience approaches for scientific software projects. Overall, the DOE community can benefit from user experience while contributing a unique perspective to the multiple disciplines that have studied the practices and cultures of scientific software development.*

Challenge: Software engineering research underscores that the practices and cultures of scientific software development work are distinct from commercial software engineering due to domain specific needs and challenges successfully fostering collaboration between scientists and engineers [3,8]. Computer Supported Cooperative Work (CSCW) studies demonstrate the need to reframe our perceptions to understand software as a key material developed and used by stakeholders in the conduct of the scientific method [1,6] while also pointing out the cultural factors influencing the sharing of scientific software [2]. In particular, CSCW research illustrates how software is a vital research and design ***material*** for scientists to express their methods, capture and interrogate data, and convey their evolving knowledge [4]. This work is a uniquely human effort that is more than just an outgrowth of scientific work, and a perspective shift is essential to recognizing scientific software cultures as practice-driven. This scholarship emphasizes that effectively producing scientific software requires continual dialogue among people, instruments, data, practices, and organizations through interactive, often at least partially human-in-the-loop computing workflows.

Sustainability challenges emerge due to the impacts of constantly evolving computing ecosystems, project timelines, and varying organizational endeavors influencing development work. The increasing diversity of computational environments (PCs, clusters, cloud, HPC) complicates scientific software development by introducing new technology stacks and practices that stakeholders may have minimal experience leveraging in their efforts. In addition, the evolving range of software products essential to data work have varying degrees of maturity and stability — from scripts encapsulating analysis decisions to the data processing pipelines of projects and general purpose open source libraries (e.g. tools from the scientific Python stack such as matplotlib and Jupyter Notebooks, or HPC oriented tools like VisTrails or PETSc) — which introduces another layer of complexity for the communities building scientific software. Centering software and data work as a material element of development practices contends that sustainability of scientific software (and reproducibility of research) is not just a matter of having one version of a software product to re-run. It is just as importantly about recognizing the human practices and processes of work while capturing and communicating the nuanced histories of myriad artifacts. This perspective on software and data work is increasingly not only important but also essential as experimental, observational, and simulation data and processing meld together in complex new assemblages that span multiple teams, national labs, and so on.

Opportunity: Sustainable scientific software not only needs stable funding sources but transparency and a wider focus on user needs in the processes of its production. Open and accountable collaboration among developers and users, increased recognition of the diversity of types of software shaping the research landscape, and an emphasis on improving the overall usability of scientific tools is vital. More widely adopting qualitative user research will help scientific software teams better understand the scientific practices they are designing solutions for, while helping gather fundamental insights into the evolving ways science is done. Adapting commercial user experience approaches to DOE environments can help our software teams better scope the correct design problems and requirements through systematic interviews and observations of research practices, and ensure a broader understanding of this work among engineers and managers working in the team. Furthermore, leveraging evaluative methods to assess the products being created through usability testing can improve usability. Experiences within the community emphasize the need to adapt commercial practices due to the scientific environment's different needs and demands [5]. As part of efforts adopting and adapting these approaches, and continuing to develop new approaches, we should furthermore undertake efforts to democratize user research practices. Democratizing user research will enable engineers and other team members to contribute to the collection of vital data, and help shape designs, while developing more empathy and grasp of scientific work practices.

Timeliness or Maturity: We see that qualitative UX research is commonplace in commercial software development and has seen success among teams building scientific software [5,7]. Increasingly commercial organizations are undertaking efforts to democratize user research to enable a wider range of stakeholders to gather information needed to shape products [9]. Democratization of UX enables user research teams to educate and empower engineering, and other, colleagues to be able to strategically gather valid qualitative data and generate insights themselves. UX researchers in turn help guide their colleagues and ensure the organization systematically obtains the insights necessary to sustain successful products.

References

1. Marisa Leavitt Cohn. 2019. Keeping Software Present: Software as a Timely Object for STS Studies of the Digital. In *digitalSTS: A Field Guide for Science & Technology Studies*, Janet Vertesi, David Ribes, Carl DiSalvo, Yanni Loukissas, Laura Forlano, Daniela K. Rosner, Steven J. Jackson and Hanna Rose Shell (eds.). Princeton University Press, 423–446. Retrieved from http://www.jstor.org/stable/j.ctvc77mp9.31
2. James Howison and James D. Herbsleb. 2011. *Scientific Software Production: Incentives and Collaboration*. ACM, Hangzhou, China. https://doi.org/10.1145/1958824.1958904
3. D. F. Kelly. 2007. A software chasm: Software engineering and scientific computing. *Software, IEEE* 24, 6: 120–119.
4. Drew Paine. 2016. Software and Space: Investigating How a Cosmology Research Group Enacts Infrastructure by Producing Software. University of Washington. Retrieved from https://hdl.handle.net/1773/36979
5. Drew Paine, Devarshi Ghoshal, and Lavanya Ramakrishnan. 2020. Experiences with a Flexible User Research Process to Build Data Change Tools. *Journal of Open Research Software* 8, 1. https://doi.org/10.5334/jors.284
6. Drew Paine and Charlotte P. Lee. 2017. "Who Has Plots?": Contextualizing Scientific Software, Practice, and Visualizations. *Proceedings of the ACM on Human-Computer Interaction* 1, CSCW. https://doi.org/10.1145/3134720
7. Lavanya Ramakrishnan and Daniel Gunter. 2017. *Ten Principles for Creating Usable Software for Science*. https://doi.org/10.1109/eScience.2017.34
8. Judith Segal. 2009. Software Development Cultures and Cooperation Problems: A Field Study of the Early Stages of Development of Software for a Scientific Community. *Computer Supported Cooperative Work* 18, 5: 581–606. https://doi.org/10.1007/s10606-009-9096-9
9. Behzod Sirjani. 2020. Democratization is our Job — Yet Another Studio. Retrieved from https://yetanother.studio/blog/democratization-is-our-job
10. What is UX research? *Interaction Design Foundation*. Retrieved from https://www.interaction-design.org/literature/topics/ux-research

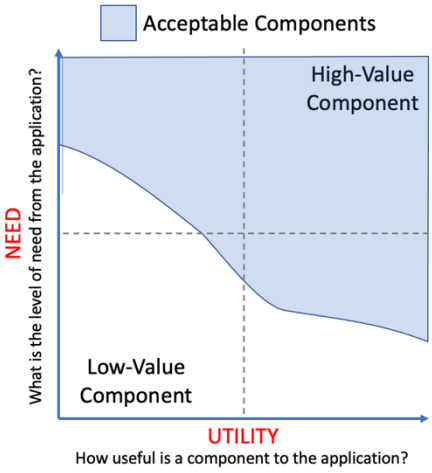
Developing Objective and Quantifiable Processes for Assessing Software Component Usefulness

Roger P. Pawlowski (rppawlo@sandia.gov) and Eric C. Cyr (eccyr@sandia.gov)
Sandia National Laboratories

Modern computational science software applications are built on top of software libraries or components. When successful, libraries can rapidly accelerate development, improve software quality, and enhance developer productivity. This has been demonstrated by successes seen with recent low-level "performance portable" libraries for computing on evolving architectures. However, when components, for reasons of design, fit, or maintenance don't meet the needs of an application their adoption can be expensive and result in reduced developer productivity. Given this high risk/reward tradeoff, we propose research on methodologies and evaluation metrics that are useful both from the perspective of component creation (a "push" model for what has **utility**), and component consumption (a "pull" model for what is **needed**).

A successful methodology will enhance the transparency of what a high-value component must contribute and expose gaps in the software ecosystem that a high-value component could fill. We envision this capability to be useful to three stakeholders i) application teams, ii) component development teams, and iii) funding decision makers. In all cases, decision for the support or adoption of a software component should be made with metrics that are designed to get to the core challenges of software development that reflects the complex attributes of computational science software products. Finally, such metrics applied faithfully, should help address challenges plaguing computational software, for instance: "not invented here syndrome," unbounded software complexity, and limited software support. With the proliferation of new components created under DOE's ECP project, the timing is right to invest in tools that provide guidance on software integration choices.

Our current perspective on this is that the key to a successful methodology is reducing the evaluation of a component to two axes, one of **utility** (push) and one of **need** (pull). The figure on the right shows a schematic that distinguishes the value of a component by one of four quadrants. From an application developer's perspective, they are required to explore how much a component is needed and weigh that against its utility. Again, using performance portability libraries as an example, these libraries encode architectural details, and parallel patterns in their design thus delivering **utility**. From the perspective of **need,** the application is required to run on those platforms to achieve its mission, and the application developer team does not possess the architectural knowledge to deliver on those platforms. Thus, those libraries, like Kokkos or RAJA might be considered a high-value component. However, if the **utility** is low (e.g., use case doesn't quite fit or a limited support model) the application team can investigate alternatives. Further, a component with high utility but low need may represent a potential growth direction (e.g., into optimal design, from a forward simulation code's perspective).



When a component is deemed low value by a particular application, the proposed assessment/metrics still could strongly benefit the component's development. For instance, an application could rank a component as high need but low utility. The low utility score could be due to the code being difficult to understand, especially for new team members. Additionally, the component could be under rapid change or not well supported, causing significant maintenance and integration overhead. Thus, the component developer could use these same metrics to provide guidance on the evolution of the software capability.

The final consumer of these metrics is funding agencies. The metrics, being a measure of both need and capability may provide guidance to software components where the investment would be particularly impactful. We are not advocating the blind use of metrics for funding decisions but having such data could provide justification when making difficult funding decisions or contentious design choices. Again, for areas where need is high, but utility is

relatively low additional investments may provide a high impact pathway to accelerating developer productivity. The metrics are viewed as being helpful for initiating a new product as well. Just because an organization can create a component doesn't necessarily mean that they should create the component. Unexpected overhead from dealing with external releases and user support can add up quite quickly. Thus, being able to evaluate the utility requirements demanded to meet application needs is critical to make efficient investments.

Need	Utility
TEAM KNOWLEDGE: Is there a strong knowledge base of required algorithms within the application team? Does the application require external component experts?	**USABILITY**: Is the component easy to use? Easy to integrate into the application? Are the interfaces clear/easy to understand? Documentation available?
REQUIRED MAINTENANCE: How much time does the application team spend maintaining/updating the component? Significant debugging every update? How often is the component updated?	**ROBUSTNESS**: Is the component robust? Are the algorithms stable? Is there good test coverage to protect against regression?
OPPORTUNITY: Capabilities you don't know you need/want, but when its "easy" it becomes useful.	**CAPABILITY**: Does the component provide what the application really needs? Is it a "close enough" fit?
REQUIRED SUPPORT: How much time is required from the component team? Are they embedded into the application team?	**TEAM**: Is the component team responsive to issues?
CRITICALITY: Is this capability critical to application success? If critical, the application should own this – it's "your thing".	**PERFORMANCE**: Is the component performant in the context of use in the application? Performance may differ by use case.
FOOTPRINT: Is this a fundamental data structure that the entire team needs to learn since it will be encountered all over the code? Can the component be isolated and hidden from the majority of development team?	**CLARITY**: Is the algorithm clear and easy to understand from perspective of both the component code and the application calling code? Can new developers read the code and quickly understand the algorithm? Do the interfaces obscure the algorithm for an application use case?
LIFECYCLE USE: Is the component required to meet short term project deliverables? Could it be used in a product demonstration? Could long term production codes phase this out?	**RISK**: Is the component funded long term? Is the component open source and supported/used by the community? If the developers leave the project, is the application team comfortable taking ownership of component?

While the utility/need axes are useful for quick evaluation, a more detailed approach is required to create the information about the component. In the table above, we suggest important categories useful for evaluating a component's potential in an application. The resulting need and utility metric formulas should account for the categories in the table while also considering organizational priorities. Developers on our applications rate each category, and then the distribution on the need/utility plot can be used to assess the perceived value of the component. This has been useful in breaking down barriers, sometimes our own, that might not be so firmly rooted in the needs of the application as much as affinity to software designs/products. Moreover, we've also used this to strengthen our connection to other software products and provide concrete suggestion for improvement.

Ideally studies should be able to correlate the metrics with successful and unsuccessful component development efforts from past software projects. Further, development of tools could provide predictions of success or warnings for components currently under development – allowing teams to make improvements. The research suggested here should provide data analysis tools for determining true costs of maintaining/integrating a component. Data mining and machine learning could play a critical role here. We envision tools that could skim external and internal project databases such as gitlab and github (not just the component repositories but consumer application repositories as well) for markers that a component is providing value and assessing if it is placing an unreasonable burden on a project. It could measure burdens in terms of performance, platform portability, disruptions due to API changes and software bugs. In many cases developers might be living with burdens, not realizing what an investment a component has created.

LLNL-PROP-829314

Lawrence Livermore National Laboratory

Performance Diagnostics and Debugging at Scale: How is that Simulation Executing?

Principal Investigator Olga Pearce, David Boehme
Institution Lawrence Livermore National Laboratory
Contact olga@llnl.gov, boehme3@llnl.gov

Challenges

Key Advantages of this Project

1. Live window into the simulation execution.
2. Actionable and insightful supplement to developer's mental model of the simulation.
3. Leveraging existing infrastructure and code integration efforts.

The scientific software development cycle includes implementation, testing, and verification of results. When incorrect results are produced, the developer embarks on a debugging journey of constructing a hypothesis and performing further testing to either validate or invalidate this hypothesis. This process is executed whether one is trying to locate and correct a bug (code development), or to gain understanding about the behavior of solutions to the model in the code (code application).

One of the biggest challenges facing the modern computational scientist is diagnostics. Due to their compute-bound nature, scientific codes are notoriously opaque to OS-level performance monitoring tools often used for commercial server workloads. As a result, the developer's toolbox is woefully out of date, with printf and a debugger being the most commonly used approaches. The challenging question we see to address is: ***Can performance tools provide a live window into the simulation execution to enable the developers to understand what is going on?*** We describe possible opportunities in three areas: (1) performance metrics; (2) memory layouts; (3) communication patterns.

Opportunities

Performance Metrics

Simulation developers understand which application-specific performance metrics can indicate trouble (e.g., solver iterations). Currently, codes either write summarized metrics into logs to provide proof that the simulation continues to run but do not provide much detail, or record time-series data for post-mortem analysis that requires significant resources at runtime and may be lost if the application crashes. Would the developers be better able to react to the performance issues as they happen, without having to re-run the code, if they were able to observe continuously streaming performance data? Would they be in a better position to understand the state of the simulation if they could query detailed sets of metrics and how they are changing over time?

Full *in-situ* visualization of performance data mapped onto simulation spaces has been prototyped in the past [1], and is likely of interest where numerical questions arise. Would performance data be easier to correlate back to the source code if the developer were instead given a map of performance metrics to the application call graph at each time step? Would a tool that helps understand the changes to performance of each component of the call graph over time supplement the developer's mental model of the simulation execution, enabling introspection necessary to correct problems?

Memory Layouts

Memory usage metrics, when tied to code regions, can provide insight into application's behavior. Is a code region using more and more memory over time? Perhaps this is a function of mesh refinement -

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

or perhaps a memory leak, which could be easily and efficiently diagnosed with a simple memory usage tracker. Is a code region hitting more and more cache misses? This can hint on fragmentation of key data structures - or divergent table lookups necessitating further understanding.

Communication Patterns

Communication patterns in the simulation can change over time, e.g., due to mesh refinement or patch creation. While tools like Score-P [4] offer MPI traces, setting up the application to work with them and re-running the application separately to merely see the traces can be prohibitive. What if a simple mechanism existed to expose very basic data: how many ranks and I communicating with over time? How much data am I sending? Presented in proper context, this very basic (and easy to collect/store) information can be very telling to the simulation developer.

We believe these questions can be solved by integrating performance engineering techniques early in the development process

Better *in-situ* performance analysis and debugging methods

We believe that *in-situ* performance analysis and debugging methods could become more useful (and potentially mainstream) if they are better integrated into production development tools and frameworks, such as the Ubiquitous Performance Analysis [2] approach, which has already transformed the adoption of performance regression testing for scientific codes. Mapping performance metrics onto the developers' mental model of the simulation, the call graph of the simulation, will facilitate actionable discoveries, enabling improvements to the source code. DOE needs to invest in research in tools for performance debugging so the computational scientists can be more productive in developing the cutting-edge scientific software.

Timeliness

Addressing the above challenges is now more feasible than before and the solutions would be very impactful. DOE scientific codes have come a long way in adopting performance analysis tools, which pave the way to *in-situ* performance analysis. We can build upon existing infrastructure and code integration efforts, including the increasing adoption of performance tools in DOE codes, and ongoing efforts to integrate application-level profiling with system monitoring tools like LDMS [3]. We need to fund research in using performance tools to improve the scientific software cycle – and to ultimately drive scientific progress.

References

[1] Wood C. et al.: (2019) *Projecting Performance Data over Simulation Geometry Using SOSflow and ALPINE*. In: ESPT 2017, ESPT 2018, VPA 2017, VPA 2018. Lecture Notes in Computer Science, vol 11027. Springer, Cham.

[2] Boehme D. et al.: (2021) *Ubiquitous Performance Analysis* In: ISC High Performance 2021. Lecture Notes in Computer Science, vol 12728. Springer, Cham.

[3] Izadpanah, Ramin, et al.: *Production application performance data streaming for system monitoring.* In: ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 4.2 (2019): 1-25.

[4] Knüpfer A. et al.: (2012) *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir*. In: Tools for High Performance Computing 2011. Springer, Berlin, Heidelberg.

White Paper: Improving Assurance of Scientific Software

Authors: Barton Miller (University of Wisconsin, Madison; bart@cs.uwisc.edu)
Sean Peisert (Berkeley Lab; sppeisert@lbl.gov)

Challenge: In 2021, the NSF Cybersecurity Center of Excellence (Trusted CI), conducted a study on the security of software used by several NSF-funded scientific computing and cyberinfrastructure projects with the goal of shedding light on the risks incurred in creating and using scientific software [1]. Our primary finding was that few of the groups we interviewed have the support to implement complete software security programs, as defined by traditional industry and government practices, and all could benefit from additional guidance and funding resources specifically supporting secure software development.

For example, with respect to process and management related issues, we found that in most of the scientific software projects that we evaluated, we did not observe threat modeling being performed, either formally or informally, and thus developers were left unaware as to how to prioritize assurance to mitigate the spectrum of threats. We noticed across the projects a lack of documentation. And, where the documentation was present, it was out-of-date for the version of code that was in production. Another observation was finding code that is not written in modern, type-safe languages. Examples of this include code, particularly older code that is either written in languages that are not type safe (e.g., C, C++), inherently insecure (e.g., PHP), or simply out of date (e.g., Python 2).

In addition, program analysis tools are a critical resource for any software project. These tools can be static, applied to the program's code or repository, or dynamic, where they monitor the program's execution. However, we also noticed that developers had difficulty in using tools that can help improve software assurance, including both *static* and *dynamic analysis*. While we found that several organizations used some form of static or dependency tool, those tools were often not used effectively. For example, tools were run directly on containers, but most tools are not designed for that mode of operation. Or alternatively, some organizations use the tools that are integrated in GitHub, while often these integrated tools do not represent the best functionality available.

Notably, an issue with static analysis tools is that they report a high number of false positives. This is apart from the fact that there are some security issues that they are unable to detect, such as incorrect design, code that incorrectly implements the design, configuration issues, since they are not represented in the code, and complex weaknesses involving multiple software components.

Another issue comes up in testing. Several organizations reported testing was done in an ad hoc way. Few leveraged any kind of automated testing, and even fewer included testing for security rather than functionality. Only one organization reported using a form of fuzz (random) testing.

Opportunity: Describe how the identified challenges may be addressed, whether through new tools and techniques, new technologies, new methodologies, or new groups collaborating in the process

There are many obstacles to improving software assurance in science, including funding levels that directly affect elements such as staffing and training resources. At the same time, it is automation that often supports scaling of particular practices without an increase in resource allocation. However, as we've seen, automation is infrequently being used in scientific software development, and when it is used, is frequently used ineffectively. One simple reason is that these tools are often hard to use.

Ideally we need to make it easier for software developers to use these tools effectively. This does not necessarily mean that we need more advanced tools. Rather, we need tools that are simply more straightforward to just run. For example, static analysis tools are often specific to one or two programming languages. However, what if we gave programmers a single interface that was common to numerous static analysis tools, thereby reducing the barrier to understanding how to run different tools against code written in different languages. Or what if we had simpler pipelines that can enable tools to easily be run against the contents of containers? With respect to testing, even simple *black box fuzzing* can be extremely helpful in identifying software errors [2]. And yet given the complexity of even executing so much scientific software, we need simpler methods of applying fuzz testing to such software.

Timeliness or maturity: Why now? What breakthrough or change makes progress possible now where it wasn't possible previously? What will be the impact of success?

Testing and analysis tools have made enormous strides in recent years, as has the computational abilities of computer systems to leverage these tools at scale. Furthermore, the threats against computer systems, particularly via nation state actors, have reached a boiling point [3]. The time to take all necessary steps to reduce risk is now.

References

[1] A. Adams, K. Avila, E. Heymann, M. Krenz, J. R. Lee, B. P. Miller, and S. Peisert. The State of the Scientific Software World: Findings of the 2021 Trusted CI Software Assurance Annual Challenge Interviews. Sept. 2021. https://hdl.handle.net/2022/26799

[2] B. P. Miller, M. Zhang, and E. R. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, 2020.

[3] S. Peisert. Unsafe at Any Clock Speed: The Insecurity of Computer System Design, Implementation, and Operation. *IEEE Security and Privacy*, 20(1), Jan./Feb. 2022.

Position Paper: **Software for Simulations and Analysis of Complex Engineered Systems**

Slaven Peles <peless@ornl.gov>, ORNL, Ivan Celanovic, Typhoon HIL, Cosmin Petra, LLNL

Challenge: Complexity of computational problems and mathematical algorithms today combined with the complexity of emerging heterogeneous accelerator-based hardware poses monumental software engineering challenges that cannot be addressed without significant new research. Scientific software needs to facilitate *productivity* for users and developers alike, deliver *performance* within domain requirements and be *portable* to different hardware platforms. These three objectives often pull software development in opposing directions.

One class of domain areas that are particularly affected by the lack of adequate scientific software are complex engineered systems. This class of problems arises at the intersection of evolving complex multi-physics energy systems, autonomous and intelligent multilevel control system, and extensive data collected from the system operation. Specific domains include power systems, microgrids, energy efficient buildings, and electrical storage systems, to name a few. Technology gaps in scientific software for those disciplines are such that high-performance computing made only modest inroads leaving domain experts to rely heavily on reduced order modeling to be able to work within limitations of the available software. The lack of adequate modeling software limits the ability of Department of Energy (DOE) applied programs to address major mission-critical challenges such as power grid modernization, digitalization, resilience and decarbonization.

For complex systems, component models are the main modeling currency and system models are assembled using methods such as nodal analysis [1]. This approach was pioneered in SPICE [2] and formalized as bow-tie paradigm in [3]. While the modeling approach is relatively well understood, available software implementations are typically not scalable and not portable. For example, irregular and extremely sparse nature of the models can lead to computational bottlenecks and load imbalance, if the software implementation is not expertly designed. Addressing these scientific software challenges requires broader interdisciplinary research involving domain, applied mathematics, graph theoretical, and computational science in addition to software engineering expertise.

Current state-of-the-art solutions are typically proprietary, closed-source tools with limited parallel computing capabilities and proprietary solvers based on legacy code. These software solutions are extremely inflexible, not portable, hard to interface with other libraries and cannot handle large data sets and complex systems. Component models are often solver specific (e.g. have integration formula embedded) making it difficult to use third-party solvers. This limits the ability to rapidly test and evolve new computational techniques for complex engineered systems and also makes realistic engineering models inaccessible to computational science and mathematics experts. This leaves a huge computational science area underexplored.

Heterogeneous nature of mathematical models of complex engineered systems also makes designing user interfaces challenging. The size and the complexity make it infeasible to manage these models through mathematical equations and require use of hierarchical graphical representation of the governing physics. Most of the user interfaces today are based on either proprietary legacy solutions that are not scalable and limit how much detail is accessible to the user or very general open-source solutions (e.g. Julia) that lack the appropriate tools, models, and language abstractions to make them effective for specific domain purposes. Indeed, the lack of effective bridges between domain and mathematical sciences is heavily impeding progress.

Opportunity: To address technology gaps in computational science for complex engineered systems, there needs to be a concerted effort to develop core modeling libraries, define model interfaces, data interfaces, user interfaces, compilers, modeling languages, and modeling protocols that would provide for easy to use, portable, and scalable computations in this area. Such framework would enable development of lightweight single-purpose portable tools that are interoperable with each other and can be interfaced with different state-of-the-art numerical solvers, data sources, and user interfaces. The framework will be used by mathematicians and computational scientists as a research platform, and as a development platform by different DOE applied programs. Schematic representation of such framework is given in Figure 1.

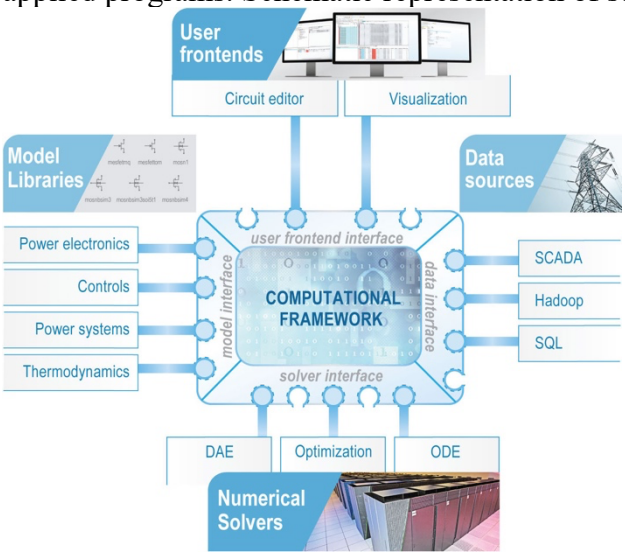


Figure 1: Proposed software platform for development of simulations and analysis tools for complex engineered systems. Plug-in based design allows for creating single-purpose lightweight applications that are interoperable with each other.

However, designing system composition protocols, abstract interfaces, and enabling interoperability is a nontrivial task and requires intense interdisciplinary research that is far beyond typical software engineering effort. We envision this platform being implemented as an open-source software development kit with standardized interfaces that would set industry standards for analysis tools interoperability.

Timeliness: Recent successes in developing portable modeling techniques in ExaGO library within ExaSGD project (power grid optimization at exascale) [4] and distributed memory parallel modeling framework in Xyce circuit simulator [5] at Sandia National Laboratories suggest there is a feasible technical path to address existing gaps in scientific software for complex engineered systems. Improvements in scientific software development practices through projects like IDEAS, as well as advances in compiler technology, most notably new C++ language standards, enabled scientists to tackle these problems more effectively.

The industry, as well as DOE applied programs, rely heavily on reduced order models to be able to work within limitations of the current tools for simulation and analysis of complex engineered systems. Sacrificing fidelity often leads to poor predictions and more expensive hardware testing which in turn causes budget overruns and missed deadlines. Sacrificing portability limits the ability to grow the computational capability in the future. Delivering a comprehensive computational software to support simulations and analysis of complex engineered systems will bring dramatic improvements to different industries, as well as DOE applied programs.

References

[1] Günther, M., U. Feldmann, and J. ter Maten. "Modelling and discretization of circuit problems." Handbook of numerical analysis 13 (2005): 523-659.

[2] Pederson, D. O., and D. O. Nagel. "SPICE (Simulation Program with Integrated Circuit Emphasis)." Sixteenth Midwest Symposium on Circuit Theory, Waterloo, Canada, Paper VI-1. 1973.

[3] Csete, M., and J. Doyle. "Bow ties, metabolism and disease." Trends in Biotechnology 22.9 (2004): 446-450.

[4] ExaSGD – Grid Optimization at Exascale, https://www.exascaleproject.org/research-project/exasgd/.

[5] Verley, J., E. R. Keiter, and H. K. Thornquist. Xyce: Open Source Simulation for Large-Scale Circuits. No. SAND2018-10372C. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

AI/ML and Scientific Software Development and Distribution, Maintenance, and Code Teams

E. W. Bethel, Z. Bai, T. Perciano*, R. Sadre — LBNL
J. Wright, G. Wallace — MIT
S. Shiraiwa, N. Bertelli — PPPL

1 Overview

The increasing interest in Artificial Intelligence (AI) and Machine Learning (ML) technologies in the scientific computing landscape reflects the broad community belief that AI/ML methods have the potential to fundamentally transform the way we do computational and experimental science [8]. These transformational changes will require work across all dimensions, including the increasingly multidisciplinary nature of modern scientific software development [4]. We focus on three specific issues where AI/ML is a part of multidisciplinary science code teams, including the increasingly complicated nature of software engineering and software distributions that include AI/ML methods (§2), the impact on and risk to code investments from changes in a constantly evolving computational landscape (§3), and the increasingly diverse nature of science code teams (§4).

2 Evolution of Solvers to Include AI/ML Methods

Challenge: Increasing complexity of Software Engineering, Distribution. There are numerous efforts underway to make use of AI/ML methods as surrogate models to reduce the computational time of physics codes. Such efforts are made possible by rapid advances in AI/ML technologies as well as by a growing awareness of AI/ML capabilities and their potential benefit to scientific computing and scientific data understanding applications. These new uses pose several challenges for science code teams.

First is a set of decisions around the issue of the scope of the software "release" and software "project". Does the physics code release also need to include the source code and related matter associated with the AI/ML methods? If so, what are the implications in terms of dependencies on third-party libraries and particular versions of software in the stack?

Second is the matter of curating and disseminating training data associated with supervised AI/ML approaches. Such methods require training data, which are most often carefully curated and published in a way to promote transparency and broad use. In some cases, such training data may be quite large, and in other cases, it may even be proprietary or otherwise not suitable for an open source release.

Third is the increasing complexity of verification and validation testing. Verification testing focuses on determining if the answer a code produces is correct for a given set of inputs, whereas validation tests whether or not the code accomplishes its intended purpose, such as atmospheric modeling, etc [2]. These concepts are an important part of software engineering and testing, and they should apply to AI/ML methods as well as to the physics codes themselves.

Opportunity. Leveraging AI/ML methods as surrogates can significantly reduce computational runtime. Despite the challenges, there is enormous opportunity for leveraging AI/ML-based methods in scientific computing. Such opportunities include AI-assisted design, operation, and optimization of Office of Science Scientific User Facilities. So-called "hybrid" HPC models – those with AI surrogates – have been identified as an important class of the scientific computing workload on next-generation exascale systems [8].

Timeliness, Maturity, Impact. In terms of methodologies for the exchange of AI/ML methods themselves, projects like the Open Neural Network Exchange (ONNX) provide an intermediary framework converting models between different machine learning frameworks, and is supported by an industry consortium [1]. It provides the ability to use the same model across different platforms, and enables the exchange of models between different frameworks. The presence of curated training data has proven to be highly beneficial in terms of accelerating technology development: for example, the MNIST database consists of a collection of 60,000 images of handwritten digits [3]. The presence of this database led to a "space race" of sorts where researchers competed with one another in the development of more accurate ML-methods over the span of about two decades [6]. In terms of disseminating curated data that could be used for training, testing, and validation for scientific communities, efforts like the NIH BioMedical Informatics Coordinating Committee (BMIC) makes available about 70 different domain-specific data repositories [7] that focus on a specific type of data (e.g., metabolomics) or a specific disease (e.g., Alzheimer's). Another example is the SciML-Bench [5], which provides a benchmark suite focused on machine learning for science, including datasets and machine learning applications performing scientific tasks.

3 Legacy code bases

Challenge: Research software is vulnerable to changes in the constantly evolving computational landscape. Many scientific codes are written in legacy programming languages (*i.e.*, FORTRAN-77) and have minimal documentation of the code. These codes have in many instances been extensively validated and verified against experimental measurements, analytic theories, and other codes. These codes may still represent the "gold standard" in many cases, but their utility looking forward is limited by a lack of familiarity and/or interest in "obsolete" programming languages among younger software developers, as well as changes in hardware architecture and compiler support. Furthermore, what is a popular "modern" language today (*i.e.*, Python) is destined to become out-of-date at some future time.

Opportunity: Future proof the code and the coders. Two opportunities exist to help solve this problem. The first is to bake-in "future-proofing" when creating or extending code bases. This can come in the form of good documentation practices (verbose pseudo-code, descriptions of what each function or section of code accomplishes and how, *etc*). This makes software written in an unfamiliar programming language easier for future developers to understand and translate into a new language. This strategy is already a best, but unfortunately uncommon, practice. How to encourage thorough and helpful documentation in the scientific coding is an educational/organizational/managerial problem yet to be solved on a wide scale.

A second opportunity is to develop automated machine-learning tools that are able to perform transformations on source code, such as translating from one programming language to another or to update API calls to 3rd party libraries that inevitably evolve and change over time. Natural language processing (NLP) and translation of human

*Corresponding author: tperciano@lbl.gov

languages has made great strides in the past decade. Similar techniques may be used to translate computer programming languages as well.

Timeliness, Maturity, Impact: The impact of this effort would be significant by reducing duplication of effort in re-engineering research software capabilities that already exist. Resources could then be directed to extending existing code bases to increase capability/fidelity or including additional physics.

4 The Growing Diversity of Scientific Software Code Teams

Challenge: The diversity of scientific software teams increases as more technologies are brought to bear. In the early days of scientific computing, a multidisciplinary code team may have consisted of a domain scientist, a computer scientist, and an applied mathematician. In the present day, with the increasing diversity of technologies being brought to bear on scientific computing problems, the team also becomes increasingly diverse. Nowadays, it is not uncommon to see teams comprised of persons with expertise in domain science, computer science, applied mathematics, data science (including AI/ML), and data management, among others. The increasing team diversity poses challenges, such as learning how to speak and understand "each other's language", as well as in discovering how to work together productively in completely new ways leveraging what can often be completely unfamiliar technologies: there can be a steep learning curve in the formation and launching of such teams.

Opportunity. The multidisciplinary team is the engine the drives transformational scientific discovery. There are many potential ways that leveraging AI/ML in scientific data-centric workloads can have an impact. These range from automation, such as automatic data labelling and classification, to automated experiment setup and operation, to discovery of completely new phenomena in both new and previously collected scientific data. This very subject — identifying potential ways that AI/ML can transform science — was the subject of a 4-part AI Townhall sponsored by the DOE in 2020, which produced a report describing the numerous ways that AI/ML may affect science and scientific discovery [8].

Timeliness, Maturity, Impact. One of the hallmarks of science in the national laboratories is the formation of multidisciplinary teams. In the present, where virtually all modern science relies on software and computational infrastructure, the nature of multidisciplinary teams has undergone evolution. Some recent work suggests that the software ecosystem itself may be a model for multidisciplinary collaboration [4]. In the present day, with the increasing role of AI/ML in scientific software teams, the diversity of science and technology areas continues to grow and evolve to include more data-centric areas like data science, AI/ML, data provenance, digital libraries, and concepts like data being Findable, Accessible, Interoperable, and Reusable (FAIR) [9].

While it is the case that in the present day there are many funded scientific research projects that aim to produce scientific software that uses diverse technologies, including AI/ML, we observe that the relatively recent increased inclusion of AI/ML suggests the time is right to pay closer attention to the different dimensions of activity in these increasingly diverse code teams. Doing so would deepen our understanding of best practices and approaches for scientific software development in a time of increasing technological and application diversity.

References

[1] J. Bai, F. Lu, K. Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[2] J. S. Carson. Verification and validation: A consultant's perspective. In *Proceedings of the 21st Conference on Winter Simulation*, WSC '89, p. 552–558. Association for Computing Machinery, New York, NY, USA, 1989. doi: 10.1145/76738.76810

[3] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[4] M. A. Heroux, L. McInnes, D. E. Bernholdt, A. Dubey, E. Gonsiorowski, O. Marques, J. D. Moulton, B. Norris, E. Raybourn, S. Balay, R. A. Bartlett, L. Childers, T. Gamblin, P. Grubel, R. Gupta, R. Hartman-Baker, J. C. Hill, S. Hudson, C. Junghans, A. Klinvex, R. Milewicz, M. Miller, H. Ah Nam, J. O'Neal, K. Riley, B. Sims, J. Schuler, B. F. Smith, L. Vernon, G. R. Watson, J. Willenbring, and P. Wolfenbarger. Advancing scientific productivity through better scientific software: Developer productivity and software sustainability report. Technical report, Oak Ridge National Laboratory, 1 2020. `https://www.osti.gov/biblio/1606662`. doi: 10.2172/1606662

[5] S. J. J. P. M. S. G. F. T. H. Jeyan Thiyagalingam, Kuangdai Leng. Scimlbench: A benchmarking suite for ai for science, 2021.

[6] Y. LeCun, C. Cortes, and C. J. C. Burges. The MNIST Dateabase of Handwritten Digits, 2021. Online at `http://yann.lecun.com/exdb/mnist/`, last accessed Nov. 2021.

[7] N. I. of Health. Open Domain-Specific Data Sharing Repositories, 2021. Online at `https://www.nlm.nih.gov/NIHbmic/domain_specific_repositories.html`, last accessed Nov. 2021.

[8] R. Stevens, V. Taylor, J. Nichols, A. B. Maccabe, K. Yelick, and D. Brown. AI for Science: Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science, 2020. Online at `https://www.osti.gov/biblio/1604756`, last accessed Nov. 2021. doi: 10.2172/1604756

[9] M. D. Wilkinson and et al. The FAIR Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data*, 3, 2016. doi: 10.1038/sdata.2016.18

Evidence-based interventions for research institutions: Perspectives from NREL's software engineering interest group.

Jordan Perr-Sauer, Sagi Zisman, Rafael Mudafort, Robert White
The National Renewable Energy Laboratory (NREL)

November 2021

The National Renewable Energy Laboratory (NREL) is the only United States Department of Energy (DOE) lab focused on the development of renewable energy resources. The laboratory has multiple scientific focus areas with applications in the renewable energy sector, and also has a leadership-class high performance computing (HPC) facility. Much of the research conducted by NREL requires the development of custom software. In response to interest from the NREL software engineering community to improve the quality of its software products and software development activities, some of the authors of this position paper established a *Software Engineering Professional Interest Group* to bring together research software engineers from across the lab, provide a forum to identify and solve challenges, and build a community in which researcher-developers can find best practices, guidance, and mentorship. Inspiration for our group came from the presence of organizations such as Better Scientific Software (BSSw) and US-RSE, as well as a growing body of literature [1, 4, 5, 6] motivating and describing research software engineering (RSE) groups from universities and other large research organizations. The central position of this paper is that **scientific software is developed within research institutions, and evidence-based interventions must be made available to guide and motivate changes within those institutions to improve scientific software.**

The Challenge: Lack of evidence-based best practices to guide institutions

One of the first activities undertaken by the interest group was to identify the major challenges faced by researchers who develop software at NREL, and to suggest interventions that might help address these challenges. Examples of these challenges and possible interventions are listed in Table 1.The authors do not believe these challenges are unique to NREL. We see anecdotes of similar challenges throughout the literature, such as reports from Germany's RSE community [3], and from the DOE's Exascale Computing Project [5]. After enumerating the challenges and potential interventions, our group faces **the main challenge identified by this position paper:** *How do we provide recommendations to laboratory management to improve software development practice?* Furthermore, how can we be confident that a given intervention will provide value to the organization? Drawing a parallel to biological medicine: Medical practitioners are aware of the trade-offs associated with each medical intervention, since the interventions are rigorously and systematically studied before they hit the market. In the same way, it is imperative to understand the effects of institutional interventions before advocating for their implementation.

Table 1: Challenges and possible interventions identified by NREL software developers: Are the identified challenges universal across research institutions? Which possible interventions (if any) would provide the best return on investment for your institution?

Identified Challenges	Possible Interventions
• Users of research code are themselves developers, with varying levels of software development skill. This places a premium on usability and a penalty on complexity.	• Conduct interviews of researchers across the lab to better identify their challenges and recommendations to improve software development.
• Test cases are often times not knowable up front, since the research code is written to produce novel results.	• Build an informal community of software engineers (e.g., Professional Interest Group).
• Funding and management incentives may not be aligned with software quality, such as software maintenance or refactoring.	• Build a formal, funded, research software engineering group that provides software development services to research groups.
• Ensure that research code is high quality and results are reproducible.	• Creation of a research software engineer career track with distinct performance metrics and mentorship.
• Ensure continuity of knowledge and institutional knowledge transfer.	• Hiring more professional software, cloud, and data engineers.
• Developers may be isolated in research groups and lack knowledge about available institutional resources.	• Invest in cross-cutting, lab-wide infrastructure and internal tooling such as GitLab, JupyterHub server, other Scientific Gateway software, standardized environments, workflow management systems, and centralized data systems.
• Difficult to fund and support the development and maintenance of cross-cutting, reusable code-bases.	• Require a software development plan for new projects, with a review by a trained research software engineer.
• Accumulation of technical debt within research code-bases.	• Provide formal training, tutorials, and continuing education in software carpentry, software engineering, engineering management, programming languages, and services available for researcher staff.
	• Provide funding to support open source projects, and publicize scientific software on the laboratory's website and through social media channels.
	• Development of institution-wide best practices recommendations, such as code styles, commenting and documentation standards, architecture diagrams, and suggested libraries and frameworks.
	• Promote and provide training for agile development methodologies among managers and PIs.
	• Promote software-as-a-service architectures with well defined internal APIs.
	• Funded time for code reviews, and the development of a standard review process.

The Opportunity: Generate evidence to identify the most valuable institutional interventions

The literature provides some evidence for the impact of organization structure on code quality in the commercial world. One study from Microsoft Research found that the number of faults in Windows binaries was best predicted by metrics of the organizational structure of the business unit that produced the binary, as opposed to metrics of the quality of the source code [7]. Do these results apply to research institutions as well? Furthermore, we have Conway's Law, an adage dating back to the 1960s, which states: "Any organization that designs a system...will produce a design whose structure is a copy of the organization's communication structure" [2]. What does this say about research institutions that are organized to produce research works, for which software development is a secondary concern? In this framework, is it even possible for research groups to take advantage of reusable, cross cutting software, if the institution does not have the requisite cross-cutting structure, policy, and funding to support such software? This provides **the main opportunity:** *to focus scientific resources on generating evidence that can motivate change within research institutions that develop software* (e.g., funding models, organizational structures, and internal processes), *generating value for the research institution.*

Showing a causal link between any given intervention and software development metrics (e.g., code quality metrics and performance metrics) may not itself be sufficient to prove that an intervention will create value for the institution. In order to motivate change, the improved software quality must be shown to impact those goals which are desired by laboratory leadership (e.g., research PIs, group managers, mentors, and lab directors). A non-exhaustive list of these goals follows:

- Support the DOE's goals such as reproducible, reusable, and validated code. Maximize quantity, quality, validity, and impact of research output.
- Minimize missed research opportunities, win as many grants as possible to maximize funding.
- Improve employee's job satisfaction, personal growth goals, and institutional support, making the laboratory a more attractive place for world-class researchers to work.
- Minimize development time and cost and reduce the uncertainty around development timelines.

Timeliness: RSE groups like ours are seeking such evidence today

Research software engineering groups such as ours have formed or are starting to form organically across a variety of research institutions. Providing recommendations to laboratory management is a challenge that our group at NREL faces today. In order to recommend one intervention over another and to build a business case to laboratory management for their implementation, we wish to see the development of a body of scientific evidence, either post-hoc analysis of survey data based on best practices, or randomized-controlled-trials attempted with research code-bases being developed within large research institutions, that validates or invalidates the effectiveness and the value of each possible intervention. To begin, some potential research questions may include: (1) What is the impact of the presence of a software engineer on project milestone accomplishments? (2) Are cost and time estimates more accurate when a computer scientist is involved in the proposal writing process? (3) What are the differences in project outcomes when milestones are written to focus on software product development? (4) Does the presence of a centralized research software engineering group correlate with a greater amount of software reuse? Starting with research questions like these, we believe a body of evidence could be built to support or reject each possible intervention, ultimately improving the ability of our institutions to produce high quality research software more efficiently. Backed by a body of scientific evidence, laboratory leaders could confidently implement those interventions which are most likely to improve software development outcomes and provide value for the institution.

References

[1] Jeremy Cohen, Daniel S. Katz, Michelle Barker, Robert Haines, and Neil Chue Hong. Building a sustainable structure for research software engineering activities. pages 31–32, 2018. Publisher: IEEE ISBN: 9781538691564.

[2] Melvin E Conway. How Do Committees Invent? *Datamation*, page 4, April 1968.

[3] Hartwig Anzt et al. An environment for sustainable research software in Germany and beyond: current state, open challenges, and call for action. 9(May):295, 2020.

[4] Carina Haupt, Tobias Schlauch, and Michael Meinel. The software engineering initiative of DLR: Overcome the obstacles and develop sustainable software. pages 16–19, 2018. Publisher: ACM ISBN: 9781450357487.

[5] IDEAS-ECP Team and Collaborators. Advancing Scientific Productivity through Better Scientific Software: Developer Productivity and Software Sustainability Report. Technical report, US Department of Energy, January 2020.

[6] Daniel S. Katz, Kenton McHenry, Caleb Reinking, and Robert Haines. Research software development & management in Universities: Case studies from manchester's RSDS Group, Illinois' NCSA, and Notre Dame's CRC. (August):17–24, 2019. Publisher: IEEE ISBN: 9781728122762.

[7] Nachiappan Nagappan, Brendan Murphy, and Victor R Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. page 10, Leipzig, Germany, May 2008. ACM.

Title:

The call to quantify the impact of scientific software

Authors (with affiliations and email addresses)

Peter F. Peterson petersonpf@ornl.gov Computer Science and Mathematics Division, Oak Ridge National Laboratory

Challenge:

The purpose of science is to not to provide understanding, but to promote understanding by others. Without passing on understanding in a way that can digested , one has missed then entire point of the scientific endeavor. This is true with experimental and theoretical efforts as well as advancement in scientific software and computing. Keeping firmly focused on the ideal of propagating understanding, we as a scientific community should discus how one can measure that an individual software effort has met that goal. This goal requires two key components to provide insight as to whether an individual software product promotes scientific understanding by others: impact (either by breadth or specificity) and authenticity.

The impact of an individual software product is frequently difficult to measure. Between the lack of standard for how to reference a software product, the unwillingness of some researchers to cite software, and the opacity of dependencies, this can be a daunting task. This is not entirely the fault of the final journal author or the the group producing the software. An excellent analogy is to describe these contributions is by thinking of dependency management. A software product likely knows its direct dependencies, but likely does not know is indirect dependencies. Similarly, other than seminal references, current work tends cite recent work. There are practical limits on citation as well.

Related to the issue of citation is cite-ability. Not only is it difficult to cite an individual package, but there is a challenge of citing a package that has had a long life and evolving list of contributors can be difficult. How does one give credit? Is it to the current development team? The list of all developers? The scientific software community has yet to come with widespread practical guidance for the software producers or the software users.

The second challenge posed above is authenticity: why a software product trusted. Two axes that can be, in principle, measured are reliability (how well the software produces corrected results in expected situations) and reputation. Reliability can be thought of through mostly software engineering efforts. Does the software produce known results for know situations? Have sufficient reviews confirmed that the

software implements the requirements? Reputation, while more ephemeral, can be observable by the sheer number of citations. Related to citations, is how long a package has existed with continued use. However, simply being old should in no way be a sole endorsement of a software product.

The goal of research in this area should be to figure out what software products are impactful and why.

Opportunity:

The main argument of this paper is that scientific software has persisted for more than a half century without a proper analysis of what makes any particular project useful in advancing scientific understanding for others to build upon. The discussion above suggested avenues for changing this lack of understanding. The community needs to leverage the techniques of observational science to tackle the questions posed. However, there will need to be some effort to determine what source software should be included in surveying the state of the community before the process of studying them can be done effectively. Like any “new” area, there is expected to be an abundance of opportunities for the new researcher who is willing to explore.

Timeliness or maturity:

Rather than asking “why now?”, ask “why has this not been explored already?” One reason could be that scientific software existed in a variety of places, accessible through a variety of tools. The relatively recent move of software to hosted locations such as github, gitlab, and bitbucket have finally made it possible to have a large field of view for observing practices and methods of scientific software. This change makes studies more relevant because they can look at a wide array of projects that was previously impractical.

References:

none

Benchmark Suites for the Evolution and Maintenance of HPC Applications

Peter Pirkelbauer and Chunhua Liao

Lawrence Livermore National Laboratory[1]

Advances in system architecture and software framework design in High-performance Computing (HPC) create a constant need for source code maintenance and adaptation. Frequently, existing applications need to undergo fundamental changes in their software design before they can utilize novel hardware efficiently. Typically, this requires significant changes to how software interacts with one or more layers of abstractions (e.g., libraries, parallel programming models). In addition, the HPC architecture landscape is notoriously diverse and future systems will feature an unprecedented level of heterogeneity and parallelism in form of (CPU, GPU, specialized processing units and accelerators).

Another recurring source of change is the need to continuously evolve software to answer refined scientific questions.

HPC Software Engineering Tools

Developing, maintaining and deploying code while managing a plethora of concerns is non-trivial, expensive, time-consuming, and error prone. Novel transformation tools and techniques must be developed to simplify software deployment, maintenance and code evolution. Standard software tools developed for commercial software will be insufficient since they address the need of a different subset of software products. Arguably, HPC software engineering tool research is on the forefront of software engineering tool research as it is required to address concerns that do not appear yet (or are not important) in other areas of software engineering (e.g., the need to manage data in distributed systems with multiple levels of parallelism).

To simplify HPC software development, portability, maintenance, and deployment, software engineers need tool support, either with human-in-the-loop or automated, that can deal with the complexities of modern HPC software design and deployment. It is expected that such tools incorporate novel approaches based on machine learning and artificial intelligence into their decision system. A single tool will likely address only a subset of concerns. Thus, some applications may need to undergo a series of transformations and optimizations by different tools before they are deployed on a target platform. This raises question of correctness but also transformation composability. Before such a software engineering workflow (either with a single or with multiple tools) can be adopted, scientists need to trust that the tools “do the

[1] This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL LDRD 21-ERD-020. LLNL-MI-829420.

right thing”. A correct translation must reasonably improve the result while not introducing bugs.

Assessing Tool Capabilities

The HPC community needs ways to assess such tools. A way to achieve that is by providing a benchmark suite that guides tool developers and that helps the community assess the tools in terms of soundness and quality of transformations.

However, existing benchmark suites (e.g., dataracebench, SPEC, Rodinia, HPC proxy applications) are insufficient. Many of the benchmarks are small benchmarks that focus on a single layer of abstractions, for example OpenMP or CUDA kernels. Often these benchmark suites focus on performance and not correctness of transformations. Realistic software is composed of multiple components (e.g., simulation and data analytics) with several layers of abstractions (e.g., MPI, OpenMP, CUDA, domain specific libraries). Thus, most benchmark suites do not capture the complexity of realistic applications.

Novel benchmark suites that are representative of realistic applications in terms of scope and complexity are needed. Such a benchmark suite would need to:

- contain software artifacts that reflect the complexities of modern HPC software systems
- describe desired transformation opportunities
- provide input data sets that mimic important use cases
- provide input data sets that test "perceived" critical paths (e.g., input sets that may reveal data races or floating-point errors/anomalies in transformed code)
- be maintained to evolve the benchmark suite in line with current challenges

The compilation of a high-quality benchmark suite would offer significant benefits to tool developers and the HPC community. It would guide tool developers and set ambitious goals for transformation tools. Similarly, a good benchmark suite would foster competition among transformation tool developers. For tools that rely on machine learning, the benchmark suite would offer the opportunity to train and evaluate transformations.

The benefit for HPC software engineers is that it would allow them to assess the quality of transformation and optimization tools.

Real(istic) Specifications of Software

2021 Workshop on the Science of Scientific-Software Development and Use

Samuel D. Pollard [2], Ariel Kellison [1,2], John Bender [2], Heidi K. Thornquist [2], and Geoffrey C. Hulette [2]

[1]Cornell University, Ithaca, New York
[2]Sandia National Laboratories, Livermore, California *

Abstract: Scientific software is used for many high-impact areas such as disease and climate modeling. However, the vast majority of scientific codes do not have any machine-checkable notion of correctness. In many cases, correct behavior is checked through statistical methods, either with testing or uncertainty quantification (UQ). Since UQ can be done without understanding a program's source, it can only capture behavior and not intent of the underlying software. One promising technique to provide stronger evidence of correctness for numerical programs is to leverage domain knowledge by annotating programs with real-valued specifications about their expected behavior. These annotations can provide more optimization opportunities, better software documentation, and the ability to formally prove properties of software.

Challenge

Scientific software is used in high-consequence applications such as epidemic modeling [7] but evidence of correctness is not provided beyond the inherently limited approaches of software testing. Formal methods have been used with great success in many fields, such as distributed systems [5] and hardware design [6], but have not migrated to scientific software with the same success. One reason for this is the solution domain. For example, in hardware design, behavior for every bit is typically well-specified, or at least deterministic if unspecified. But modern scientific software execute arithmetic in parallel, which typically permits many possible solutions (for example, with floating-point arithmetic). While there have been efforts to provide reproducible parallel code [3] these have large overhead and are typically not necessary: correctness should be defined by how closely a solution models reality rather than how closely it models a previous execution.

In practice, this notion of reality is not described anywhere formally but instead is verified by domain experts who check solutions, using either testing or sophisticated UQ techniques. One strength of these methods are their scalability with respect to the size of the software. However, testing can only provide an *absence of evidence of incorrectness*, and not *evidence of correctness*.

Opportunity

Figure 1 shows an annotation of a C function. In this program, the absolute error of the square root is bounded by 2^{-43}. In this case, the property is described using Frama-C and then dispatched to the prover Gappa. Currently, these approaches work well for small-scale codes [1] but do not work well for large programs. However, the use of formal methods is not "all or nothing." Further development of specifications can prevent large classes of errors caused by invalid inputs of a function, for example.

```
/*@ requires 0.5 <= x <= 2;
 ensures \abs(\result-\sqrt(x))
 <= 2^-43 * \abs(\sqrt(x));
*/
double sqrt(double x)
{
  // implementation
}
```

Figure 1: Example real-number specification using Gappa [2] and Frama-C [4].

*Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Beyond correctness, these types of annotations can aid code optimization. For example, given that $0.5 \leq x \leq 2$, the square root implementation need not perform checks to determine whether x is negative and may require fewer floating-point operations to achieve the error bound of 2^{-43}. While this example is simple, the result generalizes. For example, properties of a matrix such as symmetric positive definiteness permit entire classes of algorithms which do not converge for general matrices.

Timeliness & Maturity

Automated analysis of floating-point error has seen a resurgence in popularity in the last decade, brought about from two hardware trends resulting from the energy and computing requirements of the exascale age. The first is the vast increase of parallelism from modern architectures. The second is the emergence of more hardware heterogeneity such as multiprecision and non-IEEE 754 floating-point arithmetic.

Because of the size and complexity of scientific software, scientific simulation is seen as "just" a simulation, and must in turn be verified empirically. This need not, and should not, be the case. The scientific community must take steps to increase the trust in its computer programs so we need not waste precious time, resources, and lives verifying what a computer says with empirical evidence. *A computer program should be evidence enough.*

This is a tall order. Developers have vastly different ways in which to approach and solve problems and asking them to change can be disruptive. However, real-number specifications—and their corresponding machine-checked properties—can help these developers by encoding some of the wealth of information contained in scientists' minds.

References

[1] Boldo, S., Clément, F., Filliítre, J.-C., Mayero, M., Melquiond, G., and Weis, P. Trusting computations: A mechanized proof from partial differential equations to actual program. *Computers and Mathematics with Applications* (2014).

[2] de Dinechin, F., Lauter, C. Q., and Melquiond, G. Assisted verification of elementary functions using gappa. In *Proceedings of the ACM Symposium on Applied Computing* (Dijon, France, 2006), SAC '06, ACM, pp. 1318–1322.

[3] Demmel, J., Ahrens, P., and Nguyen, H. D. Efficient reproducible floating point summation and BLAS. Tech. Rep. UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016.

[4] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. *Formal Aspects of Computing 27* (May 2015), 573–609.

[5] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[6] Russinoff, D. M. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, Cham, 2019.

[7] Sagoff, J. Argonne epidemiological supercomputing model showcases innovation, 2020.

Design Systems for Science

Sarah Poon, Drew Paine, Lavanya Ramakrishnan, Dan Gunter
Scientific Data Division, Lawrence Berkeley National Laboratory
[sspoon, pained, lramakrishnan, dkgunter]@lbl.gov

The growth in scientific data has resulted in fundamental changes in the user interaction model in the scientific software ecosystem. While traditionally command-line interfaces were commonplace, increasingly web based interfaces have become the primary mode of interactions for users. Scientific web interfaces are now used to access facilities, instruments, data, and computing across the DOE complex. These web applications are often custom designed to fit the unique characteristics of the workflows and infrastructure they support. The custom nature of these applications makes it difficult to reuse and repurpose code, particularly across scientific domains and facilities. This leads to challenges scaling software development efforts and results in an inconsistent user experience when you consider these applications as a collective. Design systems, which are a collection of standardized UI components and guidelines, have been developed in industry to address these same issues. We believe that a design system for science can help address these challenges in the DOE environment by providing a codified standard for scientific user interfaces that can be reused and customized.

Challenge: Scientists commonly interface with several software applications (often simultaneously) in order to conduct their work. For example, a scientist running an experiment at a lightsource might use the lightsource control software, the software at a supercomputing facility to process or archive the data, data archives to retrieve reference data, and software developed by their project to analyze the results. This software includes GUI components written by different software teams and is likely to have little standardization across the suite of these tools. This can lead to poor learnability, a higher risk of user errors, and an overall inconsistent user experience.

Several factors require scientific software to be customized in order to be effective, including differences in compute infrastructure, domain specific data, and data collection methods. This encourages a pattern of creating from the ground up bespoke software that cannot be easily adapted for another use. This is both inefficient and non-scalable from a development standpoint.

Shared code libraries can help with developer efficiency and scaling. When it comes to collaborative, web-based software, several frontend frameworks such as Bootstrap [1] and Material Design [2] also lower the burden of developing custom software by providing standardized graphic design and UI components. However, these components are often low level, such as checkboxes and radio buttons, and do not provide sufficient guidance on how to combine these components to create usable workflows. In addition, these frameworks are typically optimized for commercial applications and need to be adapted for scientific purposes. For example, science applications often require data density in their displays, whereas commercial applications favor greater use of negative space and larger fonts for improved readability. Finally, these frameworks are often missing components that rarely appear in commercial applications but are more common in scientific applications, such as ways to inspect files in scientific data formats such as HDF5 and FITS.

Opportunity: Despite the highly customized quality of scientific software, in our experience designing and developing scientific applications for a broad range of domains, from biological and earth sciences to experimental facilities, we have noticed a common set of workflow patterns. These patterns include parameterizing solvers, preparing data for archives, and

conducting real time data taking. Even across different domains and at different facilities, we've found these patterns share similar workflows and user interface patterns. We can take advantage of these patterns to create customizable templates and guidance in the form of a design system for scientific applications. The process of creating the design system will subsume a detailed study of the similarities and differences in user interfaces for scientific workflows across groups and domains.

Incorporating a design system into the development process benefits both the developers and the end scientific users. A shared design library reduces the number of design decisions a developer must make and code and allows the work to scale in a standardized way. It also ensures that the software is more easily maintained and updated to reflect best practices. Creating a more consistent user experience through design systems improves learnability, time efficiency in completing tasks, and leads to fewer errors.

Maturity: A design system is a set of reusable components and layouts that can be combined to create most user interfaces. Commonly used in industry, the purpose of a design system is to provide a unified language for cross functional teams to create consistent products and make design and development teams more efficient[3]. In addition, they codify usability principles and best practices backed by extensive user research to ensure an intuitive, reliable user experience. Design systems can contain style guides, component repositories, and pattern libraries. Examining examples from industry, Material Design and its implementations as well as Bootstrap primarily focus on the style guide and component repository elements, whereas Salesforce's Lightning Design provides some additional pattern libraries [4].

A design system for science applications would be an adaptation and extension of these existing systems, optimizing the graphic design elements to better serve the needs of science use cases and fleshing out and codifying the common scientific workflow patterns into reusable and customizable templates. It would not eliminate the need to build custom UIs for each science use case, but would make the design and development effort more efficient and maintainable and create more consistency across the DOE ecosystem. Evaluating the impact of a design system for science would involve examining gains in developer efficiency as well as factors in usability. To evaluate efficiency, we can compare the amount of development time needed to create a new web application against prior experiences developing similar applications. From a usability perspective, we can conduct evaluations to determine the impact across the collective tools used by science projects.

We believe that the role of collaborative, web based applications will only increase in helping the DOE achieve its scientific mission. A design system for science can play a significant role in ensuring that the development effort is efficient, scalable, and maintainable and that the user experience is consistent across the DOE ecosystem.

References

1. https://getbootstrap.com
2. https://material.io/
3. Therese Fesseden, 2021. Design Systems 101. Retrieved from https://www.nngroup.com/articles/design-systems-101/
4. https://www.lightningdesignsystem.com/guidelines/overview/

Can Scientific Software Development Use the Outsourcing Model Successfully?

Sivasankaran Rajamanickam[1]

[1] *Center for Computing Research, Sandia National Laboratories*[a], *Albuquerque, NM 87123, USA*

I. CHALLENGE

The primary challenge that we focus on is to arrive at a scalable model for scientific software development **and** maintenance within the Department of Energy (DOE). Scientific software development has stayed within the traditional model of in-house development within the DOE with a few notable exceptions of using libraries developed by the industry. While this model has several advantages, this position paper explores if it is possible to use an alternate approach of outsourcing certain aspects of software development for better efficiencies while also meeting the stringent requirements placed on the Department of Energy software. We use the word outsourcing in the broadest sense possible, essentially to partner with software-centric businesses for software development/maintenance as opposed to offshoring which may not be desirable for the DOE.

Based on past experience, it is not a stretch to say almost every line of code we commit today to a DOE software product requires maintaining for a decade or even two. However, the budgets for scientific software is part of new research which leads to new software development. Software maintenance has not been high priority for funding within the Department of Energy. Even with new funding, building a software focused organization that can maintain millions of lines of code will require a monumental effort. It will require competition with industry for software development engineers, a model that prioritizes both software development and software maintenance, and a reward structure that prioritizes both aspects. While it is possible to build this software-centric organization, we owe it to ourselves to explore alternate models as well. One such approach could be to outsource some aspects of software maintenance and/or development while allowing DOE scientists to focus on the science, algorithm design and software design and core development aspects.

II. OPPORTUNITY

Scientific software development and maintenance are both critical for the DOE mission in science and engineering. Development of new scientific software requires new algorithms, new interfaces, and design of new software components. Maintenance, on the other hand, is more modularized, based on well-defined, stable requirements, built on top of clean existing interfaces, and easily verified. Both efforts are key to the long term success of the DOE software ecosystem. The former requires co-design with hardware vendors, deep knowledge of domain use cases, back and forth with domain scientists, and inter-disciplinary work. The later requires attention to both past, current and upcoming hardware, deep knowledge of software testing and maintenance, experience in software engineering and a spirit to dive into a decade old code that was written for a hardware two generations earlier with little to no documentation. Both require unique and somewhat mutually exclusive skill set.

The challenging question to be addressed is whether DOE laboratories can use the outsourcing model for such software development/maintenance? If outsourcing is possible then what are the key first steps to try a model like that. In a different context, it has been argued that *process standards* are one of the key factors to decide what functions can be effectively outsourced [1]. The closest process standard to scientific software development is the process management standard developed by the Carnegie Mellon's Software Engineering Institute (SEI) called Capability Maturity Model (CMM). As a *process management standard* CMM only addresses that processes are in place for software quality as opposed to requiring a performance metric or unifying software processes for all organizations. Such a process management standard has allowed businesses trust external organizations with their non-core functionality and successfully outsource several software components that the US economy depends on (e.g. financial transactions). However, such a standard has been considered too onerous for the purposes of science software within the DOE. As someone who started his career as a software engineer in a CMM level-5 organization this author sympathises with this view.

While a process standard used by the software services industry is onerous for scientific software components, there is a new process that has emerged within the past few years. This process has come out of the Exascale Computing Program (ECP) where all software in the projects (within software technology, co-design, and application areas) follow a similar process management standard. As this is developed by the ECP, all the labs that are part of the ECP

[a] Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

use this process standard for their ECP projects. The ECP process allows clear steps for requirements gathering, approval by a management chain within the program, results gathering and reporting, review of the results, change management etc. This process has also been accepted by the science community within DOE and several large DOE software use it as part of their ECP efforts. This process is not perfect and could be improved further. This process is also not a standard currently. We argue that standardizing this process from ECP, with community input to make it better, will allow developing a process management standard based on a widely adopted process within DOE.

Formalizing a process standard for DOE scientific software could allow for an external organization to develop software practices that adhere to the standard. This could lead to confidence in the processes of the external organization and a path to verification and acceptance of their product and thereby a successful model for outsourcing.

Such a standard would also allow the DOE laboratories to assess what are their core capabilities that have to developed in-house and what are the support capabilities that can be developed in partnership with external organizations while meeting DOE needs. For example, one can ask the question to see if it will be possible to outsource activities such as maintenance of legacy code, or testing and release of software especially considering use cases from industry and other partners into account. Both these activities can be well defined using a process management standard, hence could be well supported by an external organization. This model already works at a very small scale with few companies without a defined standard for delivery. There are some Small Business Innovation Research work that are current examples for this model as well but without a defined standard.

The process standards could also allow more than one organization to develop the abilities to serve DOE needs while meeting the standards, leading to a healthy environment where DOE could choose from several organizations to partner with based on the value they provide. This approach should not be considered in any way as an alternate approach to having a software engineering team in house. Having a healthy software engineering team will still be required for a successful software delivery as needed by DOE applications. This model assumes that the in house software engineering team will both focus on the core development and will be the primary interface for external organizations that help with software maintenance or other capabilities.

There are several open questions that need to be addressed from security issues in outsourcing maintenance, training of the external organization, and funding models for such partnerships. However, if successful, this model could lead to a scalable approach for software maintenance while reducing that burden of DOE scientists and research engineers.

III. TIMELINESS

It is important to address the issue of software maintenance now as several DOE teams have invested considerable amount of time in rewriting their software to new GPU architectures. With the exascale computers coming online in the next couple of years, we expect several small changes to improve performance on additional architectures. Some of our codes still maintain different code paths for different hardware and they have to be tested and maintained on all new architectures. New efforts, such as the effort use mixed-precision or multiple precision within our software stack, lead to a combinatorial explosion of data types that need to be tested and maintained. The variety of architectures and different precision available combined lead to a large set of combinations that need to be tested and maintained that is typically beyond the capabilities of small in house software engineering teams. In addition, several open challenges in software require a rethink of current designs. For example, developing new software for emerging data flow hardware and finding the best path for integrating traditional software libraries with machine learning frameworks for scientific machine learning use cases require new software designs. We believe the in house software engineering teams will be overwhelmed adopting and maintaining software to all these changes and serve the new needs that rise due to science use cases and hardware changes. Partnerships with external organizations has the potential reduce this burden while still successfully meet the needs of science and engineering applications.

IV. REFERENCES

[1] Thomas H Davenport. The coming commoditization of processes. *Harvard business review*, 83(6):100–108, 2005.

A Seat at The Table

Elaine M. Raybourn*
Statistics & Human Systems | Sandia National Laboratories
emraybo@sandia.gov

ABSTRACT

Software is at the heart of science, and as such, its development and use have the potential to influence human behavior like no other artifact. The ASCR workshop on the Science of Scientific-Software Development and Use could foster a much-needed dialog around the ethics, equity, and responsibility of software development for science. The present position paper argues for the early inclusion and integration of diverse viewpoints, especially from the social sciences. Start by giving everyone a seat at the table.

Keywords: social science, culture, communication, software

1 CHALLENGE

> Sixteen years ago I was using a well-known software application when it dawned on me that programmers and software designers have both the privilege and power to make enduring cultural decisions for all of us, even though most of them are completely unaware of their important role. Technology (I really do mean for the term "technology" to be interpreted in its broadest sense) reflects both the intended or unconscious values and world views of its developers, designers, and researchers. Anthropologist Edward T. Hall posited that "culture is communication" in his 1959 seminal book, Silent Language (p. 97). Whether we are aware of it or not, we all communicate culture. As designers of information technology we cannot help but communicate our own cultural values and world views through the artifact we create, software [1].

The excerpt above highlights a significant challenge for the development of software as a science—that is building awareness at all levels of execution (program managers, researchers, developers, etc.) that *software communicates and influences human behavior*. Since software is often at the heart of natural sciences, engineering, and humanities leveraging computation, media, data analytics, machine learning, artificial intelligence, and others regardless of scale—we must focus attention on its development –*why it is developed, by whom, for whom, and with what effect*.

During a visit early in my career to Redmond, WA for an interview with Microsoft User Research (now the games division), I had the pleasure of meeting a Microsoft vice president who oversaw the development of MS Office. "Welcome to Microsoft," he said, "where we build 21st century products with 19th century tools!" He was, of course, referring to the hundreds of developers and programmers who wrote the software code. I was impressed because I imagined how such a small group of people had so much agency to influence the ways millions of end users think, communicate, and learn. How many social scientists do they work with, I wondered? How often do they discuss ethics or the second and third order effects of their decisions? Do each of the developers realize the immense responsibility they bear by developing products and processes that are in use daily around the globe? That conversation turned out to be a game changing event for me. Since people develop software, it communicates, whether we want it to or not.

2 OPPORTUNITIES

To explore how software communicates and influences behavior the ASCR workshop could bring together international scientists and practitioners from major areas of scientific inquiry (social, biological, physical, computation, engineering, and mathematics) as well as the humanities by inviting them into a dialog about each person's role (at various levels) in the ethics, equity, and responsibility of software development. In other words, the best way to incorporate and integrate diverse viewpoints in shaping a nascent field of inquiry such as the science of scientific-software development, is to invite international participation from diverse perspectives and disciplines early in the process and above all, provide everyone an inclusive and equitable opportunity to shape the new area of scientific inquiry, by offering *a seat at the table*.

> An initial step…is to understand one's own culture and world view. We architects of … technology should carefully consider and *openly communicate* the cultural limitations of our approaches and solutions. Focusing on developing an acute awareness of our own biases (cultural or other) that unwittingly make their way into our software designs is a good start to owning this responsibility. We can better foster more equitable and inclusive approaches

* Elaine M. Raybourn is a social scientist and artist who has dedicated much of her career to the exploration of cultural awareness through information technology. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC- 20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. SAND2021-14911 O.

> by carefully examining the impact of ethnocentrism in our information technology development and identifying our own cultural assumptions and how these assumptions can be communicated by our software. This practice will take us further toward understanding ourselves and others, as well as toward how to best create equitable…technology [1].

While such a practice of examination is useful at all stages and among all levels of research, it is often best executed early in the development of the requirements for a funded program, that is, during the phases of strategic planning and research agenda setting. A seat at the table at this stage for social science and humanities may result in the co-creation of research questions and hypotheses that could define the impact of findings or theory development on the use of software by a global community. Beyond the execution phases of the research, a potentially more valuable role for social science and humanities is in the evaluation and assessment phase of the funded program itself, as a SETA (e.g. U.S. Department of Defense Systems Engineering and Technical Assistance)—assisting teams in articulating, meeting, and exceeding research goals. SETA personnel are contractors that act as consultants and work closely with the U.S. government staff in the execution of a research program.

There is no need to start a program of research that reflects on the characteristics of the science of software development from scratch or bear the responsibility of funding such a research initiative alone. Existing frameworks may be leveraged to guide scientific inquiry [2]. Additionally, the challenge of ethics, equity, and responsibility in the science of scientific-software development should be of import to all disciplines touched by software development, and they could be enlisted to share the financial burden (see Figure 1.).



Figure 1. Avoid this stereotype by offering a seat at the table. Computational science need not bear this responsibility alone.

2.1 European Commission model

The European Commission grant funding agency encourages project teams comprised of universities, industry, and government labs. Additionally, they enlist the service of scientists and practitioners in the roles of evaluators, observers, ethics evaluators, monitors, ethics monitors, and business coaches [3]. In these roles, individuals must maintain objectivity and may not represent or act on behalf of specific institutions. In the United States, for example, national laboratories are uniquely positioned to participate in such roles, given that the Department of Energy (DOE) national labs are federally funded research and development centers (FFRDC) and contractors for the Department of Energy.

2.2 DARPA model

The Department of Defense (DOD) Defense Advanced Research Projects Agency (DARPA) applies a similar model that frequently employs SETA (Systems Engineering and Technical Assistance) contractors. These contractors perform a variety of roles including but not limited to contract management and administration, technical, programmatic, financial and administrative program support tasks, as well as program assessment and evaluation [4]. Given the nature of the SETA role, these individuals often have a unique seat at the table and the ear of the program manager.

3 Why Now?

There is no better time than the present to explore ethics, equity, and responsibility in the science of scientific-software development and use. Let us apply a variety of scientific and critical methods from social science, natural science, computation, engineering, and humanities to examine and ultimately value the complexity of our interdisciplinary programs, teams, researchers, developers, processes, and uses of software.

> The daunting realization is that even though we remain unaware of how or what about our culture is being communicated, we can no longer ignore the tremendous responsibility that comes with the privilege of creating…information technology for global, multi-cultural users… As scientists, programmers, and software designers, we all have a role in the creation of cultural information technologies that can impact the course of human history [1].

References

[1] E. M. Raybourn. Foreword. In E. G. Blanchard and D. Allard (Eds). Handbook of Research on Culturally-Aware Information Technology: Perspectives and Models. IGI Global, 2011. http://doi:10.4018/978-1-61520-883-8.

[2] F.B. Aydemir and F. Dalpiaz. A Roadmap for Ethics-Aware Software Engineering. FairWare '18: Proceedings of the International Workshop on Software Fairness. May 15-21. https://dl.acm.org/doi/10.1145/3194770.3194778.

[3] European Union Horizon Research Funding and Tender Opportunities: Working as an Expert. Last retrieved November 22, 2021. https://webgate.ec.europa.eu/funding-tenders-opportunities/display/OM/Working+as+an+expert.

[4] Defense Business Board. Future Models for Federally Funded Research and Development Center Contracts (2016). Last retrieved November 22, 2021. https://dbb.defense.gov/Portals/35/Documents/Reports/2016/2016-10/DBB%20Briefing%20-%20FFRDCs%20-%20Approved%2020%20Oct%202016.pdf.

Is a Language Barrier Impeding Development of Better Scientific Software?

Ryan M. Richard*
email: rrichard@ameslab.gov

Theresa L. Windus*†
email: twindus@iastate.edu

*Ames Laboratory. Ames, IA 50011
†Iowa State University. Ames, IA 50011

November 19, 2021

1 Challenge

Modern software engineers have identified a number of best practices (BP) including: documentation, revision control, continuous integration, issue tracking, versioning, encapsulation, code formatting, automated testing, and code review[1]. We have noted that within our own field of electronic structure theory (EST) only a handful of development teams actively and fully embrace BP, despite anecdotal evidence that most development teams understand the value of adopting BP. The existence of this solicitation suggests that this problem extends beyond EST, to computational science in general. Within the field of EST we have noted that the software packages typically seen by the community as being exemplars of BP tend to have Python application programming interfaces (APIs) and are usually hosted on GitHub. To us, this raises two interesting questions: "Is there a correlation between a team's willingness to adopt and adhere to BP, and the programming languages/tools they use? If so, why?"

Our hypothesis is that there is a correlation between using older programming languages (*e.g.* C, C++, or Fortran) and a reluctance to adopt and adhere to BP. Again, our anecdotal evidence suggests that these same developers recognize the importance of BP, so why the reluctance? We suspect the primary reason is inertia. Many of the projects which use older languages are legacy codes, which were started before BP were widely known to most computational scientists. For legacy projects, adopting BP means repaying (sometimes decades of) technical debt. We also suspect that many of the developers relying on older programming languages have been doing so for some time. These developers have likely been using the same workflows for years and do not want to sacrifice immediate productivity by having to learn and/or implement a new workflow.

Another reason older programming languages may not use BP is language complexity. More specifically, many of the older languages are viewed as being more complex. On the other hand, newer languages such as Python, are typically seen as being very user-friendly. We suspect that the complexity of the languages affects their popularity. For example, Python is currently one of the most popular languages in the world and enjoys a large active development community. This development community has developed a plethora of tools, which can be leveraged to readily adopt BP. With less popularity, comes less available tools. When tools do exist, ensuring the tool is generally useful often requires the tool maintainers to embrace the complexities of the underlying language; this can require users to overcome significant learning curves before they can use the tool. Finally, we suspect that because older languages are more complex, developers may end up spending a lot of time exploring algorithm possibilities (especially for performance purposes), which leaves them short on time for adhering to BP.

2 Opportunity

The previous section conjectures that there is a language barrier preventing adoption of BP. The first step in rectifying this problem is to establish its existence. To this end we propose soliciting input from the scientific community about their software development practices. In particular we are after more information regarding: programming languages used; which BP the teams are aware of; which BP the teams use; for utilized BP, what fraction of the package adheres to the BP; what tools are used to implement each BP; for BP that are not used, why are those BP not used; and is the team familiar with tools that could be used to implement additional BP. For

teams which host their codes on GitHub, or a similar service, additional insight can be gleaned from the repository statistics. While still not foolproof, the repository statistics have the benefit of being more objective than a survey.

Assuming that the initial surveys corroborate our hypothesis, the next step is to identify where tooling gaps are *perceived* to exist and to establish where they *actually* exist. What we mean by this is: tools may already exist, even if the development teams are not aware of them. As an example, our own anecdotal evidence seems to suggest that many C++ developers are unaware of package management solutions, such as Spack[2] or Conan[3], which could simplify their build processes and facilitate integration with other projects. In cases where tools already exist, campaigns focusing on increasing public awareness, *e.g.*, through tutorials and seminars, may prove fruitful. In fact initiatives like better scientific software (BSSw) are already underway, and seem to be achieving increased awareness.

The more interesting and also more challenging outcome from the proposed survey is the identification of actual tooling gaps. Here the challenge is two fold: securing funding to develop the tool, and having the community actually adopt the resulting tool. The former challenge remains a perpetual problem with no easy solution; the hope is that workshops, like this one, will raise awareness with funding-agencies for the need to fund tools which support scientific software development. One way to facilitate community adoption of the developed tools is to solicit input from the affected communities before and during the development cycle. This practice is more common in non-scientific software development and relatively rare in scientific software development. Although one could argue that the scientific software developers are also users of the software, this can cause a myopic approach to the features and workflow of the software. Using the feedback practice, the tool is more likely to be responsive to the community's needs and will presumably integrate better into existing workflows.

3 Maturity

Over the last five or so years there has been increased interest in better and more sustainable development of scientific software. This interest is from both the scientists actually writing this software, and the agencies funding the development. For example, through the Exascale Computing Project (ECP), Department of Energy (DOE) has funded the creation of BSSw, and the Molecular Sciences Software Institute (MolSSI) has been funded through the National Science Foundation (NSF). These initiatives have provided increased awareness of BP to scientific computing communities, and perhaps more importantly have made these communities receptive to BP. With respect to the proposed opportunity, we foresee such organizations playing pivotal roles in coordinating polling efforts, organizing workshops, and continuing to increase awareness of existing tools and BP.

Another recent development which facilitates addressing the language barrier is the closer integration of continuous integration pipelines with online hosting sites. For example GitHub added "GitHub Actions" in early 2021. Using GitHub Actions, developers can interface their repositories with a number of actions (each of which is essentially a micro service). For many languages actions already exist for: versioning, linting, formatting, and many other BP. With just a couple of markup lines, developers can add actions to their repositories. Writing additional actions has a relatively small learning curve. In our opinion developing tools which tightly integrate with continuous integration pipelines is one of the ways most likely to lead to widespread adoption of BP.

Whether it is for performance, or for legacy reasons, a large portion of scientific software is written with older languages. Assuming that, as hypothesized, the complexity of these languages is a barrier to adopting BP, then increasing the community's awareness of available tools, offering training resources for said tools, and developing new tools which facilitate applying BP to these languages should catalyze adoption of BP. Long-term we expect this to lead to a significantly more productive development community.

References

[1] Better Scientific Software. Better Scientific Software (BSSw). https://bssw.io/, 2021. Accessed: 2021-11-17.

[2] T. Gamblin, 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*, Austin, Texas, 2015.

[3] JFrog LTD. Conan: The open-source C/C++ package manager. https://conan.io/, 2019. Accessed: 2021-11-17.

Towards a More Sustainable Software Development Environment for HPC Scientific Applications

Todd Ringler[1], Science and Technology Policy Institute

Overview: Earth System Models (ESMs) are used to exemplify the broader challenges in achieving more sustainable software development (SSD) practices. The example is used to motivate opportunities that impact all stakeholders, ranging from domain scientists to the top level of government. The proposed opportunities address shortcomings in cultural values, resource allocation, and policy gaps.

Understanding the Challenges: An ESM is the quintessential use case to exemplify the challenges confronting SSD. The software representation of an ESM has grown incrementally over the decades and now typically extends over hundreds of thousands of lines of code. All aspects of software development, from idea creation through code optimization, have been (and still are) predominately the purview of domain scientists who have little to no formal training in software design.

ESMs are increasingly used to inform policy-relevant decisions. The resources directed towards maintaining the software of these critical applications are increasingly misaligned with the growing uses and consequences of the simulation output. This misaligned state grew from two mutually reinforcing causes: the scientific communities' cultural values and agencies' resource allocation.

While a domain scientist can spend an inordinate amount of time developing, testing, optimizing, and debugging HPC software, the *process* rarely evolves into a *practice*. For agencies, scientific discovery is the core metric of success for evaluating ESM-related programs. The software that comprises an ESM is considered a work product produced through grant support.

The duration and size of investment, along with the increasingly consequential application space, suggests that ESMs are much more than ordinary work product. Individually, an ESM has the appearance of a "facility" and, collectively, all the ESMs have the appearance of "infrastructure." The upshot is that government investment of ESM software is not commensurate with either the expected life of the models or the importance of their applications.

Trends: At a system level, what trends are likely to be conducive to SSD?

- The role of HPC software engineers in the development, porting, and optimization of HPC-specific software will grow through time.
- HPC scientific applications will increasingly leverage libraries and community-shared code, both of which reside outside of the application code-base repository.
- Continued migration to open-source development will be necessary for projects to adopt tools, practices, and policies associated with software sustainability best practices.

[1] About the author: Dr. Ringler spent twenty years developing Earth system models in university, laboratory, and multi-laboratory settings. His expertise was in geophysical fluid dynamics and computational physics. Throughout his career he advocated for the adoption of better software development practices, including revision control, creation of requirements documents, automated regression testing, and open source development. From 2018 through 2021, he was a National Security Science Fellow on Capitol Hill with a legislative portfolio including science, technology, and national security policy. He now works at the Science and Technology Policy Institute, an FFRDC supporting the White House Office of Science and Technology Policy.

Characteristics: Collectively, what are the characteristics of opportunities that are likely to be conducive to sustainable software development?

- Intervention across all levels -- scientist, program, laboratory, agency, and interagency – will be required to elicit systemic change to SSD practices.
- Certain scientific HPC applications, community-shared applications, and/or scientific software libraries will "lead the way" for the demonstration, evaluation, and adoption of SSD practices.
- Critical levels of agency and scientist engagement, along with sufficiently long-time commitments, will be needed to elicit a durable adoption of SSD practices.

Mainstreaming: The first opportunity is built on the concept of "mainstreaming," i.e., the integration and consideration of SSD practices across all aspects of the scientific process. Since the proposal process is the foundation to supporting HPC applications at science-mission agencies, the most obvious mainstreaming opportunities reside in the proposal life-cycle process. In each stage of the process – grant submission, external review, program manager decision, interim reviews, and renewal – the scientific team is obligated to articulate their SSD strategy. Scientific Focus Areas (SFAs) within DOE's Office of Science are particularly good candidates for mainstreaming. Integrating SSD practices into the SFA process would also provide an opportunity for offices and divisions within DOE to collectively articulate their expectations of the scientific community with respect to SSD practices.

Elevate stature of HPC scientific applications: Designating certain HPC scientific applications to be of national importance would serve two purposes. First, the designation implies importance of HPC software as an enterprise. Second, the designation sets the stages for creating new lines of funding to support software sustainability and productivity. At a minimum, this idea has value in stimulating thinking for how to elevate the importance of HPC scientific applications across the U.S. Government.

An ARPA-like approach: It is likely that continuous, incremental change toward better software sustainability and productivity practices is not the optimal path. Rather, better paths to improved HPC software practices might be discovered through the exploration of radical, transformational ideas in ARPA-like programs that embrace high-risk, high-reward ventures.

Competitively awarded software sustainability projects: This idea, while less ambitious than the ARPA-like approach, entails the creation of agency-level funding lines to implement software sustainability practices (or demonstration projects) within existing HPC applications, community codes, and/or libraries. For example, one could imagine a software sustainability funding program where all DOE Office of Science SFAs were able to compete.

Require agencies to articulate a software access and impact strategy: The last idea is to expand the 2013 Office of Science and Technology Policy (OSTP) Memo on public access of federally funded research results to include software access and impact.[2] The 2013 OSTP memo does not mention access or impact of federally funded software. Software access strategies would require agencies to consider how software is the same and different from publications and data. This, in turn, opens the door to highlighting the uniqueness of HPC scientific software applications, their importance to the nation, and the urgent need to develop tailored policies and programs in support.

[2] Holdren, J.P., 2013. Memorandum for the Heads of Executive Departments and Agencies: Increasing Access to the Results of Federally Funded Scientific Research.

Investigating Software Developer Roles Supporting Reproducibility and Sustainability

David M. Rogers
rogersdm@ornl.gov
ORNL NCCS
Nov. 22, 2021

Challenge

As a computational scientist, I want to be able to exactly reproduce and easily extend existing computational results in order to make improvements and comparisons. However, there is a well-documented and persistent difficulty in reproducing computational results. This position paper argues that the disconnect is directly caused by lack of incentives to software developers for producing community-accessible software. The ultimate cause of this shortage may be an over-reliance on individual, business unit, and organization-level "metrics" for individual success as opposed to community success.

My argument is derived from an economic model of the costs and benefits of producing open-source scientific software and computational research results. After establishing that software developer-hours directly determines the cost of software, I examine three categories of developers: *federally-funded applied-science researchers*, *software center and product specialists* (including full-time maintainers of large community science codes), and *hardware vendors*. I also define three organizational structures motivating consumption of software: *applied industry*, *consortiums*, and *academic grant-funding agencies*. The conclusion is that effort towards sustaining software requires clearly defined project scope and recognition for those contributions.

Argument

Digital publishing (including software) is unique because its cost derives entirely from labor. The product itself can be copied essentially for free. So, its cost can be divided among all recipients. Software consortiums like the Apache, Python and OpenJS Foundations recognize this fact and structure their organization to maximize recipients who derive value from the software. Open-source achieves equitable cost division when the cost of labor devoted to building a software project equals the relative value that organization derives from the product. If one organization produces a software product, but a different organization derives the majority of the value of that product, some payment must change hands. Otherwise, the producer will not be able to continue to develop and support the product. In order to understand the cost of releasing a large software package, we need to look at these exchanges and imbalances at an organizational and at a person-role level.

The "cost per developer-hour" rules the cost companies must pay for digital infrastructure. Developers are also humans, who have personal, moral preferences for places, companies, and projects. This is why defense contractors in extreme climates who are required to work late nights on projects must have hugely inflated salaries, and also why scientists and graduate students working under favorable management conditions on academic projects they feel personally attached to require relatively little salary. Economically, achieving software-related goals like reusability, reproducibility, and sustainability requires diverting developer hours to those tasks.

Now we consider the incentives for each category of developer. *Federally-funded applied science researchers* typically have small groups of principal investigators, external collaborators, and students focused on an application-driven project. Their publication and grant output is the primary metric to their success, and hence there is little or no incentive to produce sustainable software or research artifacts. The situation is very different for *software-focused centers*, who develop and maintain large simulation

packages for a community of researchers. I include national research facilities in the category of *software-focused centers* because the focus is on enabling community outcomes. Their primary output is successful community adoption of a package or facility. This model is not without flaws, as it does not necessarily encourage inter-operability with other centers. The extreme of the "shrouded source" model is reached by yet another group, the *hardware vendors*. Vendors are motivated to create easy to use and performant systems by customizing both physical products and software together for their customers. However, reproducibility and inter-operability carries a high cost both to the customer and the vendor. Those costs must be somehow born by the community if it wants to maximize external benefit from new products.

Software consortiums are one powerful mechanism to amortize the cost building in reproducibility. By producing standards for hardware and software interactions, consortiums can leverage effort from all parties for common gain. *Industrial and federally sponsored research funding agencies* typically tie project and grant funding to demonstration of new outcomes and applications. When agencies ignore the long-term utility of reproducible research, the definition of "high-impact" science and software unconsciously shifts to a more commercial character. This has been a direct contributor to, if not the cause of, the reproducibility crisis in science. Expecting novel discoveries to be patented and principal investigators to become research stars discourages effort towards making science reproducible. This has a self-perpetuating effect when research stars gain exclusive access to and control of funding decisions. It also has a chilling effect on collaborations that has been interpreted as "ego."[1] Consider, as evidence, that principal investigator salaries are commensurate with project managers at corporations, and their bonuses and career advancement are derived from winning contracts.[2] Nevertheless, *Consortiums*, are not without flaws either. Consortium incentives are set directly by the input of whatever community is empowered enough to be engaged, leaving little or no incentive for supporting small, applied (usually academic) projects.

Opportunity& Timeliness

Our present system of interactions between applications, developers, corporations, and agencies does not directly incentivize sustainability, reproducibility, and inter-operability together. Prioritizing the accessibility and sustainability of scientific software over short-term outcomes represents an enormous, potential gain for the community. The benefit would be measurable by increased access to and reproduction of software-related scientific results, enabling solid, steady progress. However, realizing this gain requires adjusting our understanding of organizational and academic output to include gains made by incremental, reproducible means. We have already seen some of the outcome of this approach to science within both DOE labs and parts of the Exascale Compute Project. *Software-focused centers, product specialists*, and *consortiums* that are funded based on positive overall community-outcomes have become centers of collaboration and re-usable software components. Further adopting a collaborative, *consortium-style* model to agencies supporting those centers could further decrease barriers to producing inter-operable software components. We have also seen the opposite trend with short-term academic projects. Rigidly defined roles for "lead" investigators prevents software development or collaboration because of competition for funding and lack of recognition for members who "only join" a community.[1,2] As above, the success of a sustainable community software model requires funding agencies to look at organizations, rather than the individual accomplishments.

References:

1. Sharing the spoils: Incentives and collaboration in scientific software development James Howison and James D Herbsleb. Proc. CSCW13, 459-470, 2013.
2. Jaynes, E. T., 1993, `A Backward Look to the Future,' in *Physics and Probability,* W. T. Grandy, Jr. and P. W. Milonni, Cambridge Univ. Press, Cambridge, England; as a postscript file.

Viewing Scientific Software Development Through the Lens of Economics

Philip C. Roth
Oak Ridge National Laboratory
Oak Ridge, TN 37831 USA
Email: rothpc@ornl.gov

I. Challenge

One of the aspects of my role within the Oak Ridge National Laboratory (ORNL) National Center for Computational Sciences (NCCS) is to engage with teams that are developing applications and libraries to run on NCCS systems. This engagement may take several forms, from porting software so that it will run on a system with a different type of accelerator or interconnection network than the team has used before, to designing and implementing new capabilities that are tailored to our systems' hardware or software, to measuring and improving the performance of software already running on those systems. For these engagements to be successful, it is important to obtain a solid understanding of a project team's goals, skill sets, experience, and preferences.

It is possible to obtain some amount of this information using ad hoc methods. For instance, early in an engagement with a new project team I often try to gauge their relative interest in portability versus absolute performance on a specific system by posing them a hypothetical scenario: if I gave them a budget of $100, how much of it would they spend on achieving absolute best performance on our current system versus on support for portability? Often, the response is to ask for $200. The desire to develop software with both characteristics is understandable, but the response suggests the project team may not yet have clearly thought about their project's needs and goals and how they change over time. While ad hoc methods such as this shed some light on important software development project attributes, it is generally difficult to quantify them in a way that allows facility staff and the project team to best understand each others' priorities and timelines, and to identify gaps that would negatively impact the project team's potential to develop software that runs effectively on our facility's systems.

II. Opportunity

In the spirit of my $100 scenario, I propose that we could better understand the development activities of scientific software teams by viewing them through the lens of economics. The idea of applying economics to engineering and technology activities is not particularly new, in that economists have studied various aspects of engineering and information technology for decades [1], [2]. Indeed, the microeconomics subfield of "engineering economy" (also known as "engineering economics") has existed for at least six decades and focuses specifically on the analysis of the economics of engineering decisions [2], [3]. However, the bulk of the work in this area seems to be more focused on physical engineering activities (e.g., civil) than software engineering. More recently, there have been efforts to apply economics principles to analyze software development [4], [5]. To my knowledge, none have focused specifically on the development of *scientific* software. While I fully expect there to be common characteristics between a project developing a web browser and another developing a molecular dynamics simulation, I also suspect there are points where they diverge, such as incentives based on the way they are funded, what drives their timeline, and the needs of varying types of team members (e.g., chemistry students working to earn their PhD versus computer scientists versus trained software engineers).

Underlying my proposal to view scientific software development through an economic lens is the hypothesis that we could better understand (and thus manage) the development process if we take into account the incentives and constraints facing the project team and other stakeholders, and had a clearer and quantifiable notion of the utility (i.e., value) of various parts of their activity. For instance, there may be great value associated with adding a new force calculation to the molecular dynamics simulator mentioned earlier, but only if it is completed in time to be used for the PhD student's dissertation. As another example, it could be useful to quantify the value of having more accurate performance predictions when they are used to define the requested run time to a batch job scheduling system, especially if the degree of accuracy were variable. E.g., should the project team devote the effort into developing an implementation that is 10% more accurate than its existing implementation, if it isn't clear that it would produce a 10% better schedule? And what metric should be used to identify that the new schedule is 10% better? And yet another example: in my $100 scenario, what *is* the actual benefit of devoting $75 (i.e., 3/4 of one's time and resources) to implementing the software in a portable way, versus only devoting $25?

Fundamentally, economics seems to provide the right tools for analyzing these common scientific software situations. I do not have an economics background, and economists are not typical members of the high performance computing community. Thus in proposing an effort to study scientific software development using tools provided by economics, we would be forging a collaboration with members of a distinct

community, but one that has the potential to provide great insight into an key activity that greatly impacts ours.

III. Timeliness

To my knowledge, there is no particular recent breakthrough in either economics or scientific software development that enables a collaboration between scientific software developers and economists that was not possible before. Instead, this type of collaboration may be timely *only* because it hasn't been done before, coupled with the fact that problems with the development of scientific software has become enough of a concern for people to organize workshops such as this.

The impact of success for such a collaboration is expected to be large. For the software developers, a better understanding of their needs, goals, and incentives could help them better manage their development process, including helping them justify why they should undertake certain activities (e.g., introducing continuous integration, porting to a performance portable abstraction layer). For the facility staff engaged with such software developers, it would help prioritize who needs attention and effort, and on what type of activity. For the agencies funding the research that is driving the software development, it would help them recognize valuable activities that should continue to be supported versus those that are less critical. Regardless of the stakeholder, I expect the application of economics principles to understand scientific software development to have great positive impact going forward. But quantifying that impact is, in itself, another question of economics.

References

[1] H. R. Varian, J. Farrell, and C. Shapiro, *The Economics of Information Technology: An Introduction*, ser. Raffaele Mattioli Lectures. Cambridge University Press, 2004.

[2] W. T. Morris, *Engineering Economy: The Analysis of Management Decisions*. R. D. Irwin, 1960. [Online]. Available: https://books.google.com/books?id=VMcmAAAAMAAJ

[3] W. G. Sullivan, E. M. Wicks, and C. P. Koelling, *Engineering Economy*. Pearson, 2018.

[4] Y. Wang, *Software Engineering Foundations: A Software Science Perspective*. Auerbach Publications, 2007, chapter 12 focuses on economics.

[5] J. Bitzer and P. J. Schröder, Eds., *The Economics of Open Source Software Development*. Elsevier Science, 2006.

Expanding the Scientific Software Development Community Necessitates Studying Cognitive and Social Issues in Programming-Language Learning and Bias

Damian Rouson*, Ondřej Čertík**, Brad Richardson***, Renée Blake****, and Zachary Jaggers*****

*Computer Languages & Systems Software Group, Lawrence Berkeley National Laboratory (rouson@lbl.gov)

**Computational Physics and Methods, Los Alamos National Laboratory (certik@lanl.gov)

***Research Software Engineer, Archaeologic Inc. and Sourcery Institute (brad@archaeologic.codes)

****Department of Linguistics and Social & Cultural Analysis, New York University (renee.blake@nyu.edu)

*****Language Engineer, Amazon (zjaggers@amazon.com)

Challenge: The call for position papers notes that the development of scientific software has leveraged a diverse community — including domain scientists, applied mathematicians, and computer scientists — to achieve tremendous advances in the models, computer architectures, and software that support scientific discovery. The call envisions expanding upon these successes by nurturing collaborations between computational scientists, cognitive scientists, social scientists, and software engineers to study the software development process itself. This paper, co-authored by computational scientists, a research software engineer, and social scientists, focuses on programming languages. We identify two challenges as necessary conditions for growing the community that develops scientific software and advancing the community's practices:

1. We must better understand cognitive issues around how developers learn programming languages.
2. We must better understand how to ensure language inclusivity in AI-driven development.

Opportunity: Confronting the above two challenges offers an opportunity to examine several of the questions posed in the call for position papers, including the questions this paper addresses.

How can we best integrate social and cognitive sciences into scientific software activities? Social and cognitive scientists have published an extensive body of literature examining the effectiveness of various strategies for learning second languages. For example, neuroscience and social-science scholars who studied adults learning to speak and understand an artificial language found that, among research subjects who exhibited high levels of proficiency after either an "explicit, classroom-like, or implicit, immersion-like training," the implicitly trained group demonstrated more native-like neural processing after a several-month break in instruction [1]. The scientific community's increasing adoption of interpreted programming languages such as Python and Julia, which offer interactive, immersive modes of use, presents an opportunity for a comparative study of learning interactive languages versus learning the compiled languages that remain prevalent in high-performance computing. Better understanding one experience might inform and improve the other.

How will AI-driven development tools affect best practices for scientific software development and use? How can we study changes in culture affecting scientific software development and use? How are scientific software developers and users

atypical from other larger software communities? The intersection of these three questions exposes fruitful territory for inquiry into how scientific software development communities can ensure that AI-driven development of scientific software provides a productive development experience for a community working in diverse programming languages. Surveys of real-world AI applications have identified important problems related to bias and fairness and have created a taxonomy for fairness definitions [2]. Scientific software developers' work inherently involves a related atypical and subtle paradox: we communicate through code that aims to instruct computers despite the actual target of source code being humans [3]. Hence, one can easily miss the analogy between programming-language biases and broader societal biases that marginalize groups who speak in coded language forms. To wit, a recent study of the use of Aviation English (AE) code by non-native English speakers in avionic transmissions demonstrated the AE speakers' disadvantaged status and heavier burden relative to native English-speaking pilots and air controllers, despite AE's utility. [4] We assert that similar biases exist between programming language communities and it will be important to ensure that such biases do not creep into AI-driven development in the manner that other forms of bias have infected other AI applications.

Timeliness: Recent work has demonstrated that even compiled languages can provide an interactive experience similar to those of popular interpreted languages. LFortran, for example, is a modern, open-source, interactive Fortran compiler built atop the LLVM compiler infrastructure. [5] In addition to its interactive mode resembling an interpreter, LFortran can compile to binaries. Similarly, the jupyter-CAF-kernel project offers an interactive experience with Fortran's native parallel programming model running in the cloud [6]. Jupyter-CAF-kernel leverages the open-source GNU Fortran compiler and OpenCoarrays parallel runtime library. Developed with newcomers in mind, LFortran and jupyter-CAF-kernel could facilitate studying cognitive issues related to learning an interpreted versus a compiled language *without* varying the language. At the same time, an emerging awareness of social issues in AI and in coded language inspires a partnership with social scientists to study fairness in AI-driven development tools and to foster an inclusive culture across the diverse family of scientific programming languages.

References

[1] Morgan-Short, K., Finger, I., Grey, S., & Ullman, M. T. (2012). Second language processing shows increased native-like neural responses after months of no exposure. *PLoS One*, *7*(3), e32974.

[2] Roselli, D., Matthews, J., & Talagala, N. (2019, May). Managing bias in AI. In *Companion Proceedings of The 2019 World Wide Web Conference* (pp. 539-544).

[3] Richardson, B. Communicating with Whom? (https://bit.ly/3oLyqbe)

[4] Trippe, J. (2019) Aviation English listening and repeating task for native English speaker and non-native English speaker pilots, 20th International Symposium on Aviation Psychology, 235-240. (https://corescholar.libraries.wright.edu/isap_2019/40)

[5] Curcic, M., Čertík, O., Richardson, B., Ehlert, S., Kedward, L., Markus, A., Pribec, I. & Vandenplas, J. (2021). Toward Modern Fortran Tooling and a Thriving Developer Community. *arXiv preprint arXiv:2109.07382*.

[6] Beekman, I. and Nemichev, V. (https://github.com/sourceryinstitute/jupyter-CAF-kernel)

Improving adoption of scientific software best practices through the development of a software Impact Factor

Kelley Ruehl, Sandia National Laboratories, kmruehl@sandia.gov
Katherine Klise, Sandia National Laboratories, kaklise@sandia.gov

Challenge

Research projects often require development of custom scientific software, or research software, to address novel research questions and advance state-of-the-art methods. Scientific software is often written for a very specific purpose. It is commonly used to prototype ideas and generate results quickly. Generally, the software is not intended for use outside of the main research group. Consequently, software development best practices, like writing documentation and testing the software, are often not prioritized. This focus on quick results can be driven by many factors, such as an immediate deadline, limited budget, and team dynamics. This can be great for short-term success, but there are often long-term costs.

Several challenges exist for scientific software, including verification, validation, reproducibility, and software maintenance [1,2,3]. With a growing number of software options available for research, it is important that software not be trusted blindly. Both scientific and open source software packages should be vetted before use. In the early stages of a project, the research team must decide how the custom software will be developed and maintained and which software dependencies will be included. Too often, software development starts without fully considering how these software architecture decisions will affect short-term results and long-term impact.

What starts as a software prototype can become the basis of a multi-year research project and eventually evolve into an open source software package. Research teams can quickly grow and evolve, and software can easily grow from a thousand to millions of lines of code. When speed is prioritized over best practices, the result is often unreadable code, undocumented features, possible errors in the software, and single-developer knowledge of features. Many times, these challenges are only identified once the project is over and key developers have moved on. These factors make long-term development and support of scientific software very challenging.

Opportunity

Software development best practices include version control, documentation, quality assurance, source code management, and a software development plan. These aspects of software development can seem burdensome, especially for scientific software developed for an immediate research need. However, writing clean code, documenting features, generating tests, and completing rigorous review by team members enables long-term success. In addition to producing a more cohesive software product, they enable higher quality research and ensure reproducibility by the research team and by members of the scientific community. The initial investment can be reduced by leveraging existing styles and packages to free time and budget to create documentation and tests. This saves time and budget in the long-term, resources that can then be used to advance future research.

The increased use of open source software within research projects has improved adoption of software development best practices. This applies to both scientific software that is intended for open source distribution and to software that is never intended to be publicly distributed. Adoption of best practices can be attributed to software development tools like GitHub and GitLab that support version

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

control, continuous integration testing, and documentation. Well established style guides, like PEP 8, also contribute to more uniformity in the software development approach. Integrating best practices into scientific software development requires allocating resources to make critical software design decisions and maintaining this focus throughout the project lifecycle. While it can be challenging to reserve funding for this purpose, the long-term benefits can far outweigh the initial cost. Once best practices become part of the research culture, they can be leveraged to benefit multiple projects, and used to establish a research community with a common software development style.

While software development standards for open source software packages are not always appropriate or realistic for scientific software, there is much to be learned from those examples. Open source software often prioritizes best practices over speed and efficiency. This relates to both the software development timeline, and the way the software is written. Developers might choose to write clean and readable code over a computationally faster implementation because the highest priority is the software's usability. This disproportionate focus on usability is part of what makes open source software unique. However, open source software faces its own challenges, like funding. While open source software projects have the potential for broader application and long-term impact, they have limited appeal in the short-term. Funding agencies are less inclined to commit to the long-term investment required to support an open source software package's continued maintenance and support. Application-based scientific software driven by a desire for quick results can be a more palatable investment for funding agencies. To reward best practices and secure funding to develop high quality scientific software, developers need methods to quantify the impact of their software. Similar metrics have been successfully adopted for data management through the FAIR (Findable, Accessible, Interoperable and Reusable) data principles [4].

Timeliness/Maturity

Source control, testing platforms, and style guides have been available for a long time, but the current abundance and transparency of open source software provides the scientific research community countless high-quality examples to guide their development. Adopting well established norms enables research teams to focus on custom features for their scientific software. It also ensures reproducibility which ultimately results in higher quality research. However, while the benefits of best practices are well established, developers must be incentivized to adopt them. Software does not have the journal equivalent of an Impact Factor, but such a quality metric is desperately needed. There is currently an abundance of data (e.g. code coverage, number of downloads, build status), but there is no reputable way to determine the Impact Factor of a software package. Without this knowledge, researchers have no way of evaluating the relative merit of one software package to another. Additionally, this will incentivize best practices because increasing the software's Impact Factor will be a necessary goal of the software, just as it is the goal of a journal.

References

[1] Goble, C., 2014. Better software, better research. *IEEE Internet Computing*, *18*(5), pp.4-8.

[2] Joppa, L.N., McInerny, G., Harper, R., Salido, L., Takeda, K., O'Hara, K., Gavaghan, D. and Emmott, S., 2013. Troubling trends in scientific software use. *Science*, *340*(6134), pp.814-815.

[3] Stodden, V., McNutt, M., Bailey, D.H., Deelman, E., Gil, Y., Hanson, B., Heroux, M.A., Ioannidis, J.P. and Taufer, M., 2016. Enhancing reproducibility for computational methods. *Science*, *354*(6317), pp.1240-1241.

[4] Wilkinson, M., Dumontier, M., Aalbersberg, I. *et al.* The FAIR Guiding Principles for scientific data management and stewardship. *Sci Data* **3,** 160018 (2016). https://doi.org/10.1038/sdata.2016.18

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Balancing innovation and stability of long-lived scientific software with containers

J. S. Sachdev (jsachdev@pppl.gov, Princeton Plasma Physics Laboratory)

Challenge: Scientific software originates from the need to study a specific problem in a specific domain. As this effort is accepted by the relevant community, the user base can increase rapidly from tens to hundreds of researchers. Clearly acceptance and use by the broad community should be the objective of every scientific software project; however, success can also lead to challenges that need to be considered. Bug fixes and model enhancements are updates expected by the users of the software. Appropriate use of continuous integration practices should ensure that new updates do not adversely affect the backwards compatibility of the code.[1] However, after the project achieves a certain level of maturity, the development team would likely be met by strong resistance by the user community to any thoughts of significant changes to the code base. This can be problematic for several reasons:

- New algorithms or libraries should replace their older counterparts but may not maintain strict backward compatibility
- Maintaining unneeded code (science or math libraries) or support for older hardware and compilers can be time-consuming and lead to significant clutter in the repository making it difficult to work with and adversely affect portability
- Scientific software is most often written with current or near future computing technologies in mind, porting to new computing technologies may require significant restructuring of the software base
- The development team is likely to lose their best developers to other projects if they are not given the freedom to innovate and improve their program; bringing on new developers will be difficult if the work is not interesting

To an extent, incrementing the major number of the semantic versioning of the software can help address these concerns but the larger issue is the relationship between the developers and users. The user community will prefer stability over innovation. If left unchecked, the codebase will stagnate and eventually lead the code to achieve legacy status at which point upgrades will be significantly more expensive to implement. This plateau in development will spur the creation of competing projects.

---

[1] By necessity some bug fixes negate backward compatibility. Full backward compatibility could be achieved by adding an input variable to use the buggy code. Although this should be considered bad practice, such things are not unheard of.

The challenge here is how to best balance the need to innovate within a major code base while appeasing the user's desire for stability. The scientific software relevant to the exascale project likely have not reached a stagnation point in terms of their development; however, the DOE has many scientific software programs that have been used for decades that are faced with these types of issues [1, 2].

Opportunity: One way this challenge could be addressed is by establishing best practices for containerization of scientific software. Modern tools for software development and quality assurance have significantly improved the productivity and effectiveness of scientific software development. Repository hosting sites, such as GitHub [3] and GitLab [4], provide effective collaboration tools for managing issues and pull/merge requests. Continuous integration and deployment methods, such as GitHub Actions, GitLab CI, and Jenkins [5], automate the testing and build processes. Appropriate use of these tools will improve the stability and quality of the project. However, the adoption of software container technologies in the scientific software community will be the most effective tool for maintaining a stable code for users while allowing the developers more freedom to innovate. Software containers package up the code and its dependencies enabling the application to run reliably and predictably on any computing environment and thereby allowing the user community access to older versions of the software without needing to recompile the code. Docker [6], Singularity [7], Shifter [8], and Charliecloud [9] are the most common containers used in HPC.

Timeliness: Software containers are less than a decade old and their adoption has been more widespread outside of the scientific software communities. Definition of a set of best practices for container usage would be very beneficial to the lifecycle of scientific software. The user community would have easy access to containers for every version of the code which would allow the developers more freedom to innovate.

References:

[1] https://tinyurl.com/58wd4xtd
[2] https://doi.org/10.6084/m9.figshare.7758305.v1
[3] https://github.com/
[4] https://about.gitlab.com/
[5] https://www.jenkins.io/
[6] https://www.docker.com/
[7] https://sylabs.io/singularity/
[8] https://github.com/NERSC/shifter
[9] https://hpc.github.io/charliecloud/

Revisiting Software Abstractions, Standards and Best Practices for A Fast Changing Scientific Computing Landscape

Sudip K. Seal (sealsk@ornl.gov), Oak Ridge National Laboratory, Oak Ridge, TN, USA

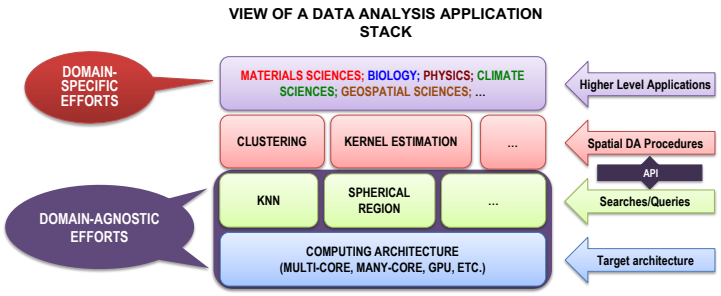
Challenge: State-of-the-art scientific computing today continues to be dominated mainly by traditional large-scale linear algebra-based use cases in physics, chemistry, fluid dynamics, molecular dynamics, climate simulations, etc. This article posits that the scope of scientific computing has evolved considerably beyond BLAS/LAPACK-based linear algebra-dominated applications and now includes a multitude of use cases in newer fields such as biology, microscopy, neutron diffractometry, geo-spatial sciences, mobility/social sciences and others, necessitating: (a) the identification of new, often hidden, application-independent abstractions beyond the well-established standardizations of linear algebra-based operations, and (b) the establishment of community standards and best practices to encourage independent standalone development of these new abstractions in the same footing as linear algebra. Such abstractions, though challenging to identify from a myriad of seemingly disparate applications, have the clear potential to (a) deliver high-quality sustainable architecture-aware functionalities for use by scientists in new non-traditional applications, and (b) deploy these new applications on emerging hardware(s) without the need to duplicate resource-intensive efforts for their efficient scalable architecture-aware implementations.

Identifying new abstractions and rallying the computational science community to recognize the importance of standards and best practices is challenging. For example, applications of fast multipole methods (FMM) in seemingly disparate application areas like cosmology, antenna design, molecular dynamics and neutron diffractometry rely on the same underlying computational data structures. The fundamental differences in the use of FMM from one application to another are in the mathematical kernels that describe the physics of the problem. The underlying tree data structure and the algorithms for associated tree operations like construction, traversal and accumulations that underpin all FMM-based computations are, however, independent of the mathematical details of these domain-specific kernels. Many excellent domain-specific implementations of FMM exist [1]. Yet, no *standards-based architecture-aware domain-agnostic* implementations of the core data structures and associated tree operations are available. Atom probe tomography [2], geospatial sciences [3] and electric-grid applications [4] routinely require two- and three-dimensional compute-intensive neighborhood search algorithms built on spatial data structures like octrees, compressed octrees, range trees, k-d trees, etc. for query-based analyses of massive data sets of spatial entities [5]. Yet, *standards-based architecture-aware domain-agnostic* scalable implementations of these fundamental spatial data structures are not available in the scientific computing community for ready use without the need for resource-intensive duplication of code development and scaling efforts.

Opportunity: Despite their late entry, new use cases from a host of non-traditional scientific computing areas (e.g., ML applications) were able to quickly scale up and across a variety of computing platforms, including leadership-class installations, benefiting immensely from the ready availability of robust, well-developed software, primarily in linear algebra, that were independently developed as *application-independent abstractions* in support of older *application-dependent* scientific computing use cases. The bottom line is that the linear algebra abstractions that were used to create robust architecture-aware implementations of core application-independent operations in support of older use cases fueled a new generation of scientific computing use cases in very short order. This is an important lesson learnt!

The concept of standards and practices for domain-agnostic software artifacts is not new. High-level linear algebra-based libraries like PETSc, Trilinos, SUNDIALS, etc. continue to benefit from the standalone developments of low-level linear algebra libraries like BLAS and LAPACK. Similarly, higher-level application-independent ML frameworks like PyTorch, TensorFlow, etc. are built atop cuDNN which implements linear algebra operations on GPUs. In all these cases, a well-defined demarcation separates the linear algebra operations from the higher-level frameworks or application-dependent procedures that call them. Natural demarcations that expose similar abstractions in today's more diverse and rapidly changing scientific computing landscape are not readily available but will need to be identified and agreed up on.

Historically, scientific methodologies have benefitted from establishing taxonomies or groupings of entities based on shared characteristics (particle classification in HEP, periodic table in chemistry, kingdoms/phyla in biology, etc.). Establishing a taxonomy of scientific computing methodologies based on computing characteristics is one way to extract application-independent abstractions from large classes of applications and enabling sustained development of these abstractions as independent software pipelines. There are many ways to categorize scientific computing applications. Presently, the community at large differentiates two main types of scientific computing – numerical and combinatorial scientific computing to distinguish applications based on matrix/vector-based linear algebra operations and those based on more irregular discrete data structures such as graphs, trees, etc. It is perhaps opportune to include more than the one axis to categorize scientific computing applications. For example, an additional axis for classification can consist of three categories of scientific computing applications – simulations (involves time advancements), analysis (involves computations of time-independent operators and functions) and optimizations (involves extremizing an objective function). Simulations can themselves be time-stepped or discrete event-based while analysis applications can be regular or irregular depending on whether they involve hierarchical or non-hierarchical computations and on the types of data/memory accesses, etc. The community must rally together to classify scientific computations into a bottom-up set of categories based on shared computational operations rather than a top-down approach guided purely by application-specific requirements and design.



Example: The adjoining figure illustrates a possible view of a scientific computing application stack for spatial data analysis. At the very top, domain-specific higher-level applications depend on spatial data analysis procedures like clustering, correlations, etc., which in turn are built atop fundamental search and query algorithms like range queries, k-nearest neighbor queries, shared nearest neighbor queries, etc. These operations, abstracted out from a wide variety of analysis procedures, form a rich space of data structures (octrees, range trees, kd-trees, etc.) and associated search/query algorithms. Once the category of *scientific computations for spatial data analysis* is identified, isolating the application-independent computational operations become possible and R&D efforts for scalable architecture-aware implementations of these core data structures and algorithms can be carried out independently by professional developers much like the lifecycle of BLAS/LAPACK development for numerical scientific computing applications. The analogy with numerical scientific computing in the adjoining figure would be to imagine PETSc/Trilinos in the second from the top layer (pink) and BLAS/LAPACK/ScaLAPACK/MAGMA routines in the third from the top layer (green).

Timeliness: The need to deploy new and non-traditional scientific computing applications, *quickly* and *gainfully*, on emerging computing architectures *without duplication of resource-intensive efforts* depends on our ability to evolve and streamline the scientific software development-and-use methodology to meet the demands of a rapidly changing scientific computing landscape. Efforts in this direction, such as those proposed here, become imperative in light of the imminent arrival of the exascale computing systems.

References:

[1] R. Yokota, "ExaFMM." https://github.com/exafmm/exafmm.

[2] M. Kühbach, P. Bajaj, H. Zhao, M. H. Çelik, E. A. Jägle, and B. Gault, "On strong-scaling and open-source tools for analyzing atom probe tomography data," *npj Comput. Mater.*, vol. 7, no. 1, 2021, doi: 10.1038/s41524-020-00486-1.

[3] J. Choi, B. Jang, and G. J. Kim, "Organizing and presenting geospatial tags in location-based augmented reality," *Pers. Ubiquitous Comput.*, vol. 15, no. 6, 2011, doi: 10.1007/s00779-010-0343-3.

[4] Y. Zhang, T. Huang, and E. F. Bompard, "Big data analytics in smart grids: a review," *Energy Informatics*, vol. 1, no. 1. 2018, doi: 10.1186/s42162-018-0007-5.

[5] H. Lu, S. K. Seal, G. Muzyn, W. Guo, and J. D. Poplawsky, "Efficient, parallel at-scale correlation analysis for atom probe tomography on hybrid architectures," 2018, doi: 10.1109/IPDPS.2018.00016.

Software as a Multi-Modality Knowledge Graph: Enhancing Scientific Software Development with AI-Capabilities

Edoardo Serra, edoardoserra@boisestate.edu
Boise State University and Pacific Northwest National Laboratory

Mahantesh Halappanavar, hala@pnnl.gov
Pacific Northwest National Laboratory

Scientists are facing many challenges when developing scientific code, especially in high performance environments. The scientific software is different from standard software in many ways: software quality testing rather than simply testing, tuning of parameters to avoid poor performance, high performance computing portability and debugging, and legacy software and underdeveloped code. Problems such as identifying bottlenecks, deadlocks, quality issues, and incompatibility with legacy libraries in scientific code are entirely demanded to humans, their understanding of the software, and their ability to debug the code (especially difficult in HPC). Such kinds of problems are timely expensive, and often, due to the complexity of the scientific code, finding solutions for such problems is not guaranteed.

Artificial intelligence is reaching today outstanding results in helping humans solving several problems. To accomplish similar achievements in the field of scientific code is required to transform the code in a format that is directly understandable by AI/ML systems. Graphs such as Control Flow Graphs (CFG), Data Flow Graph (DFG), and Abstract Syntax Tree (AST) are often used in software engineering to visualize and process the code [1, 2]. Today with the advent of graph representation learning, AI is accomplishing great success on graph data. However, it has been proved that only one simple kind of graph in software representation is not enough to properly represent a code with all its properties; this is even more appropriate with scientific code that has to be connected with specif infrastructure in order to be executed. Research effort should be directed on the definition of a more general form of graphs that is able to integrate all the different forms of software graph representations with in addition the capability of integrating not structured information such as text (e.g., to encapsulate available textual software documentation). We call such kind of graph generalization, a multi-modality knowledge graph, which integrates the multiple graph views of the software code, all the available textual documentation, and other standard data providing information about the execution ecosystem.

To develop AI-capabilities exploiting the full potential of the multi-modality knowledge graphs for software representation, effort should be directed on procedures for multi-modality knowledge graph representation learning, which are able to preserve in the condensed vectorial representations all the structural and non structural information contained in such knowledge graphs. Multi-modality knowledge graph representation learning procedures will serve as the bridge for ML-driven AI solutions helping the development of scientific code. Furthermore, effort should also be given in the interpretability of such ML-driven AI solutions with a special focus on the interpretation of the learned graph representations.

Scientific software challenges potentially solvable with AI AI tools should be able to address scientific software challenges by analyzing the software code without executing it. They should detect and locate scientific software issues by only scanning the code. In the following, we list some of the most common challenges of scientific software development [3]:

- **Testing software quality rather than standard software testing**. In scientific software, the focus is on identifying algorithmic defects (evaluating the algorithm's quality) rather than coding defects (errors in implementing the algorithm in the source code). It involves qualitative analysis to determine how the algorithm behaves, which is more challenging than finding and fixing coding defects (use of regression tests to catch any errors programmers introduce while modifying the code). Examples of such issues are: checking numerical stability, checking if certain quantities are exactly or approximately conserved, if symmetry properties hold, etc. A module that works correctly in isolation might behave unexpectedly when coupled with others.

- **Tuning the software because of poor performance.** Changes to the software architecture or to a particular subsystem may create a bottleneck.

- **Debugging.** Debugging tools such as Purify and Ensure are only working when running the program on a single processor. Debugging software to identify errors or identify deadlocks when the code run on multiple processors or distributed systems is hard.
- **Porting.** When a new machine appears, the porting of the software becomes a challenge. There is to need to identify what can the potential part of the code that can be affected by the use of a new machine.
- **Legacy Software and Underdeveloped Software** Legacy software libraries are omnipresent in scientific software and can cause a lot of instabilities. Detecting which legacy libraries or part of it are conflicting with others requires a lot of effort. Similarly, not underdeveloped software produce similar kind of issues, and detecting what to modify is crucial.

Multi-modality knowledge graph To make AI models working on scientific software, it is required to represent scientific software with flexible data structures able to represent all the software information. Graphs such as AST, CFG, and DFG are already employed to represent software code. However, these single graphs are not able to totally represent all the variety of information contained in the code alone. First, different kinds of graphs represent different kinds of information. Then, it is needed to merge all different kinds of graphs in a unique formalism that shows interconnections among the graphs. Second, a special graph modeling the parallel execution of such software should be created. Third, graph structures are not the only forms of information that can characterize software: methods, libraries, API calls, and variables can be paired with plain text descriptions obtained from the software documentation. Third, information characterizing the specific architecture used to run the software should also be provided. A way to encode all these aspects in a unique data structure is to use a knowledge graph that is a graph with labels on the edges and features attributes on the nodes. The labels on the edges can interconnect multiple kinds of graphs at the same time. While, feature attributes on the nodes can represent text descriptions of the different entities (e.g., libraries, API calls, variables, etc.) and the kind of architecture where specific parts of code are executed. The multi-modality is motivated by the heterogeneous information that can be integrated into such a knowledge graph.

Preliminary results on AI application and graphs Some of successful examples of AI applications and graph processing that can serve as a starting point for creating AI capability for scientific software development are provided in the following. AI applications using graph structure largely use graph representation learning procedures (e.g., SIR-GN [4] and GCN) generating vectorial representations of the node, edges and graphs. Such vectorial representations can be used in many machine learning models to predict execution performance, software execution anomalies, etc. Link prediction in a description graph is an example of how text description can be integrated into a graph concept, and it can be used to predict missing edges. [5] winner of the best application paper award use such technology to interconnect software vulnerability and weaknesses.

Conclusion Graph representation learning and multi-modality knowledge graphs can be the core technologies for AI applications enhancing the scientific software development process. However, to produce effective AI applications, research should focus on (i) ways to materialize multi-modality knowledge graphs from software code, (ii) methodologies for multi-modality knowledge graph representations, and (iii) interpretability procedures for machine learning models integrated with multi-modality knowledge graph representation techniques.

References

[1] S. M. Ghaffarian and H. R. Shahriari, "Neural software vulnerability analysis using rich intermediate graph representations of programs," *Information Sciences*, vol. 553, pp. 189–207, 2021.

[2] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 4, pp. 1–36, 2008.

[3] L. Hochstein and V. R. Basili, "The asc-alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, no. 3, pp. 50–58, 2008.

[4] M. Joaristi and E. Serra, "Sir-gn: A fast structural iterative representation learning approach for graph nodes," *TKDD*, vol. 15, no. 6, pp. 1–39, 2021.

[5] S. S. Das, E. Serra, M. Halappanavar, A. Pothen, and E. Al-Shaer, "V2W-BERT: A framework for effective hierarchical multiclass classification of software vulnerabilities," in *DSAA 2021, Porto, Portugal, October 6-9, 2021*. IEEE, 2021.

Future Challenges for High Energy Physics Software

The High Energy Physics Software Foundation (HSF)
Prepared by Elizabeth Sexton-Kennedy, Fermilab, sexton@fnal.gov;
Graeme Andrew Stewart, CERN, graeme.andrew.stewart@cern.ch;
Eduardo Rodrigues, U Liverpool, Eduardo.Rodrigues@cern.ch

Challenge

High Energy Physics (HEP) is one of the largest consumers of computing resources in the current landscape of scientific computing, with the experiments at the Large Hadron Collider (LHC) at CERN already curating an exabyte of data across the globe and using more than a million CPU cores continuously [WLCG Ref]. The sum of the codes which are run are of the order of 50 million lines (mainly C++, but also Python and some legacy Fortran) and much of this code is decades old, designed for a completely different era of microprocessor architecture.

The field faces significant additional challenges before the next major upgrade to the High Luminosity LHC, which will see event rates for the ATLAS and CMS experiments climb by about x10 and event complexity also rise with higher luminosity [HL-LHC Ref].

In the face of such challenges the HEP community has placed an increasing emphasis on commonly developed software, which was a founding motivation for the HEP Software Foundation [HSF Ref]. The HSF has been playing a key role in understanding the issues and motivating investment in software as a critical element of the scientific success of the field [CWP Ref].

Modern complex CPU architectures are now no longer seen as optimal for many workloads and the performance gains on different platforms, such as GPUs or FPGAs, have been much greater than on CPUs. Adapting our large code bases to this new hardware landscape often requires a fundamental redesign of algorithms from traditional serial implementations. The current set of C++ abstractions used for two decades in both the algorithmic and framework codes needs to be substantially rethought.

This rethinking of HEP software takes place in a social environment where training is frequently insufficient and experts are few and occupied frequently with the continued maintenance of the current system. In addition, the career path for research software engineers in HEP is very uncertain. Domain scientists who are attracted to this type of work are often not able to find their next position, and wind up leaving the field.

Opportunity

The significant changes that need to be made to HEP software are also an opportunity to adapt or rewrite codes to achieve better performance and better sustainability. Better performance comes from targeting modern architectures, paying heed to the data flow through the system from mass storage to processor registers; as well as rethinking algorithmic approaches to ones better suited to parallelism. Better sustainability arises from using new abstractions to separate the core physics tasks from their implementation on devices and avoiding older costly approaches, such as virtual function calls.

New tools and languages should be investigated that allow more natural expressions of the intent of physicists; that can check for correctness; that can target multiple backend architectures; and that can overcome validation problems when bit-for-bit results can no longer be guaranteed.

There is also a human opportunity to develop the next generation of physicists capable of developing the software needed to make experiments a success, instead of simply tweaking the solutions of the past.

Maturity

Advances in abstraction toolkits make such goals appear within reach, but there is a need for HEP codes, which are challengingly complex, to be more engaged with toolkit authors and engineering teams. Research into alternative approaches is happening, but they need to be given sufficient space to prove their promise and to build momentum. New HPC systems are arising and offering diverse platforms that can be used to test ideas in practice; when successful these can help satisfy the computing needs of the field, which will always be significant.

Overall, HEP codes are ripe with opportunity and provide a proving ground for new scientific approaches to software development and sustainability.

References

[WLCG Ref] https://home.cern/science/computing/grid
[HL-LHC Ref] https://home.cern/science/accelerators/high-luminosity-lhc
[HSF Ref] https://hepsoftwarefoundation.org
[CWP Ref] A Roadmap for HEP Software and Computing R&D for the 2020s, The HEP Software Foundation, Albrecht, J. et al., Comput Softw Big Sci (2019) 3, 7 https://doi.org/10.1007/s41781-018-0018-8.

Accelerate high fidelity physics simulations for fusion energy using machine learning technology

S. Shiraiwa (shirawa@princeton.edu), N. Bertelli (nbertell@pppl.gov)
Princeton Plasma Physics Laboratory / Princeton University, Princeton NJ

G. Wallace (wallaceg@mit.edu), J. C. Wright (jcwright@mit.edu)
Plasma Science Fusion Center/Massachusetts Institute of Technology, Boston MA

E. W. Bethel (ewbethel@lbl.gov) , Z. Bai (zhebai@lbl.gov), T. Perciano (TPerciano@lbl.gov),
R. Sadre (rssadre@lbl.gov)
Lawrence Berkeley National Laboratory, Berkeley CA

Overview:
This paper promotes the need of sustained effort to accelerate high fidelity physics simulations for fusion energy development. Fusion reactor has a unique complexity which presents challenges when developing and using scientific software: 1) a reactor is a nuclear facility and thus requires the highest level of security/safety, 2) physics parameter space, where a reactor is operated, will be outside the range achieved in any existing fusion experiments, 3) construction and operation period can extend over a long time (> 20 years). Therefore, from plant operation/control to the sensor/diagnostic data processing, software used in a reactor should be extremely robust, reliable and well-maintained. It is widely accepted that we need to develop an accurate computational model which can be used to predict the entire reactor operation and power plant performance.

Challenges: Computational model based on first principles and high geometrical fidelity are extremely expensive.

In fact, because it is too expensive to execute frequently enough, many integrated reactor simulations tend to employ less computationally expensive physics model ("reduced model"). Such a reduced model relies on a variety of approximations to simplify the physics problem and geometrical complexity. In the RF actuator research, a common approximation used for the wave propagation is the geometrical optics (Wenzel-Kramers-Brillouin) and a stratified plasma geometry. The codes like ANTITER [Messiaen 2021] , ALOHA [Hillairet 2010], GENRAY [Smirnov 1999] fall into this category.

It is noteworthy that AI/ML has been also used in combination with physics approximations described above to develop a reduced/accelerated simulation model. The approach is to use a large set of simulation output to train a network in order to reproduce the result quickly [Meneghini 2019].

In contrast to a reduced physics model, the fusion community has been developing so-called first principle models or high fidelity models which are much more computationally expensive to run, but they include more detailed and accurate physics models. Such expensive simulation models are being used to verify and to define the validity range of reduced models. In the RF research area, examples include the all-order spectral code for ICRF waves (AORSA) [Jager 2001] and the finite element based 3D full wave code (Petra-M) [Shiraiwa 2017].

Opportunities: Accelerate the high fidelity simulation using diverse ASCR research portfolio including AI/ML and GPUs

We emphasize that such a new endeavor must be done by a team with a very diverse background. We believe AI/ML can help / play a significant role. However, due to its high computational costs, an important aspect to consider is that a large training data set is difficult to assemble. Therefore, the natural alternative would be to use AI/ML to accelerate the numeral algorithms used inside the first principle codes. To tackle such a challenge, a team consisting of physicists, applied mathematics, and data scientists tightly working together would be necessary. A variety of code algorithms including linear solver, to mesh optimization could potentially benefit from such a team.

Another issue that the team must address is to ensure that the developed software will be maintained long enough. As discussed earlier, the lifespan of a fusion reactor is quite long, and it is well expected that development culture/style would evolve over time. This suggests that the team needs to include people from even more diversified backgrounds. Leading such a team to success could be a challenge.

TImeliness or maturity:

In fusion energy, internationally ITER is being built and its first plasma is approaching (2026), domestically the fusion community is undertaking the effort of fusion power plant design. Meanwhile, we are facing the fact that computing algorithms, hardware architecture, computing language and various research disciplines are getting more and more diversified and interconnected. Therefore, we believe that it is quite timely to start a team leveraging the full ASCR research portfolio to realize a step forward to simulate the entire fusion plant with full physics fidelity.

References

J. Hillairet *et al*., *Nuclear Fusion* **50**, 125010 (2010)

E.F. Jaeger *et al., Phys. Plasmas* **8**, 1573 (2001).

A. Messiaen, V. Maquet and J. Ongena, *Plasma Phys. Contr. Fusion* **63**, 045021 (2021).

S. Shiraiwa *et al., EPJ Web Conf.* **157**, 03048 (2017).

A. I. Smirnov and E. Y. Petrov, in Proceedings of the 26th EPS Conference on Plasma Physics and Controlled Fusion (European Physical Society, 1999), Vol. **23J**, p. 1797.

O. Meneghini et al., Nuclear Fusion **55**, 083008 (2015)

Cognitive Analytic Engines are Critical to Scientific User Facilities Reaching their Potential

Jeff Shrager[1*], Wan-Lin Hu[2], Paul Fuoss[2], Jana Thayer[2]
[1] xCures and Stanford Symbolic Systems Program (Adjunct)
[2] SLAC National Accelerator Laboratory
*Presenting author is Jeff Shrager: jshrager@stanford.edu, 650-380-6306

Challenge: Most high-performance settings, such as commercial aviation, the military, utility operations (e.g., nuclear plants), and even large scientific facilities, such as CERN, are relatively stable, operating in more-or-less the same configuration for extended periods, and strictly excluding inexperienced persons from participating closely in real-time operations. Remote operations and team structure in such settings are also generally well worked out and stable.

In contrast, Scientific User Facilities (SUFs), such as LCLS, regularly run experiments involving novel workflows and analyses, often employing novel instrumentation and controls. SUF teams frequently involve scientists with varying backgrounds and experience. Staff scientists (called "instrument scientists" at LCLS) have extensive experience with the hardware and software, but non-staff scientists (aka "users") have diverse backgrounds and experience levels. Moreover, for reasons of cost, travel complexity, or safety, users often participate in experiments remotely, leading to complex, variable, and often unstable real-time remote collaborations.

Making matters even more complex, rapid advances in instrumentation and computing capabilities are enabling SUFs to produce much more data much more rapidly. This makes efficient and correct decision-making in real-time operations, efficient error recovery, and correct real-time data triage even more critical.

Software designed for a high level of usability and learnability is at the core of SUF human-in-the-loop operations. Instruments are under constant modification, with novel experiments being run all the time, and the instruments are almost entirely manually controlled via software interfaces. Data is taken and analyzed through complex software pipelines that need to be managed so that the users know what data is where, and can find it again. Finally, as the rate of data production increases we are rapidly reaching the point where data must be triaged at the point of collection. In this case, real-time data analysis has to be carried out in order to decide which data is relevant and must be captured, which can be dumped, and, moreover, which measurements need to be repeated (or more data captured for them).

For all these reasons, software usability is critical. Moreover, it is especially important for the SUF case that the software be learnable, as users of varying backgrounds and experience, even if they do not have their hands directly on the controls(which sometimes they do), must interact in real-time, often remotely, with the instrument scientists and data analysts. Therefore, even if users needn't themselves learn what buttons to press and knobs to turn, they need to learn how to read the displays and analytical outputs in order to interact efficiently with the operators and analysts.

Of course, automation has an important role to play in improving the efficiency and correctness of high-performance real-time instrument operations. However, as we have seen with self-driving cars, and even with just the experience of commonplace cruise controls, far from removing the humans from the loop, automation puts even more pressure on humans to be able to work the difficult cases where the automation is unable to cope. Machine learning (ML) based automation makes this all-the-more

complex: When experiments change rapidly (for example, week-to-week at LCLS), and operators have varying degrees of skill and varying work practices, ML-based automation may learn to operate well with one experiment setting and/or operator, but then inappropriately transfer this learning to a new setting/operator, actually creating more problems that are even more difficult to resolve because of the simultaneous importance and inscrutability of the learned model.

Taken together, these factors make SUF operations unique and difficult settings in which efficient and correct human-in-the-loop operations are increasingly critical. And because instrument time is highly over-subscribed, and generally very expensive, even small improvements in operational efficiency can have a significant impact on scientific productivity.

Opportunity: Modern Cognitive Engineering (CE) methods, based upon over 50 years of cognitive science, can improve real-time experiment operations in challenging environments of this sort. Classical CE methods, such as interviews with instrument scientists and users, user studies, design reviews, and direct observation of operations, are useful, but these are inadequate to deal with the complexity of high-variability high-performance settings, as found in advanced SUFs. However, modern CE methods that employ cognitive simulations based upon well-validated theories of reasoning, attention, decision-making, performance, and learning, are capable of semi-automatically analyzing human-in-the-loop workflows and UI/UX designs, and highlighting potential inefficiencies and error modes in such settings. Such models can provide accurate response time and error rate predictions for very complex real-time control systems.

The technology underlying these methods is a Cognitive Analytic Engine (CAE) -- a joint cognitive/physics simulator, based upon a cognitive architecture, such as ACT-R [http://act-r.psy.cmu.edu/]. CAEs produce quantitative estimates of the efficiency of human-in-the-loop activities by simulating and analyzing workflows, UIs, and real-time data analysis and visualization tools. CAEs can also simulate and analyze analytic- and ML-based automation under various learning conditions, helping to head off problems associated with the introduction of these tools into highly variable settings. CAEs can also simulate error modes, error recovery activities, and UI/UX designs with respect to error recovery. Finally, CAEs can support "flight simulator"-like capabilities that enable testing and validation of advanced designs, and can directly support user training and remote collaboration.

Maturity and Timeliness: Cognitive Analytic Engines have been under development in the cognitive science community since the 1980s, and are, at this point, well-developed, evidence-based technologies. They have been deployed to positive effect in many high-performance industrial and government settings including nearly every branch of the military, commercial aviation and space flight, utility operations (e.g, nuclear power plant operations), and many others. Less common is the application of CAEs in facilities such as SUFs, where the setting is rapidly changing (e.g., a new experiment each week at LCLS), where the users have a wide range of experience and backgrounds, and where team structure and collaboration is highly variable. Yet we believe that this is easily within reach of these technologies.

It is our position that as user instruments become more and more capable and complex, cognitive engineering and, in particular, Cognitive Analytic Engines will serve a critical role in realizing the enormous potential of these facilities.

Social science research is essential to the future of the scientific software enterprise at Department of Energy laboratories

Benjamin Sims, Los Alamos National Laboratory (bsims@lanl.gov)
Elaine M. Raybourn, Sandia National Laboratories (emraybo@sandia.gov)
Reed Milewicz, Sandia National Laboratories (rmilewi@sandia.gov)
David Moulton, Los Alamos National Laboratory (moulton@lanl.gov)
Will Sutherland, University of Washington (willtsk@uw.edu)

Challenge

Social science research is essential to our ability to anticipate and respond to the significant human and organizational challenges of future scientific software development efforts. These challenges include: continued growth in the scale and ambition of scientific software projects, new levels of social complexity in scientific software work driven by the need to integrate activities across domains, and a growing and diversifying scientific software workforce. An increasing number of dedicated scientific software professionals—for example, research software engineers (RSEs)—have careers that may not readily fit into existing PhD-oriented research career paths and reward systems [1, 2]. These issues are particularly relevant to Department of Energy (DOE) laboratories because scientific software projects in these institutions run the gamut from small, PI-driven projects to very large institutional efforts. In light of these challenges, we propose that a scientific approach to the development and use of scientific software must include a strong social science component.[1]

Opportunity

The opportunity here is to build on emerging efforts to understand scientific software development as a social phenomenon, both inside and outside of DOE. This will develop capabilities to enable DOE and other institutions to address the human and organizational challenges of future scientific software initiatives. It will also allow for more systematic approaches to understanding and defining best practices, roles, learning processes, teaming and organizational strategies, and workforce issues in scientific software development.

In order to understand the complex, multifaceted socio-technical phenomenon of scientific software development, we propose a *mixed methods* approach. *Qualitative methods* such as interview- or observation- based studies are ideal for identifying and gaining in-depth understanding of key issues. *Quantitative methods*, such as analyzing survey, workforce, or software repository data, can provide broader and more objective views. We also advocate *participatory research* that incorporates input from relevant communities at all stages. Examples of potential research directions include: observing or interviewing software teams to better understand work processes and how they might be scaled to a larger, more integrated enterprise [3]; interviewing or surveying software professionals to better understand their work

[1] Cognitive and behavioral sciences are also highly relevant here but are outside our scope for this paper.

and the potential relevance of roles like RSE; or analyzing publications and software repositories to identify collaboration patterns that contribute to increased scientific productivity [4].

Several authors of this paper represent a growing network of social scientists and other researchers who have begun to explore the issues and methods described here in the context of the Exascale Computing Project (ECP) [3-4]. Another represents the growing community of researchers in this area at academic institutions. We propose to continue building ties between these communities through collaborations, workshops, and other outreach activities.

Timeliness or maturity

This is an auspicious time to develop a research agenda in the social and human aspects of scientific software in DOE, for several reasons. First, there is an increasingly active research community in this area, which has produced a number of relevant studies that provide a starting point for further research [4-6, for example]. Second, DOE's interest in integrated, multi-institutional projects like ECP creates unique opportunities for studying these issues. Third, to address emerging challenges, institutions need guidance on best practices for organizing and supporting software projects and teams. Finally, scientific software professionals are increasingly seeking to define roles, career paths, and reward structures, efforts which social science research can support.

Success in these efforts will advance the state of the art in understanding the social aspects of scientific software. This will enable DOE laboratories and other institutions to more efficiently disseminate best practices, develop effective teams, and produce high-quality software in an increasingly complex software environment. It will also enable more effective support of the scientific software workforce, improving morale, recruitment, and retention.

References

[1] J. Cohen, D.S. Katz, M. Barker, N.C. Hong, R. Haines and C. Jay, "The four pillars of research software engineering," *IEEE Software*, vol. 38, no. 1, pp. 97-105, 2020.

[2] N. Berente, J. Howison, J. Cutcher-Gershenfeld, J.L. King, S.R. Barley, and J. Towns, "Professionalization in cyberinfrastructure," 2017. Available at https://ssrn.com/abstract=3138592.

[3] E.M. Raybourn, J.D. Moulton, and A. Hungerford, "Scaling productivity and innovation on the path to exascale with a 'team of teams' approach," in Nah and Siau (eds) *HCI in Business, Government and Organizations: Information Systems and Analytics, Proc. HCII 2019*. Lecture Notes in Computer Science, vol. 11589.

[4] E. Raybourn, R. Milewicz, D. Rogers, B. Sims, G. Watson, E. Gonsiorowski, and J. Willenbring, "A Data-Driven Approach to Rethinking Open Source Software Organizations as a Team of Teams," *CSCE Conf. on Software Engineering Research and Practice*, 2021. (To be published in proceedings.)

[5] Chen, L. Ramakrishnan, S.S. Poon, and C. Aragon, "Harnessing complexity in high performance computing ecosystems: A complex adaptive systems framework," in *Proc. 52nd Hawaii Int'l. Conf. on System Sciences*, 2019, uri: https://hdl.handle.net/10125/60065.

[6] J. Howison, E. Deelman, M.J. McLennan, R. Ferreira da Silva, and J.D. Herbsleb, "Understanding the scientific software ecosystem and its impact: Current and future measures," *Research Evaluation*, vol. 24, no. 4, pp. 454-470, October 2015.

Long-Term Productivity Based on Science, not Preference

Spencer Smith
McMaster University
Hamilton, Canada
smiths@mcmaster.ca

Jacques Carette
McMaster University
Hamilton, Canada
carette@mcmaster.ca

Our goal is to identify inhibitors and catalysts for productive long-term scientific software development. The inhibitors and catalysts could take the form of processes, tools, techniques, environmental factors (like working conditions) and software artifacts (such as user manuals, unit tests, design documents and code). The effort (time) invested in catalysts will pay off in the long-term, while inhibitors will take up resources, and can lower product quality.

Developer surveys on inhibitors and catalysts will yield responses as varied as the education and experiential backgrounds of the respondents. Although well-meaning, responses will predictably be biased. For instance, developers may be guilty of the *sunk cost fallacy*, promoting a technology they have invested considerable hours in learning, even if the current costs outweigh the benefits. Likewise developers may recommend against spending time on proper requirements, not as an indication that requirements are not valuable, only that current practice doesn't promote requirements [2]. Another perceived inhibitor is time spent in meetings. For instance, the lack of visible short-term benefits renders department retreats unpopular, even though relationship building and strategic decision making may provide significant future rewards. Evaluating the usefulness of meetings is difficult. Rather than relying on preference and perception, as these examples illustrate, *we need to measure the long-term impact of development choices to make wise ones.*

1 BUILDING BLOCKS

A scientific approach requires a solid foundation. The building blocks for scientific discourse are: communicating concepts via an unambiguous language, formulating hypotheses, planning data collection, and analyzing models and theories. To start with, we need to classify the software under discussion. Likely dimensions include: general purpose scientific tools versus special purpose physical models, scientific domain, open source versus commercial software, project maturity, project size, and level of safety criticality.

We also need to be precise about our software quality goals. Qualities such as reliability, sustainability, reproducibility and productivity need precise definitions. Attempts have been made since the 1970s [7], but the resulting definitions aren't usually specific to scientific software (as shown by the confusion between precision and accuracy is the ISO/IEC definitions [4]). Moreover, the definitions often focus on measurability, where the first priority should be conceptual clarity, analogous to the unmeasurable, but conceptually clear, definition of forward error, which requires knowing the (usually unknown) true answer.

For each relevant quality we recommend collecting as many distinct definitions as possible. Once collected, they can be assessed against the following criteria (based on IEEE [3]): completeness, consistency, modifiability, traceability, unambiguity and abstractness. The understanding gained from this systematic survey and analysis can be used to either choose solid definitions, or propose new ones. In all cases, the definitions should enable *reasoning about quality*.

2 PRODUCTIVITY

Our definition of long-term productivity [9] provides an example of our vision, and meets our criteria. We define productivity as:

$$P = O/I$$

$$I = \int_0^T H(t)\ dt$$

$$O = \int_0^T \sum_{c \in C} F(S_c(t), K_c(t))\ dt$$

where P is productivity, I is the inputs, O is the outputs, 0 is the time the project started, T is the time *in the future* where we want to take stock, H is the total number of hours available by all personnel, C represents different classes of users (external as well as internal), S is *user satisfaction* and K is *effective knowledge*, and F is a weighing function that indicates "value". Thus productivity is measured in "value per year." and is a mixture of external and internal value produced. *Value* should not be equated with money; measuring the productivity of free software development is just as important as for commercial software.

While the most straightforward use of such a formula is to measure productivity of a team, it can also be used in "what if" scenarios to assist in planning interventions, i.e. changes intended to improve productivity.

Measuring over too short a time-frame will assuredly give warped results. This leads some to argue that productivity shouldn't even be measured [5].

3 MEASURING

Proper science requires measurement. We can only determine whether a given intervention is a catalyst or inhibitor by measuring its impact. Let us examine in more details the consequences of our proposed definition.

First, the time integrals emphasize that productivity is something that happens *over time*. The most interesting kind of productivity is that of an organization over the span of years. Measuring over too short a time frame is one of the main sources of *technical debt* [6] as it devalues planning, team work, being strategic, etc.

Secondly, as Drucker [1] reminds us, quality is at least as important as quantity. Here we use a proxy for quality, namely *user satisfaction*. It is important to note that unreleased products and unreleased features induce **no** user satisfaction. A broken product might be even worse, and produce negative satisfaction.

The input H is the number of hours worked by the team, including managers and support staff, as appropriate. To optimize

productivity, we want to make I, and thus H, *small*. This is the raw input being applied, whether effective or not.

We use user satisfaction (S) as a proxy for effective quality. How to measure this is left for future study. It can be approximated by measures such as numbers of users, number of citations, number of forks of a repository, number of "stars", surveys of existing users, number of mentions in the issue tracker, and usability experiments.

Probably the trickiest part is *effective knowledge* (K). The idea is that while source code embodies operational knowledge that has the potential to directly lead to user satisfaction, a project usually also generates a lot of *tacit knowledge* about design, including the rationale for various choices. This is the kind of knowledge that is lost when employees leave, and is the most costly to build and replace. In other words, human-reusable knowledge such as documentation factors in here. The best measure for knowledge is an area for future exploration.

4 ARTIFACTS PRODUCED

Software development typically produces many artifacts, such as requirements, specifications, user manuals, unit tests, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. We regard all of these as containing *knowledge*, albeit encoded in different forms. Furthermore, it is crucial to recognize that the knowledge of a single product is *distributed* amongst those artifacts. In particular, the various artifacts contain many copies of the same core knowledge — by design.

To understand the importance of certain artifacts, it makes sense to look at the productivity impact of their presence/absence. For example, long-lived projects will inevitably encounter contributor turnover. How long should it take for new contributors to be productive? How much training by peer mentors will it take? Could some documentation be written that would shorten this learning period and, just as importantly, reduce the time it takes from experienced people? Of course, documentation that is out-of-date could be even worse: a false sense of knowledge that results in even more wasted work that needs repairing.

As we gain understanding on measures of value, we can use them to evaluate the state of practice in different research software domains. We can estimate the knowledge K embedded in, and the user value S derived from, existing artifacts. In particular, we can compare these to the artifacts produced by recommended processes from standard software engineering textbooks. For example, we can test the hypothesis that knowledge duplication between code and requirements, coupled with the fact that requirements get desynchronized from the code and the tenuous link to user value, is the likely reason for low adoption of requirements in scientific software development [2].

Nevertheless, documentation remains useful, especially for the very long term. Another means to judge the utility of documentation is to look at assurance cases. An assurance case [8] presents an organized and explicit argument for correctness (or whatever other software quality is deemed important) through a series of sub-arguments and evidence. Assurance cases gives at least one measure of which documentation is relevant and necessary.

5 PRODUCTION METHODS

One way to improve productivity is to waste less on non-productive or counter-productive activities. That code is the most visible artifact that contributes user-value, along with with testing (because quality is an extremely important factor in user-value) explains the inordinate focus on just those artifacts. Furthermore, the deemphasis on documentation, even to the extreme of some methodologies having none, can feel like productivity improvements in the short term! A better approach would be to capture knowledge in ways that keeps it continuously synchronized between the various artifacts where it appears.

One promising approach is to generate all artifacts from a single knowledge base [10]. This relies on a solid understanding of the contents of all of the artifacts present in the software engineering process. Our proof-of-concept shows that this is possible. As the artifacts are now generated, knowledge duplication is not a problem. Even better, the knowledge is synchronized-by-construction. Furthermore, it becomes easy to tailor artifacts, documentation as well as code, to different classes of "users".

6 CONCLUDING REMARKS

Our position is that decisions on processes, tools, techniques and software artifacts should be driven by science, not by personal preference. Decisions should not be based on anecdotal evidence, gut instinct or the path of least resistance. Moreover, decisions should vary depending on the users and the context. In most cases of interest, this means that a longer term view should be adopted. We need to use a scientific approach based on unambiguous definitions, empirical evidence, hypothesis testing and rigorous processes.

By developing an understanding of where input hours are spent, what most contributes to user satisfaction, and how to leverage knowledge produced, we can determine what has the greatest return on investment. We will be able to recommend software production processes that justify their value because the long-term output benefits are high compared to the required input resources.

REFERENCES

[1] P.F. Drucker. 1999. Knowledge-Worker Productivity: The Biggest Challenge. *California Management Review* 41, 2 (1999), 79–94.

[2] Dustin Heaton and Jeffrey C. Carver. 2015. Claims About the Use of Software Engineering Practices in Science. *Inf. Softw. Technol.* 67, C (Nov. 2015), 207–219. https://doi.org/10.1016/j.infsof.2015.07.011

[3] IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286

[4] ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC.

[5] Amy J. Ko. 2019. *Why We Should Not Measure Productivity.* Apress, Berkeley, CA, 21–26. https://doi.org/10.1007/978-1-4842-4221-6_3

[6] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

[7] J. McCall, P. Richards, and G. Walters. 1977. *Factors in Software Quality.* NTIS AD-A049-014, 015, 055.

[8] David J. Rinehart, John C. Knight, and Jonathan Rowanhill. 2015. *Current Practices in Constructing and Evaluating Assurance Cases with Applications to Aviation.* Technical Report CR-2014-218678. National Aeronautics and Space Administration (NASA), Langley Research Centre, Hampton, Virginia.

[9] Spencer Smith and Jacques Carette. 2020. Long-term Productivity for Long-term Impact, arXiv report. https://arxiv.org/abs/2009.14015. arXiv:2009.14015 [cs.SE]

[10] Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE).* Austin, Texas, United States. 4 pp.

Improving Development Practices through Culture and Collaboration

Vanessa Sochat David Beckingsale Robert Neely

Lawrence Livermore National Laboratory
Livermore, CA

Challenge

Scientific software development is becoming increasingly more complex in both high performance computing (HPC) and cloud computing environments. As a result, software developers are increasingly tasked with creating practices to develop, test, and collaborate across communities and environments, and disagreement in these practices can create friction. Greater understanding combined with comparison against other communities is needed to further innovate and improve upon current practices.

As a motivating example, we can look at challenges with continuous integration (CI). Given the limited availability of resources, many groups use a potpourri of free and open source solutions to version control their code, run tests in the cloud, and perform code reviews. These resources are typically distinct from the institutional resources where they ultimately might be used. This diversity of practices and tools introduces problems that other communities simply do not have, such as needing package managers to install across environments, testing on an increasing number of niche platforms, and extensive use of time to debug when things do not work. Further, the production environments are tightly controlled, making it harder to experiment with potential new developer workflows. In contrast, in industry it is common to have more controlled environments and a greater access to standard development environments and tools [2].

How can we develop a community and culture to foster modern development workflows in the unique area of scientific software development?

Opportunity

The growing complexity of the scientific software development process and culture gives us an opportunity to collaborate, build community, and identify best practices and areas for growth. We have identified a number of key questions that will allow us to accurately map this space:

- Who are the key stakeholders in this landscape, and what are their roles? For example, the role of a research software engineer at an academic institution differs from that of a researcher at a national laboratory. However, there will be key areas of overlap in their daily software development processes.
- What are the different kinds of scientific software development? The needs of lower-level software development such as operating systems, runtimes and job schedulers are different from that of domain-specific simulation software.
- What tools are currently being used?
- How do practices and tools compare between development and production environments? Due to numerous constraints, development and testing are typically performed on resources distinct from those used to run the software in production.
- How do practices in scientific software communities compare with industry, and where do academic/HPC communities have room to improve? Which problems are specific to HPC and are they avoidable (problems we've created for ourselves)?
- How do we create an environment where researchers can experiment with ideas in this space? Prototyping new ideas for development in the HPC space can be difficult due to security concerns

LLNL-PROP-829310

meaning the average user has very limited privileges on the systems they are using. A shared, robust testing environment would allow researchers to make faster progress identifying solutions.

- To what extent do leadership computing facilities build-in support for testing on their production systems, or provide dedicated test and development systems for developers to apply CI and testing? Can we quantify the benefits of these policies to the user community and make the case for dedicated resources for the developer community?

- How can we securely enable CI in computing facilities to not only allow internal contributors to test proposed changes, but also allow accepting pull requests from community contributors?

With the establishment of the United States branch of the Research Software Engineer (RSE) community in 2018, we have a network of professional RSEs across academic institutions, national labs, and industry[1]. This network presents us with an opportunity to better collaborate on many of the problems in this space and empower developers to innovate. Given the hybrid nature of working environments across cloud and HPC, DevOps practices are logically being adopted in scientific software development. We see the need for ASCR to engage the community defined above to study this problem, specifically studying problems specific to developers in the HPC and RSE communities. Some of these unique challenges include: increasing utilization of containers and cloud practices in HPC environments, more easily building research software despite limited permissions, and quickly identifying performance regressions in any software being tested. Studying this problem methodically will require collecting data to understand tools and practices that are used, both within the academic and industry-oriented developer communities, and comparing within and between groups to identify areas for improvement. By engaging with the research software engineering community, we have an opportunity to engage the broader community and develop best practices for sustainable scientific software development.

Timeliness

Addressing the research engineering operations challenge is timely due to the growing diversity of software and hardware resources being applied to HPC problems. It is essential that scientific software developers find a balance between supporting the complex needs of high-performance computing without being burdened by disorganized development practices. By accurately defining the participants and the research and development workflows being employed, we are better able to understand where we can leverage best practices from industry, and where we as a community must innovate to define our own best practices. If we define RSE-ops as a community and culture oriented around developing practices, resources, and tools for the research software engineering community akin to Dev-Ops, RSE-ops give us an opportunity to engage the broader community and develop best practices for sustainable scientific software development. Embracing DevOps practices presents an opportunity to not just engage with the larger research software engineering community, but also to save the DOE money, time, and resources if it's done right.

References

A not-so-brief history of research software engineers. (n.d.). `https://www.software.ac.uk/blog/2016-08-17-not-so-brief-history-research-software-engineers-0`. (Accessed: 2021-8-14)

Potvin, R., & Levenberg, J. (2016, jun). Why google stores billions of lines of code in a single repository. *Commun. ACM*, *59*(7), 78–87. Retrieved from `https://doi.org/10.1145/2854146` doi: 10.1145/2854146

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Combining Repository & Process Mining for Better Scientific Software

Thorsten Sommer*

November 19, 2021

1. Challenge

The typical development of scientific software begins as a prototype for own needs, or those of the own organizational unit. For such prototypical development, developers often forgo recognized techniques from professional software development, such as requirements analysis, user stories, documentation of use cases, development of an appropriate architecture, and consultation with potential users. This leaves issues such as IT security, quality, usability, maintainability and scalability out of consideration. Complicating matters further, scientific software is often developed by scientists from all disciplines without the necessary training and support (cf. Haupt et al., 2018). At the same time, scientists working as software developers cannot be expected to become computer scientists in parallel. After all, the challenges in computer science are so complex that computer scientists spend a lifetime working and researching on them every day. In consequence of these conditions, so-called "technical debt" arises, see also Besker et al., 2019; the following four symptoms (S1 to S4) might result from the described situation.

S1: IT Security

During early development, IT security is deliberately ignored. The initial aim is to demonstrate feasibility or to solve a problem under high time pressure. Years later, the program continues to be used without the security aspects ever having been checked and implemented (cf. Siavvas et al., 2020). Ultimately, this leads to the overall IT security of the research organization being weakened.

S2: Oracle Problem

In early development, quality was laid out for the one immediate use case that needed to be solved. Under the conditions of this first use case, the software has always worked and produced correct results. After years of use, the software finds more users and gradually becomes a product. The original developer is no longer part of the organization, therefore the knowledge of how the software works has been lost; the software is now a black box for most users. The quality and correctness for the new conditions is and was not checked, so possibly unnoticed wrong results are produced (cf. Vogel et al., 2019). This issue is called the "oracle problem" and "large variability", see Vogel et al., 2019.

S3: Productivity

Where quality was not given sustained attention in scenario S2, the focus here is on usability. The software was developed for the initial task. Exactly for this purpose the software is usable ideally. Over time, more scientists, also from other disciplines, work with it. This gradually changes the use case. This can lead to lower productivity compared to the original use case (cf. Macaulay et al., 2009).

S4: Blackbox

For most scientists, universities are transit stations on their careers. In countries such as Germany, the maximum duration of stay is even regulated by law, for example by the "Wissenschaftszeitvertragsgesetz" of 1999. Under these conditions, the prototypical development described at the beginning is a problem that becomes apparent in the medium and long term. During the initial development, no or inadequate documentation is produced. In the best case there are publications about the software. Years later, after the original developers are no longer part of the organization, the knowledge about the implementation is lost. The scientific HPC community has already recognized this issue for themselves and is trying to counteract it (Anzt and Flegar, 2019). However, this issue also occurs in a similar form when scientists spend a lifetime developing software and then retire. Further development becomes expensive or even impossible in both cases.

2. Opportunity

Nowadays, a common method to make these challenges manageable is the introduction of software development guidelines (cf. Haupt et al., 2018). Three of the four symptoms described here are thereby mitigated, but not eliminated. In particular S2 cannot be mitigated in this way, since the initial developer could not know which future use cases would occur. This is similarly true for

*thorsten.sommer@dlr.de; ORCID 0000-0002-3264-9934; German Aerospace Center (DLR), Institute for Software Technology, Intelligent and Distributed Systems Group

the other three symptoms. For example, no developer can know what IT security attacks unknown today will occur in the future.

In support of the software development guidelines, I therefore propose the development of a generic framework that uses repository and process mining to interactively support developers. Repository mining uses data science methods to, for example, derive information about the software development process (cf. Kurnatowski et al., 2020). Hence, similarity to other projects (El Baff et al., 2021) and IT security vulnerabilities (Schreiber et al., 2021) can be determined automatically.

Process mining can be used to derive information about processes and thus also about their use. In the software domain, there is the approach of working with telemetry data. The software under investigation sends user events to a server. By evaluating these user events, insight can be gained into the use of software and the emerging issues, for example in the area of usability.

Combining these approaches results in a solution that scales to any number of scientific software and can be applied to both new and legacy software. Symptom S1 can thus be completely eliminated. In the case of S2, telemetry can at least detect when new use cases make inputs that potentially lead to incorrect results so that scientists can be warned. Likewise, the loss of productivity in the case of S3 can be detected so that new requirements can be assessed and the software can be adjusted in terms of its usability. In the sense of the black box in S4, repository mining can be used to recognize retrospectively how the software was created step by step in order to gain knowledge about its implementation. Additionally, an unknown software architecture can be visualized in such a way that the viewer can explore the architecture to improve its understanding (cf. Schreiber and Misiak, 2018).

In order to achieve this goal, the framework to be developed would have to be able to handle the most common programming languages, such as C, C++, C#, Python and Java. The repository mining part of the framework must be able to analyze these languages in order to identify e.g. IT security vulnerabilities and programming flaws. For the process mining aspect, the framework needs an embedded web server. For maximum interoperability, language-neutral interfaces would then be provided using the OpenAPI standard, allowing integration into new and existing scientific software with little effort. If such a generic framework would be maintained as an open source solution by the scientific community, it could establish itself as a standard for scientific software development in the future.

3. Timeliness

Approaches such as repository and process mining are relatively recent. In the meantime, enough experience exists so that the framework considered here can be implemented. In addition, thanks to open source, the worldwide software engineering ecosystem has now reached a state in which not only the own code but almost all of its external dependencies can be analyzed by means of repository mining, so that a holistic overall picture can obtained with the approach presented here.

4. References

Anzt, H., Flegar, G., 2019. Are we Doing the Right Thing? — A Critical Analysis of the Academic HPC Community, in: 2019 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW). Presented at the IEEE Intl. Parallel and Distributed Processing Symposium Workshops, IEEE, Rio de Janeiro, Brazil, pp. 739–745. DOI: 10.1109/IPDPSW.2019.00122

Besker, T., Martini, A., Bosch, J., 2019. Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work. Journal of Systems and Software 156, 41–61. DOI: 10.1016/j.jss.2019.06.004

El Baff, R., Santhanam, S., Hecking, T., 2021. Quantifying Synergy between Software Projects using README Files Only, in: Proceedings of the 33rd Intl. Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute Graduate School, Pittsburgh, USA. DOI: 10.18293/SEKE2021-162

Haupt, C., Schlauch, T., Meinel, M., 2018. The Software Engineering Initiative of DLR - Overcome the obstacles and develop sustainable software. Presented at the ACM/IEEE Intl. Workshop on Software Engineering for Science, ACM, Gothenburg, Sweden, pp. 16–19.

Kurnatowski, L. von, Stoffers, M., Weigel, M., Meinel, M., Wasser, Y., Rack, K., Fiedler, H., 2020. Scientific Software Engineering: Mining Repositories to gain insights into BACARDI, in: 2020 IEEE Aerospace Conference, AERO 2020. Presented at the IEEE Aerospace Conference, IEEE, Big Sky, MT, USA, pp. 1–10.

Macaulay, C., Sloan, D., Jiang, X., Forbes, P., Loynton, S., Swedlow, J.R., Gregor, P., 2009. Usability and User-Centered Design in Scientific Software Development. IEEE Software 26, 96–102. DOI: 10.1109/MS.2009.27

Schreiber, A., Misiak, M., 2018. Visualizing Software Architectures in Virtual Reality with an Island Metaphor, in: Chen, J.Y.C., Fragomeni, G. (Eds.), Lecture Notes in Computer Science. Presented at the VAMR 2018: Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation, Springer Intl. Publishing, Las Vegas, USA, pp. 168–182.

Schreiber, A., Sonnekalb, T., Heinze, T., Kurnatowski, L., Gonzalez-Barahona, J.M., Packer, H., 2021. Provenance-Based Security Audits and Its Application to COVID-19 Contact Tracing Apps. Lecture Notes in Computer Science 12839, 88–105.

Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Tzovaras, D., 2020. Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. Enterprise Information Systems 1–43. DOI: 10.1080/17517575.2020.1824017

Vogel, T., Druskat, S., Scheidgen, M., Draxl, C., Grunske, L., 2019. Challenges for Verifying and Validating Scientific Software in Computational Materials Science. 14th Intl. Workshop on Software Engineering for Science, IEEE, Montreal, Canada, pp. 25–32. DOI: 10.1109/SE4Science.2019.00010

Formal reasoning to improve trust for scientific applications

Matthew Sottile[1], Markus Schordan, Dan Quinlan
Lawrence Livermore National Laboratory

Challenge:

Verifiability of trustworthy code in scientific computing requires addressing multiple concerns typically treated separately by the software assurance community. Trust of scientific code is broader than security. A trustworthy code must be correct with respect to the domain problem that it models; it must be resilient in the presence of disruptions or data corruption; and it must be secure with respect to user-defined policies around information protection and isolation. Logic-based formal methods are the only known approaches to gain assurance of trust that is mathematically rigorous such that statements about trust can be formally proven. These approaches also raise the process of reasoning about programs from an ad-hoc process to one in which the scientific method can be applied to analyzing and predicting what programs do.

Barriers To Adoption: Techniques from formal methods can provide high assurance of software artifacts by using a foundation of mathematical logic to express correctness properties and the analysis methods to verify them. Even though these methods have existed for nearly 50 years, they are not in widespread use and justifiably viewed with skepticism by potential users due to several factors. A key limiting factor in user uptake is the specification problem [1]. Application-level specifications of program logic must be encoded at a relatively low level of abstraction. They require specifications to be encoded using a language outside those which developers write their programs, often requiring knowledge of specialized constructs and type systems that are not present in mainstream languages. In many ways, the state of formal methods resembles general purpose programming using machine code in the early days of computing: full of promise but limited to a small set of specialists who must have intimate knowledge of the underlying system and low-level specification languages. We believe research is required to reduce this gap between formal specifications and the languages used to implement scientific programs to broaden their adoption.

Opportunities:

Integration with conventional languages: Formal specifications are written in the language of mathematical logic. The use of logic is critical for the verification process: established sound proof techniques can be applied to deduce whether or not properties of code hold. The only formal verification technique that has found widespread success bridging the gap between formal logic and conventional languages are type systems. Programmers routinely take advantage of type analysis without realizing that they are applying a specific form of formal verification to their code. Applying more sophisticated formal analyses via techniques like as

[1] Corresponding author: sottile2@llnl.gov. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Design-by-Contract would allow verification that code conforms to high-level behaviors that are beyond the scope of type systems alone. Deeper specification of types that capture the semantics of domain-level data structures is necessary for tools to reason about data properties beyond the basic representational analysis facilitated by types.

Hierarchical and compositional specification: Specifications must be hierarchical and composable to address the scale and complexity of HPC scientific programs – compositional design in conventional languages was key to increasing the complexity of programs that people can write and will be key to building complex specifications. It is critical to recognize that different formalisms are appropriate for different classes of properties and verification task [2]. Model checking of state transition systems may be the most appropriate model for reasoning about concurrency, control, and data flow, while classical deductive reasoning methods may be appropriate for reasoning about the mathematical structures that a code implements. Composition will entail not only artifacts from a single formal modality, but artifacts from different modalities that each best reflect different aspects of a single system.

Solver scalability: Formal specifications and the corresponding proofs that properties hold are often very large. The computational complexity of automated algorithms typically does not scale well, nor can it take advantage of HPC resources. Users must be able to generate analysis results in a reasonable amount of time for realistic programs. Solving this problem requires more than just a larger computer: the scalability of formal analysis tools is more often due to the asymptotic complexity class of the algorithms themselves and how users express their problems. Providing users tools to select appropriate modeling modalities as well as efficient encodings of their problems will prevent avoidable complexity.

Timeliness:

Many of the foundational technologies for formal program reasoning have improved significantly in recent years. This has been driven by widespread penetration of technology in everyday life, where software defects can cause significant problems for the privacy, safety, and well-being of millions of people. This has driven the largest technology companies to take tools that previously were limited to academic-scale problems and matured them to apply to their own production-grade problems over the last decade. Furthermore, they often have made these tools publicly available. Given this increased maturity and availability of robust methods and tools, now is an ideal time to revisit this area to transfer these advances to scientific computing applications.

Bibliography

[1] K. Y. Rozier, "Specification: The Biggest Bottleneck in Formal Methods and Autonomy," in *VSTTE 2016: Verified Software. Theories, Tools, and Experiments*, Toronto, 2016.

[2] D. Beyer, M.-C. Jakobs, T. Lemberger and H. Wehrheim, "Combining Verifiers in Conditional Model Checking via Reducers," in *Proceedings of the Conference on Software Engineering and Software Management*, Stuttgart, 2019.

The Challenges in the Development of Math Libraries

Miroslav Stoyanov, Eirik Endeve

November 22, 2021

Advanced math algorithms are at the core of scientific applications from linear algebra and solvers for partial differential equations, to data analysis using classical statistical methods or novel machine learning approaches. The complexity of scientific problems necessitates high performance computing and large scale scientific software efforts. The development cycle of a math library extends beyond the code, starting with an application informed challenge, goes through the development of a new algorithm, which then has to be implemented at scale to take advantage of the ever more complex supercomputers and only then, finally, make an impact in one or more domain applications. The software development is easily the weakest link of the process and one of the major reasons why applications are often times lagging behind the latest development of mathematics which in turn hinders scientific progress.

The development of a good mathematical library is a cross-discipline effort that requires close collaboration between mathematicians that understand the underlying algorithms, computer scientists that understand the relevant aspects of the computing environment such as accelerators and interconnect topology, as well as the domain scientists that will be the final users. Collaborative efforts towards software development are often disincentivized in the corresponding communities due to an overall lack of recognition, obscurity of the long term benefits of sustainable development, or general reluctance to cross the substantial language barrier between the disciplines.

Recognition for one's contribution is a great way to incentivize researchers to charge into new areas and to put in the extra effort of breaching communication gaps. However, software efforts seldom receive an appropriate level of recognition, especially when it comes to citations. Research software products come with either a DoI or a companion citable document (e.g., manual or concept paper) and while there is agreement that research software should be cited, the agreement is often times ignored in practice. The lack of citations leads to non-reproducible results and publications of dubious quality, and in addition, lowers the perceived value of the research software which has negative implication on both morale and career development of the researchers. Software development is often seen as a chore to be delegated or rushed in favor of work with better recognition.

Why is it that many papers go through the peer-review process with missing software citations, which would never happen if the omitted citations related to papers? The underlying cause of the phenomena could be a combination of lack of awareness that software comes with DoI, and lack of conscious recognition as valuable research. Perhaps, it is a matter of the simplicity of using a software tool as compared to using a result from a paper; reading and understanding a paper requires significant mental effort, using well written software is a mechanical regardless of the underlying code complexity.

Sustainability is a major focus of software development since the long term maintenance effort is much greater than what it takes to write the code , especially when maintenance implies support for an ever changing hardware landscape. Strict adherence to best practices and following good community policies are an investment into the longevity of the product; failure to make the investment leads to a well know "debt" phenomena that has to be repayed with interest that comes in terms of additional work-hours to fix, refactor, or even rewrite from scratch code that has become incomprehensible. Recently, the Exascale Computing Project (ECP) has changed the much in the way that software is perceived within the DoE complex; ECP has provided the needed financial support for sustainable development and, perhaps more importantly, the relatively long duration of the project has made it easier to see the far term benefits of the sustainability investment. Unfortunately, in recent years, the size and duration of ASCR math projects has decreased which has hindered the cycle of development of math libraries.

The core of building sustainability depends on the understanding of the long term benefits to writing good software. Short term projects diminish the perception of the benefits due to the uncertainty of the payoff, i.e., even thought good software can be deployed across multiple projects, when a project is short, the benefits will come in a follow-on effort that is not on the immediate horizon. This is further magnified by the tendency of researches to put themselves in narrow categories, e.g., "math", "software" or "application". The rigid separation hinters the progress of incorporating new ideas into the math libraries, since the extra efforts on one end are often seen as benefiting a different group of people. *How do we break the tendency of researchers to fit in narrow categories and move to a more fluid mentality where people shift their focus based on needs and opportunities, so that the benefits of sustainable software development are perceived as universally valuable?*

Communication between researchers of different training and background is paramount for success in the development of math libraries. While domain scientist and computer scientist take multiple math courses in their graduate studies, they are often reluctant to express their challenges in a sufficiently rigorous way; and mathematicians seldom (if ever) receive interdisciplinary training. The communication barrier is high and breaking it requires yet another sacrifice of short term productivity in favor of a long term payoff. Improving the efficiency of interdisciplinary communication can significantly improve the process of development of math libraries.

Complex terminology is necessary to express convoluted concepts in a sufficiently rigorous and deep manner, which is universally true for all areas of math, engineering and science. However, this leads to the development of independent languages where common words can have very different meaning or the meaning can be lost without the broader context. Breaking into a new field requires a significant cognitive effort to understand the language; however, collaborators across different fields seldom need to have the deep understanding of the concepts that necessitated the convoluted terminology in the first place. The communication gap can be breached much more efficiently if it is reciprocated by an effort to reduce the complexity of the domain language and to express the ideas with simpler words. Yet, efforts to simplify the ideas also require work and could be seen as oversimplifying or even diminishing the corresponding field. *How do we reduce the communication barrier by expressing the surface meaning of deep concepts with simpler and more accessible language, without oversimplifying the ideas and without creating the perception of diminishing the work in the field?*

The outlined challenges are not necessarily restricted to the math libraries and answers to the above questions would have a very broader impact and benefit to the larger scientific community.

Software Practices for Cutting-Edge Hardware Evaluation

Joshua Suetterlein, Joseph Manzano, Bruce Palmer, Andres Marquez, Kevin J. Barker
Pacific Northwest National Laboratory, Richland, WA, USA
{joshua.suetterlein, joseph.manzano, bruce.palmer, andres.marquez, kevin.barker}@pnnl.gov

As we reach the physical limits of hardware scaling, the economics of fabrication have facilitated a renaissance of new architectural designs. Domain specific (e.g., [1]) and re-configurable architectures promise the next advances in performance. Contrasting the rapid evolution of hardware is the legacy of large, complex, scientific software that can have increasingly long lifetimes. While software engineering's (SE) best practices have evolved within industry, scientific SE has remained comparatively stagnant. As scientific communities seek new experimental platforms, the lack of sufficient SE practices compounds the rising costs of hardware evaluation. The following draws on our experience from the DOE's CENter for Advanced Technology Evaluation (CENATE) [2] to describe observations and further contextualize challenges comparing our experiences with legacy NWChem [3] and emerging Machine Learning (ML) software for novel architectures.

Background: Since the DOD's Capability Maturity Model proposed in 1986, industrial software development has gone through several eras of growth, pioneering various metrics and processes [4], [5]. At the heart of this growth has been the monetary incentives of reducing Time To Market (TTM) while accruing minimal technical debt [6]. To this end, today's trends in industry promote quality software defined by its maintainability, portability, scalability, and resilience. Knowledgeable departures from current wisdom are typically only precipitated by critical limits in time and/or capital.

Since scientific discovery has become computer driven, SE in science is increasingly more important. More recently, SE has gained greater recognition within science due to the rising costs of maintaining legacy code, enabling reproducible science, and the emergence of next-gen hardware. Given the difference in funding models and incentives, the demand for greater process in scientific software development has been driven by funding agencies rather than scientists.

Of the differences between industry and the scientific community, the approach to evaluate and adopt cutting-edge technologies stands out. In both, employing a new hardware platform represent a high risk while the reward scales with the criticality of the mission/business. For many, exploring new platforms in the corporate setting is limited in time and scope to reduce cost. In the scientific community however, the draw of better resolution, faster simulations, and reduced power consumption facilitates exploration. This willingness is furthered when computational limits directly impede scientific discovery. Highlighting the differences in history, goals, and attitudes toward SE and platform adoption can provide a useful footing to pinpoint shared successful SE practices.

Challenges and Current Approaches: Given the scientific communities demand for advanced hardware, it is common to begin evaluation of a platform during its development. While beginning the evaluation process early presents a greater risk, early adopters are important not only as paying customers, but can also present valuable feedback guiding product development. The process of evaluating immature technology can be costly, presenting a technical debt not easily recovered or amortized. Moreover, the maturity of the underlying platform can directly impact the applicability of SE tools used to minimize this cost. For example, a portability layer like Kokkos [7] requires a modern C++ compiler which might not exist until later in the platform's lifetime.

To this end we present an evaluator's view of applicable SE tools across a platform's development as a strawman. In Figure 1, the timeline presented is the maturity of the hardware/software stack. Each point along the line represents the availability of a feature from the perspective of the evaluator. The highlighted boxes present when a software tool, practice, or principle becomes applicable.

From platform inception, several SE tools are readily applicable to the software developed by the evaluator. While meaningful results can be gleaned as soon as the simulator is ready, more advanced tools are not available until later versions of the software ecosystem have been completed. Principles based on object-oriented programming (e.g., SOLID [8]) become possible when more advanced system tools exist; however, they present an upfront technical cost an evaluator may not be willing to pay. Rather early adopters may wait to invest in software development until the platform is more mature (i.e., production). This more cautious approach can result in the delayed usage of Agile, robust documentation, Test-Driven Development (TDD), and containers.

While the application of Agile to scientific software development deserves a more complete discussion, we

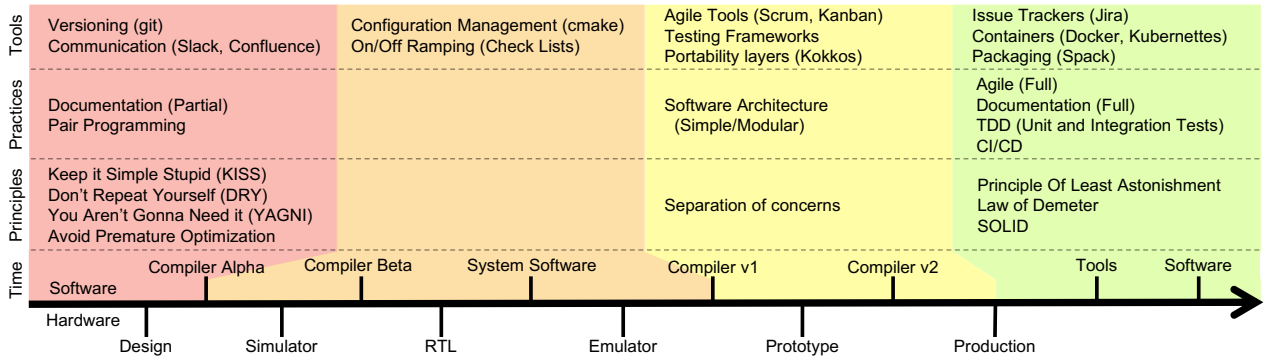


Fig. 1. Applicability of select SE tool, practices, and principles according to the maturity of platform. Each color represents an additional set of SE mechanism that become usable.

provide a defense of our current position on Agile's timeliness. From the perspective of Agile being conflated with its implementations like Scrum and Kanban, it is unclear if the overhead associated with this process is worthwhile as evaluation teams can be trivially small and their metrics (e.g., lead time) unapplicable. Conversely, the principles of Agile as presented in [9] can be more widely applied by hardware evaluators.

Opportunity: New SE approaches are required to help amortize the rising costs of platform evaluation at various stages of development. One effective way to develop new practices is to first define pertinent metrics [5]. Current software metrics based on code maintainability and portability have limited scope. For example, investing in a well-defined, reusable testing harness limits evaluation to platforms with a semi-mature compiler and OS. One potential metric of interest for hardware evaluators (regardless of platform maturity) is the rate at which features are evaluated. In this instance, features are comprised of both performance indicators (e.g., FLOPS) and software/hardware components. More work will be required to disambiguate the delay caused by communication between the vendor and customer to make this and similar metrics viable for process refinements.

Another opportunity to influence the hardware software/development cycle is through the availability of mini-apps and micro-benchmarks that model performance bottlenecks in existing applications. If small enough, this type of code artifact can be introduced early in the development cycle and can have a real chance of guiding hardware and software development. Currently, many mini-apps, although simpler than the full application, are still sufficiently complicated that they require relatively mature compilers and libraries to build. Creating a formal structure for identifying useful micro-benchmarks for use early in the development cycle is a pressing need.

Timeliness: At PNNL, we draw on our experience across two thrusts to demonstrate the timeliness of new SE practices. The first has been our experience maintaining NWChem chemistry package. During its lifetime, this package has been ported across various architectures and its kernels have been used to evaluate new platforms. Over the years, the package's interfaces have become simpler and more well defined. While its software architecture follows a modern modular design, evaluating hardware with the package is limited to more mature platforms. Conversely, under the CENATE project we have evaluated multiple ML platforms across various domains. Even though many of these platforms boast support for the latest ML frameworks, the effort to produce fair comparisons has been challenging. This process is made difficult by complex simulation environments, immature compilers, and limited runtime support.

As we enter a new era of domain driven computer architecture, now is the time to address how we evaluate new platforms in a cost-efficient manner. Understanding successful software principles and developing new practices that reduce the time and cost of hardware evaluation will help streamline adoption and facilitate greater scientific discovery.

References

[1] M. Emani *et al.*, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science Engineering*, vol. 23, no. 2, pp. 114–119, 2021.

[2] N. R. Tallent *et al.*, "Assessing advanced technology in cenate," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2016, pp. 1–2.

[3] E. AprÃ *et al.*, "Nwchem: Past, present, and future," *The Journal of Chemical Physics*, vol. 152, no. 18, p. 184102, 2020.

[4] G. Booch, "The history of software engineering," *IEEE Software*, vol. 35, no. 5, pp. 108–114, 2018.

[5] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *10TH International Software Metrics Symposium, METRICS 2004*, 2004.

[6] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[7] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, 2014.

[8] H. Singh and S. I. Hassan, "Effect of solid design principles on quality of software: An empirical assessment," *International Journal of Scientific & Engineering Research*, vol. 6, no. 4, 2015.

[9] "Principles behind the Agile Manifesto," 2001. [Online]. Available: https://agilemanifesto.org/principles.html

Studying Data Patterns to Enable Scientific Exploration with Data-Intensive Distributed Workflows

Nathan Tallent (tallent@pnnl.gov), Pacific Northwest National Lab
Zhe Feng, Pacific Northwest National Lab

November 19, 2021

Challenge

Scientific exploration is increasingly distributed and data-intensive. The answers to many questions involve composing applications with different characteristics, e.g., numerical solvers, data analytics, and machine learning. To focus on exploration rather than application development, domain scientists emphasize productivity and flexibility. The result is *workflows*, or loose compositions of different applications or tasks. Each application potentially uses a different programming models. Data sources are distributed. As a result, distributed I/O is often the de facto method for task communication or composition. Although such composition is easy to program, it creates fundamental performance limitations [3, 4] that restrict scientific exploration.

An example, consider FLEXTRKR [1, 2], a Lagrangian storm tracking workflow that applies to remote sensor data (e.g., satellite and ground-based radar observations) or massively parallel (Eulerian) CFD numerical simulations. Storm dynamics can change *faster* than the CFD's typical data output resolution — i.e., large eddy dynamics within ocean region per time unit — which is limited primarily because of I/O impacts on model integration due to limited storage rates and space. As a result, some important information on storm dynamics are lost.

Opportunity

There is an opportunity for comprehensive studies of the data patterns within distributed workflows. For example, understanding a workflow's composition patterns and having flexible methods for expressing composition options would enable the component composition so as to avoid major system bottlenecks. This would enable scientific exploration by improving the software productivity of workflow development. It could also avoid common performance bottlenecks that impose fundamental limitations on the science that can be accomplished.

Collaborations of domain and computer scientists identify data patterns within distributed workflows. Domain scientists provide a vision for productive domain programming. They would

- Establish context by defining the domain challenges that represent fundamental limitations imposed by current computing solutions;
- Identify algorithmic patterns worthy of special study, i.e., those that abstract and represent common sub-problems; and
- Comment on the state-of-the art in software practice as well as benefits and drawbacks for different software methodologies, including when novel software techniques and technology are and are not helpful.

Computer scientists would use performance modeling and simulation for software-hardware co-design. They would

- Identify plausible programming abstractions for avoiding data access bottlenecks, ranging from best practices, state-of-the-art, to visionary, that generalize domain needs with respect to domain programming patterns;
- Characterize each solution's tradeoffs between performance and productivity, as well as its dependence on static and dynamic analysis, runtime paradigms, and prevalent system architectures; and
- Inform machine procurement, including system architectures and runtimes, improving long-term resource utilization.

Data-informed workflow patterns map well to emerging system architectures. Data movement corresponds to many fundamental performance limitations. First, There is limited potential for improvement of latencies in high-end networks because network links already communicate at near fundamental limits. Second, although bandwidth improvements are expected, they are already difficult to exploit. The bandwidth of data paths is generally inversely

proportional to access distance because the density of network links becomes sparser with longer distances. Further, fully exploiting the highest-bandwidth paths is challenging. Bandwidth is realized only with bulk transfers over many data 'channels', i.e., network links, storage devices, or memory channels/banks. Thus, naturally irregular or sparse accesses must be converted into regular bulk accesses. Third, the fastest data stores are severely capacity constrained, whether in memory or storage. Since data volumes can be arbitrarily large, these fast data stores must be very carefully managed.

Data-informed workflow patterns correspond to bottlencks in emerging systems. Emerging system architectures blur memory and storage with large byte-addressable persistent pools. The challenge is the tiered nature, where the bandwidth within each tier is much higher than across its boundaries. At the same time, there is an increasing interest in smart devices, including NICs, storage, and memory. This effectively creates the potential for conducting near-data high-bandwidth computing within each tier of the data storage hierarchy. For example, operations within each tier could exploit that tier's native bandwidth, rather than the limited bandwidth of the tier's data interface.

Dynamic and static analysis provides valuable assistance. Static analysis program analysis is a common and important technique for understanding data flow. Given the importance of irregular or sparse patterns, we argue that combining dynamic and static analysis will be critical for accurate assessment of data flow. Even for "regular" problems, there are many choices for the "size" and "shape" of work that have a significant effect on accelerator performance, e.g., which loops to include, instruction mix, data partitioning, access pattern, and locality. The challenge increases for sparse and irregular problems, where it is difficult to identify good data layouts and access patterns to achieve good memory performance

Timeliness

Studying these questions is timely for several reasons. First, scientific exploration is increasingly distributed and data-intensive because it combines sensor data, numerical solvers, data analytics, and machine learning. This combination requires coordination between experimental facilities (with specialized instruments), computing facilities with supercomputers, and possibly even cloud resources. Second, there is an opportunity because of the nature of emerging systems, which blur memory and storage and incorporate smart devices. Finally, for the potential benefit of success provides benefits both for software productivity and for enabling data-intensive scientific exploration.

References

[1] Z. Feng, R. A. Houze, L. R. Leung, F. Song, J. C. Hardin, J. Wang, W. I. Gustafson, and C. R. Homeyer. Spatiotemporal characteristics and large-scale environments of mesoscale convective systems east of the rocky mountains. *Journal of Climate*, 32(21):7303 – 7328, 2019.

[2] Z. Feng, L. R. Leung, R. A. Houze Jr., S. Hagos, J. Hardin, Q. Yang, B. Han, and J. Fan. Structure and evolution of mesoscale convective systems: Sensitivity to cloud microphysics in convection-permitting simulations over the united states. *Journal of Advances in Modeling Earth Systems*, 10(7):1470–1494, 2018.

[3] R. D. Friese, B. O. Mutlu, N. R. Tallent, J. Suetterlein, and J. Strube. Effectively using remote I/O for work composition in distributed workflows. In *Proc. of the 2020 IEEE Intl. Conf. on Big Data*. IEEE Computer Society, December 2020.

[4] J. Suetterlein, R. D. Friese, N. R. Tallent, and M. Schram. TAZeR: Hiding the cost of remote I/O in distributed scientific workflows. In *Proc. of the 2019 IEEE Intl. Conf. on Big Data*, pages 383–394. IEEE Computer Society, Dec. 2019.

Literature Validation and Experimental Analysis Using Advanced Machine Learning Tools

Jeff Terry, Professor
Departments of Physics; Mechanical, Materials, and Aerospace Engineering; & Social Sciences
Illinois Institute of Technology
terryj@iit.edu

Challenge: Many of our great Scientific Grand Challenges are made more difficult to address by a growing body of unreliable and irreproducible research results that are published every day(1). According to recent studies (2; 3), more than half of x-ray photoelectron spectroscopy (XPS) results in the literature are incomplete, and at least 30% are simply incorrect. New results build on old. Thus, when previous results cannot be trusted, the entire scientific edifice is in danger of crumbling. Enormous amounts of money, time, and resources are wasted on doomed attempts to do research and develop processes based on incorrect assumptions.

Opportunity: ***This growing crisis can and should be addressed head on, by developing artificial intelligence (AI) tools that can automatically validate the scientific literature and then to allow experimenters to automatically draw from this knowledge database to analyze new experimental measurements. This would revolutionize the methodology of characterization science by developing novel AI techniques, resources, and tools for assessing and significantly improving the reliability of experimental results in the field and thereby to solve the acute reproducibility problems that slow research progress and inhibit effective technology transfer and manufacturing innovation.***

Timeliness: The development of artificial analysis tools has reached a level of maturity where it should be possible to extract and formally represent the contents of experimental results and analyses, and automatically assess the overall reliability of the published literature with the next 5 years. It should be possible to design these artificial analysis tools to be able to assist experimenters in the design of experiments, where having valid information can be used to guide experimenters to the proper choice of techniques and parameters for characterization of materials and reactions. This would be a major revolution in the field and would allow for gains in interdisciplinary research. It is time to develop multimodal analysis of the scientific literature and experimental data, integrate these with the existing scientific processes of characterization, and apply these new abilities to applications in chemistry, physics, materials science, and energy production. The AI research must involve novel application and integration of diverse techniques in natural language understanding, machine and deep learning, information extraction, and knowledge representation. By targeting both fundamental component tasks as well as task integration in this complex real-world domain, significant advances in AI fundamentals and systems development, in addition to the utility to the subject domain can be engineered. This effort will also improve scientific productivity by reducing time lost in complex analyses and replication of low reliability studies.

Two pathways must be developed, a literature understanding pipeline (Figure 1) and an experimental processing pipeline ((Figure 2). The Literature Understanding pipeline will involve extracting experimental parameters and results from the peer-reviewed chemical literature and assessing the validity and self-consistency of the information in each individual paper. Once a paper has been identified as being reasonably well conducted, the information contained within it will be collected for inclusion into the materials database (MDB). The Materials Database can then be used for experimental planning and analysis. There are challenging technical issues to address through AI automation. In order for the AI tools to analyze the scientific literature, they will need to extract and link information reliably from multiple places within an individual paper. There is no standardized structure for characterization papers. Information must be extracted from text, tables, and figures, cross-tabulating and linking relevant items together. These data must then

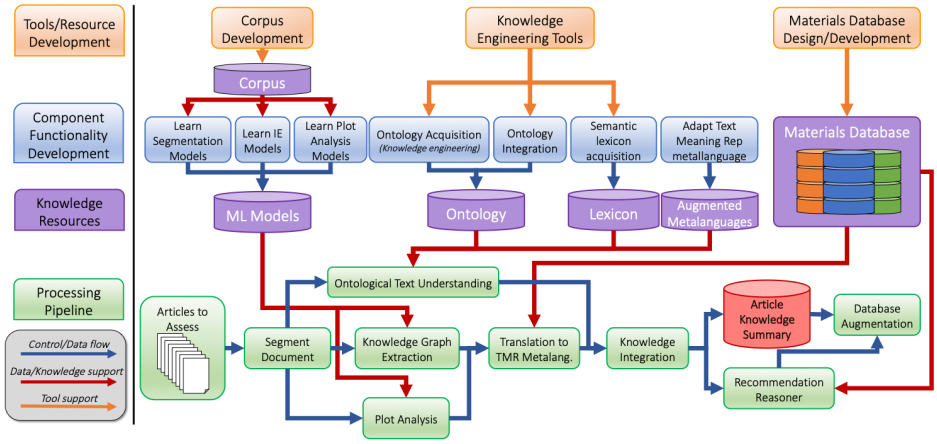


Figure 1: **Literature Understanding** - Overview of the proposed approach for content and reliability analysis of the scientific literature.

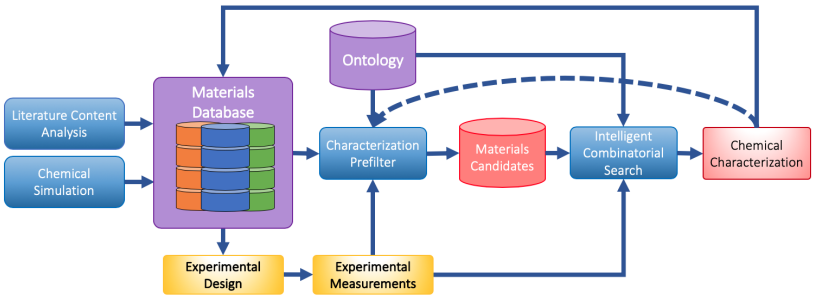


Figure 2: **Experimental Processing** - The proposed approach for using the materials database for analysis.

be checked for self-consistency and completeness, which will require the system to understand the types and parameters of the characterization tests described. It will not be a easy but now is the time to attempt to accomplish this goal.

References

[1] National Academies of Sciences, Engineering, and Medicine, *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press, 2019. [Online]. Available: https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science

[2] G. H. Major, T. G. Avval, B. Moeini, G. Pinto, D. Shah, V. Jain, V. Carver, W. Skinner, T. R. Gengenbach, C. D. Easton, A. Herrera-Gomez, T. S. Nunney, D. R. Baer, and M. R. Linford, "Assessment of the frequency and nature of erroneous x-ray photoelectron spectroscopy analyses in the scientific literature," *Journal of Vacuum Science & Technology A*, vol. 38, no. 6, p. 061204, 2020. [Online]. Available: https://doi.org/10.1116/6.0000685

[3] M. Linford, V. Smentkowski *et al.*, "Proliferation of faulty materials data analysis in the literature," *Microscopy and Microanalysis*, pp. 1–2, Jan. 2020. [Online]. Available: https://doi.org/10.1017/s1431927619015332

Software Engineering Mechanisms To Facilitate Best Practice Design Principles

Arnold Tharrington (arnoldt@ornl.gov), Mark Berrill (berrillma@ornl.gov), Reuben D. Budiardja(budiardjard@ornl.gov), Dmytro Bykov (bykovd@ornl.gov), Antigoni Georgiadou (georgiadoua@ornl.gov), Austin Harris (harrisja@ornl.gov), Justin Lietz (lietzjg@ornl.gov)

Oak Ridge National Laboratory, P.O. Box 2008. Oak Ridge, TN 37831-6008

Abstract

Scientific computer simulations and modeling are growing unabated in their importance of elucidating physical processes. In addition, as computer architectures become more powerful, scientific software correspondingly becomes more complicated to achieve better modeling fidelity. This ever-increasing software complexity often results in unwieldy code that is difficult to maintain and extend. A primary mitigation effort is for scientific programmers to practice Best Practice software development. This paper explores the challenge of creating mechanisms for enforcing Best Practices in software engineering.

Motivation and Challenges

Scientific codes may not follow Best Practice coding---commonly accepted good design principles [2]---for various reasons ranging from junior programmers who are unaware of Best Practices, to research and publishing time pressures that preclude practicing good software development. Whatever the reason, this results in poor foundational, nascent code which leads to higher future barriers of properly rewriting the code. A robust Best Practice tool would greatly mitigate creating poor nascent code. A Stripe and Harris Poll found the average developer spends 42% of their time dealing with technical debt and maintenance issues due to bad code, and the opportunity cost of bad code comes to $85 billion annually [1].

There are two primary aspects to writing scientific software: the software management aspects such as version control, issue/bug tracking, etc; and the actual writing of the code following Best Practices. Programmer competency in both aspects is critical to increasing the productivity and sustainability of code.

There are numerous software management tools focused on competency with aspects of project management. A short partial list is Jira, Git, Github, Slack, and Trello. These tools generally have reasonably low usability barriers that permits novice programmers to quickly gain a working knowledge level - i.e. it does not take several months to gain a usable working knowledge level. For developers, there are a number of tools available for debugging (gdb, DDT, TotalView, etc.), for debugging memory problems (Valgrind, etc.), static analysis/linters (cppcheck, etc.), profiling (perf tools, vampire, callgrind, etc), even code formatting (clang-format).

In contrast, there are very few tools to enforce the software engineering social norms of writing code. Existing tools such as compiler warnings/messages, static analysis/linters can identify bugs and even make certain recommendations (e.g., use standard library functions) but are

generally very limited in scope, and they are not always designed with scientific software in mind.

The central challenge presented here is the lack of mechanisms that help application developers follow the set of guidelines or Best Practices for scientific software.

Opportunity

The challenge presented above may be addressed by providing a scientific measure to codify Best Practices into metrics. As an example, consider the *Single-Responsibility Principle* (SRP) [3] which states that every function, class, module, etc. shall have responsibility for one aspect of the program. How does one measure the responsibilities of functions, modules, and classes for creating a SRP metric? As another example, consider a legacy, monolithic code that needs refactoring to be more maintainable. Upon analyzing such code, the tool would need to measure the degree of data dependencies and scopes of functions, modules and classes, and provide a refactorization recommendation. The opportunity here is to create a metric that can capture these dependencies such that the tool can use to decide whether these dependencies are too strong and recommend refactoring.

A single metric will never cover the needs of all scientific software due to variations in code lifetime, domain subcultures, algorithms, and requirements. Therefore any such automated Best Practice recommendations needs to be tunable to the situation at hand, and ideally, multiple solutions would be generated and selected from by the user.

Another opportunity is to incorporate the use of Artificial Intelligence (AI) to automatically encode a set of Best Practices into usable metrics that may then be applied to code development. The idea is similar to using AI for learning languages in that one can view Best Practices coding as learning a new language. This solution will require the collaboration of domain and computer scientists to create code data sets and train modules.

Potential Impact and Timeliness

A robust set of mechanisms that would aid software developers in Best Practice coding would have a significant impact on code sustainability. Emerging AI/ML technologies, combined with the rich source of large, diverse scientific code bases available as training data, make this an opportune time to tackle this challenge. Tools like GitHub CoPilot have already demonstrated the capability to perform automatic code generation by harnessing AI with the massive amount of open-source code. Similar techniques could be applied to perform static code analysis and implement Best Practices.

References

[1] Stripe (2018). *The developer coefficient*. Retrieved November 22nd, 2021 from https://stripe.com/files/reports/the-developer-coefficient.pdf
[2] Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall.
[3] Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

Challenges and Opportunities for Scientific-Software Development Research Incorporating Cognitive Engineering at Next Generation Light Sources

Jana Thayer (SLAC, presenting author), jana@slac.stanford.edu, 650-926-4656
Jeff Shrager (Stanford Symbolic Systems Program), Chuck Yoon (SLAC)

Challenge: Next-generation light sources, such as the Linac Coherent Light Source (LCLS-II), will collect data sets of unprecedented scale at high velocity (TB/second, EB/year), opening up new areas of science, but at the same time making it far more difficult to extract desired physical information from these data. Scientists increasingly rely on sophisticated scientific workflows and HPC to derive answers to scientific questions from a variety of datasets, however, the rate at which a broad user community acquires knowledge on how to use emerging technologies lags the rate of evolution of these technologies by a significant margin – impacting timely exploitation of DOE's investments. Deriving information from scientific data is technically challenging for many reasons, but primary barriers to "doing the science" are developing the software to manipulate and analyze the data, and finding and indexing the relevant data in the first place.

What makes scientific software development challenging for the light source community is the divide between science domain knowledge and software development expertise. Domain scientists use software development (on heterogeneous platforms) as a means to an end, to answer a specific scientific question, and have traditionally done so at scales that are tractable for small research groups to develop and exploit. As a result, these scientists develop software that works for their application, but do not generally have the time, resources, expertise, or motivation to make that software more generically applicable or available to other scientists to solve similar problems. Similarly, these traditional solutions are not scalable to the onslaught of data that will emerge from the upcoming upgrades to the light source facilities. Conversely, software developers who are formally trained to produce robust, reliable, scalable, and maintainable software that could be used by a larger community generally lack the science domain expertise necessary to answer key scientific questions or extrapolate to the next generation of experimental paradigm. Doing science is about asking questions, making observations, exploring data and ideas, identifying the unexpected, and iterating as new ideas are discovered which lead to new questions at deeper levels. Well-informed software development is necessary at each step of this cycle and must adapt with each iteration through the cycle because the questions evolve as more data comes in from analysis and feedback from the community – often with very short cycle times in the case of light source science. As a result, to solve the gap between science domain knowledge and software development expertise, one needs to find a way to significantly simplify the software development process by capturing information about the most common ways data are manipulated in a particular scientific context and making them as easy to implement as possible. The funding mechanisms that exist to develop new science techniques are not designed for long term investment and maintenance of software packages used by a community of users. Priority for porting software to sustainable libraries[2] that can be leveraged by a larger community would be a welcome intervention to improve the impact of software developments and encourage its reuse.

The second impediment to scientific software development is finding and ingesting the relevant data. The light source facilities generate large datasets, but it remains difficult to share

them across facilities and analyze data in the same context, partly because the data are dispersed and vary in format, but also because it is difficult to identify and retrieve similar data without already knowing that it exists and is suitable for use. Finding suitable data is hampered by the lack of a widely adopted standard for tagging data and tracking data provenance in the multidisciplinary, fast-changing world of light source science. To make best use of data, it is necessary to understand the conditions under which the data were acquired. This metadata is often recorded in notebooks or in spreadsheets and does not use standard identifiers, nor is it typically stored with the data itself. It is increasingly difficult to reproduce scientific results due to the challenges of sharing large datasets and the practicality of implementing a complex software stack on different architectures. Results that rely on Machine Learning are especially hard to replicate if the neural network weights are not shared.

Opportunities: One opportunity for research is to incorporate AI coding aids, such as intelligent auto-complete. While developing code today, an individual can use tab completion in iPython[3] to immediately access the method, expected arguments, and a description of what the method does. It is easy to imagine context-specific autocomplete that incorporates a large library of light source methods such as scikit-beam[2], and a small additional leap to an AI could help the scientist easily discover methods, algorithms, and libraries that might be useful in the current context. Take the idea further and incorporate audio commands to help scientists manipulate data during analysis. A scientist interacting with data could verbally command an AI to create a 2D plot, strip out bad pixels, and show only hits with intensity greater than some threshold. A request that exceeds computational capacity could be automatically launched on an HPC resource. Another research opportunity is using AI to automatically generate keywords and tags for acquired data that enable the data to be found again more easily. Analogously, creating intelligent tags for codes would help make commonly used methods or algorithms more easily findable by other scientists interested in solving a similar problem.

Modern Cognitive Engineering methods can also help user facilities, such as LCLS-II, optimize operations in the high performance, AI and automation-rich, human-in-the-loop environments that are becoming increasingly demanding as data rates and experiment complexity increase. We believe that classical design methods such as user studies and intuitive design, must be augmented by instrumented control systems, "flight simulators", and cognitive+physics simulation engines that can be used to carefully analyze the system as a whole, and computationally compare user interface and workflow designs to optimize data gathering and analysis efficiency in these increasingly demanding settings.

Maturity and Timeliness: A paradigm shift in how scientists use software and computing to address research questions is urgent given the timeline for new facilities such as LCLS-II, APS-U, ALS-U, etc. The prevalence of Artificial Intelligence and Machine Learning techniques[3], and recent emphasis on co-design teams of domain scientists, detector developers, data scientists, and software developers to solve larger problems, puts the DOE research community in a powerful position.

[1] https://ipython.org/ipython-doc/3/interactive/tutorial.html

[2] https://scikit-beam.github.io/scikit-beam/introduction.html

[3] https://doi-org.stanford.idm.oclc.org/10.1007/s42979-021-00592-x

Fundamental Needs for Establishing Evidence-Based Scientific Software Engineering Practices

Charles Edison Tripp (charles.tripp@nrel.gov), Graham Johnson (graham.johnson@nrel.gov),
Nalinrat Guba (nalinrat.guba@nrel.gov), Sagi Zisman (sagi.zisman@nrel.gov),
Jordan Perr-Sauer (jordan.perr-sauer@nrel.gov)
Data, Analysis and Visualization Group, Computational Science Center,
National Renewable Energy Laboratory

Motivation

The entirety of the scientific venture rests upon the empirical foundations of data collection, analysis, and modeling. As scientific knowledge has grown, so have the information processing resources required to sustain our progress. Computing systems executing scientific program codes are our most effective and efficient tools for achieving scientific discovery. However, we have hypocritically neglected to apply rigorous empirical study to our scientific software engineering practices, even as our utilization and reliance on software has grown from novelty to prerequisite in modern scientific inquiry. As scientists, acolytes of the scientific method, ***we know*** *that practicing evidence-based research software engineering (RSE) will multiply the efficacy and throughput of human scientific discovery* by improving the manner in which we collectively employ computing resources in humankind's grand empirical journey. To study software practices, we must first have scientifically valid observational tools, developed through systematic metrological research. Next, observational studies can apply these metrics to identify possible causal links, and controlled trials can be used to confirm these links and inform evidence-based RSE practices. While data management has been recognized as a core component of scientific inquiry, the resources and plan for the software development required to generate, record, and analyze that same data has been neglected. We have seen projects with significant software engineering requirements forced to mask software development efforts, including critical verification tasks, as implied components of accepted "scientific" project tasks.

The Challenge: Enabling Scientific Study Of Scientific Software Engineering

Scientific study requires the fundamental tools of empiricism: the ability to intervene, to observe an effect, to analyze our observations, and to do so in a replicable, verifiable manner. While there are ongoing efforts to quantify software properties, fundamental scientifically validated measurement metrics and methods have not been rigorously developed and therefore credible scientific study of RSE is not yet possible. *The principal* ***challenge*** *we face in moving towards evidence-based RSE is the rigorous development and verification of robust, repeatable, and reliable methods for observing RSE interventions, outcomes, and confounding factors.* How do practices such as code review, unit testing, pair programming, language, code style, team composition, and domain knowledge impact development time, optimal team size, error rates, and code quality in research settings? As lamented in the classic "The Mythical Man-Month" [2], we do not even have a firm grasp on the relationship between team size and development time. Without a way of making scientifically valid observations of RSE practices and outcomes, we have no means to produce scientifically valid answers to these questions. These difficulties are multiplied by the myriad of suspected confounding factors: the uniqueness of each project's goals, team composition and contributor backgrounds, computing environments, inputs and outputs, timelines, funding mechanisms, the transdisciplinary nature of scientific computing and the commensurate interaction between domain and software engineering expertise amongst team members.

Opportunity: A Path Forward

We ***propose*** *the development and validation of robust, repeatable methods of measuring RSE outcomes and potential causal factors.* As metrological research furnishes appropriate observational tools, two concurrent synergistic thrusts can begin: **observational studies of RSE projects** which identify potential correlates and causal factors, and **randomized controlled trials** (RCTs) which confirm or reject these candidate hypotheses. The **impact** of developing scientific RSE

metrics will be to enable scientifically valid studies of RSE practices, allowing us to proceed with large-scale scientific study and, for the first time, practice evidence-based scientific software engineering. While software metrics have been developed, by and large these metrics have not yet been placed on solid scientific footing. The initial thrust extends foundational metrology research in software performance, measurement, metrics, and assurance, such as those developed by NIST [4, 5]. For example, assessments of domain experts might be used as a the ground truth for a set of measurands, and a metric might be validated by ensuring that ranking is preserved by a measurement method as a homomorphism between the expert domain and the measured quantity [1]. Moreover, a framework for constructing complex measures has been suggested whereby base measures are combined into derived measures and subsequently into attributes. What we know as qualities of software such as extensibility and complexity can then be conjectured from the attributes [6].

Next, mature observational tools of observational study including case-control and longitudinal studies can be applied to RSE projects in much the same fashion as in epidemiological and psychological research. As relationships are identified, causal hypothesis will emerge; these can be evaluated through randomized controlled trials that will yield strong evidence which can inform evidence-based practices. While RCTs may appear infeasible, we propose "starting small"; for example, the effect of a particular development practice (e.g. commenting, pair-programming, unit testing) could be studied through the use of standardized programming tasks and the collection of validated measurements of the software produced. If observational and RCT studies are applied systematically, confounding factors (e.g. programmer experience) can be identified and controlled for in subsequent studies. Such practices are commonplace in evidence-based medicine and epidemiological research, and these practices can be adapted to RSE, as proposed by Chapetta and Travassos [3], and earlier by Kitchenham, Dybå, and Jørgensen [7].

Timeliness: From Darkness Springs The Dawn Of Evidence-Based RSE

The seeds for rigorously developing the observational tools of scientific study of RSE have already been planted; now we must tend to our garden and nourish these early sproutlings if we are ever to enjoy the fruits that they may bear. Pioneering researchers have begun to establish outlines and frameworks for designing and verifying observational tools for RSE studies [5]. Through the application of systematic scientific metrology to research software development, we can establish and validate rigorous measures, methodologies, benchmarks, and guidelines thereby enabling deep observations and comprehension of the interactions between RSE inputs and outcomes. In turn, these rigorous foundations will support the continued evolution and advancement of evidence-based RSE products and practices for decision makers across the DOE and wider scientific community.

References

[1] Alain Abran. *Software metrics and software metrology*. John Wiley & Sons, 2010.

[2] Frederick P Brooks. "The mythical man-month". In: *Datamation* 20.12 (1974), pp. 44–52.

[3] Wladmir Araujo Chapetta and Guilherme Horta Travassos. "Towards an evidence-based theoretical framework on factors influencing the software development productivity". en. In: *Empirical Software Engineering* 25.5 (Sept. 2020). DOI: 10.1007/s10664-020-09844-5.

[4] David Flater. "A System of Quantities from Software Metrology". In: *Measurement* 168 (Jan. 2021). DOI: 10.1016/j.measurement.2020.108435.

[5] David Flater. *Quantities and Units for Software Product Measurements*. Tech. rep. Mar. 2020. DOI: 10.6028/NIST.IR.8289.

[6] David Flater et al. *A Rational Foundation for Software Metrology*. en. 2016-01-20 2016. DOI: https://doi.org/10.6028/NIST.IR.8101.

[7] B.A. Kitchenham, T. Dyba, and M. Jorgensen. "Evidence-based software engineering". en. In: *Proceedings. 26th International Conference on Software Engineering*. Edinburgh, UK: IEEE Comput. Soc, 2004. DOI: 10.1109/ICSE.2004.1317449.

Using big data to study and improve scientific software

Váleri N. Vásquez*[1] and Richard Barnes*[1]

[1]Energy and Resources Group, University of California, Berkeley

1 Challenge

Computation has emerged as a "third leg" of science. The traditional legs—theory and experimentation—require quality instruments and communication to function well. Similarly, computation requires high-quality software that is correct, publicly available, and documented. The process of designing, developing, and refining such software should—like the discoveries it is being used to elucidate—be treated empirically, and include iterative experimentation guided by data. A primary obstacle in building the evidence base that is required to apply the scientific method to scientific software is the *observability challenge*: representative data is required, but has been historically difficult to obtain. **Here, we propose to use public archives of open source software and scientific publications to help solve the observability challenge.**

Previous work on the *observability challenge* has used surveys to characterize how scientists develop and use software in their work; however, these efforts have been largely qualitative and restricted to small sample sizes, such as $N \leq 2,000$ in [1]. This may introduce bias while limiting generalization across scientific domains. Surveys, for instance, may be unable to quantify the usability, quality, or sustainability of software and, unless they are conducted continuously, cannot track the evolution of discipline-specific software applications nor trends in the best workflows and community practices for software development.

2 Opportunity

As detailed below, information concerning the development and use of scientific software is publicly accessible at an unprecedented scale and ever-increasing rate via platforms such as GitHub, arXiv and its discipline-specific sub-archives[1], and open access academic journals. Likewise, citation metadata has become available from services like Web of Science, a global citation database. The recent proliferation of these digital venues and others like them is thanks in part to the **growing appreciation of transparency and accessibility in scientific disciplines [2]: a cultural shift in expectation and practice that presents a prime opportunity** for addressing the observability challenge.

With 73M developer-users and 200M code repositories (of which 28M are public), GitHub is the largest source code host, though services such as Code Ocean are growing in popularity. GitHub offers both the current and historical code of a project along with a rich database of metadata including the frequency and size of proposed changes to the code (pull requests), the length of time pull requests and issues remain unaddressed, counts of unique and regular contributors, cross-contributions by particular developers, and the number of users following a project. This data has been used to investigate the effects of gender bias by linking user names to aspects of the software development cycle [3], the impact of issue tracking techniques on project success [4], and how team characteristics and the choice of programming language affects software quality [5].

Over the past 30 years the number of preprints on services like arXiv has increased 63-fold [6]. While this accounts for only 4% of all research articles [6], in some fields such as theoretical computer science and machine learning over 60% of published papers are on arXiv, with usage rising in other areas [7]. Of these preprints, 41–56% go on to be published in academic journals [6, 7], with some disciplines exceeding even that: in the case of bioRxiv, nearly 38,000 preprints were uploaded in its first five years, two-thirds of which were later published in peer-reviewed journals [8]. arXiv has made science more observable by making scientific results available sooner and without paywalls in machine-readable LaTeX format. As a result, anyone can observe and analyze scientific progress in real-time. For instance, analysis of arXiv full-text

*Contributed equally.

[1]e.g., bioRxiv, ChemRxiv, EarthArXiv, medRxiv, SocArXiv

corpuses has been used to study the co-evolution of scientific topics and collaboration networks [9], text reuse and its effects on citations [10], and to predict future directions in AI [11].

The Web of Science catalogues 182M records including journals, books, and conference proceedings, spanning literature from the year 1800 onward and patents dating back to 1963. This data can be used to understand how scientific works relate to each other and what their impact is. For instance, citation metadata from Web of Science has been used to show that progress ossifies in large fields of science due to citation network effects [12] and that the deaths of preeminent scientists lead to reorganizations of citation networks and the emergence of new leaders and ideas [13]. This data has also been used to conclude that 28% of the scientific literature is open access and that open access articles receive 18% more citations [14].

3 Timeliness

The foregoing demonstrates that open access to scientific texts, source code, and citation metadata allows for myriad analyses of each of these data sources on their own. **But only the fusion of these data sources will allow us to fully characterize the development and use of scientific software**, drawing a more direct line between innovation and the practices that drive it. Some preliminary work in this direction does exist. Combining citation and code data indicates that adding software to package management systems can increase a paper's citation count by 280% [15], while combining textual and code data indicates that high-profile projects tend to use statically-typed languages and tend to have fewer female contributors [16].

However, in our review of the literature we find that most "fusion" work to date focuses on fairly simple, descriptive conclusions. For instance, a representative paper reviews all code-paper pairs in the Microsoft Academic Graph only to conclude that repository stars follow a power distribution [17]. **Emerging efforts in social and cognitive science, as well as human-computer interaction (HCI) [18], can help bridge the gap between challenge and opportunity** in scientific software. By fusing the big data sources described here with social science methods, researchers can overcome the observability challenge to answer timely questions such as:

- Do software design choices influence subsequent scientific innovation or research question creativity?
- Can the development of "foundational" software be causative to subsequent innovation?
- Do co-author demographics predict software quality or longevity, or features of the software's community?
- What methods are most successful in supporting long-term software projects (e.g., intermediate publications, citations garnered, or cyber-infrastructure grants)?
- Do good software practices (such as those described in [19]) spread via collaboration networks or are they driven by outside factors?

The impact of answering questions such as those exemplified above is an improved understanding of how to design and maintain scientific software, reward and incentivize developers, build communities, and, ultimately, drive new scientific advances.

[1] Hannay et al. 2009. DOI: 10.1109/SECSE.2009.5069155. [2] Wilkinson et al. 2016. DOI: 10.1038/sdata.2016.18. [3] Imtiaz et al. 2019. DOI: 10.1109/ICSE.2019.00079. [4] Bissyandé et al. 2013. DOI: 10.1109/ISSRE.2013.6698918. [5] Ray et al. 2017. DOI: 10.1145/3126905. [6] Xie et al. 2021. arXiv: 2102.09066. [7] Sutton et al. 2017. arXiv: 1710.05225. [8] Abdill et al. 2019. DOI: 10.7554/eLife.45133. [9] Citron et al. 2018. DOI: 10.1016/j.joi.2017.12.008. [10] Citron et al. 2015. DOI: 10.1073/pnas.1415135111. [11] Hao. 2019. URL: https://bit.ly/2YXUJlb. [12] Chu et al. 2021. DOI: 10.1073/pnas.2021636118. [13] Azoulay et al. 2019. DOI: 10.1257/aer.20161574. [14] Piwowar et al. 2018. DOI: 10.7717/peerj.4375. [15] Jalili et al. 2020. DOI: 10.1101/2020.11.16.385211. [16] Russell et al. 2018. DOI: 10.1371/journal.pone.0205898. [17] Färber. 2020. DOI: 10.1145/3383583.3398578. [18] Myers et al. 2016. DOI: 10.1109/MC.2016.200. [19] Hunter-Zinck et al. 2021. DOI: 10.1371/journal.pcbi.1009481.

Accelerating Excellence in Scientific Software

Gregory Watson (watsongr@ornl.gov), Addi Malviya Thakur (malviyaa@ornl.gov)

Oak Ridge National Laboratory, Oak Ridge, TN, USA

Challenge: The use of software to facilitate scientific exploration and discovery has changed significantly over the last 50 years. Where it was once used to provide simple mathematical calculations and models, it now forms a key pillar upon which much scientific knowledge is created. However, this increase in the importance and significance of the role of software for science has not been accompanied by major changes to the way scientific software is developed. Unlike commercial software, which has seen an explosion in techniques, practices, and tools to improve software quality, productivity, and usability, scientific software has continued to languish on an island of mediocrity, relying on virtually the same development practices that have always been used. The news is not completely bad, however, and there has been a gradual increase in awareness of quality issues along with a corresponding uptake of some new tools and platforms by scientific computing teams. Unfortunately, the adoption of better practices and development methodologies seems destined to continue to be neglected to the detriment of the community.

In commercial software, issues of quality and usability are very likely to have an significant impact on the viability of a product. This immediate feedback combined with a commercial imperative can be seen as a key driver towards improvements in software quality. However, scientific software typically does not have a similar feedback mechanism, since the experimental nature of the scientific process means that it can be difficult to determine even if it is working correctly. Instead, poor quality scientific software can have repercussions that don't show up for decades, or conceivably, never. The implications can be significant, however, ranging from failure to reproduce scientific results to impacts on public policy and the acceptance of science-based decision making. The gulf between the use of software for scientific discovery, and the ability to accurately reproduce scientific results, may be one reason why quality is often overlooked, but there are undoubtedly other drivers as well.

Opportunity: Without any systematic assessment of the tools and practices that are being employed in the development of scientific software, it is difficult for teams to create strategies for improvement. Since they often don't know what areas need improvement, or even what areas *could* be improved, teams continue to focus on what they do best, which are typically functional improvements in the software. Some attempts have been made to incentivize teams to improve practices, such as the Linux Core Infrastructure Initiative Best Practices Badge [1], the OpenSSF's Security Scorecards [2], and the Software Sustainability Institute's Online sustainability evaluation [3]. The Exascale Computing Project's IDEAS team has also created a simple tool for self assessing project practices [4], and the Extreme-scale Scientific Software Stack (E4S) project has established the E4S Community Policies [5] for its membership criteria. At Oak Ridge National Laboratory, we are experimenting with a more comprehensive self assessment known as the Software Excellence Rating, which combines four components of *originality*, *significance*, *maturity*, and *reproducibility* in order to allow teams to identify areas for improvement and create badges to showcase their achievements. However, there is no coordination or consistency between these efforts.

We believe it would be of significant benefit to the broader scientific computing community to establish a widely accepted Scientific Software Excellence Rating that would a) encourage teams to identify practices and techniques that would benefit from improvement; b) create a pathway for implementing and tracking progress towards this improvement; and c) enable teams to showcase and receive recognition for improving the quality of their software. In addition, such a rating system could be used by journals in lieu of (or as a complement to) their own rating systems (such as the ACM Artifact Badges [6]) or software publication criteria [7], and by funding agencies to demonstrate that software quality is an essential consideration in the public funding of scientific discovery.

Timeliness or maturity: The Exascale Computing Project (ECP) has created a unique environment for raising the awareness of software quality issues in the context of scientific discovery through computing. In addition to bringing together a large number of domain scientists, computational scientists, and software engineers to work collaboratively on both applications and infrastructure, the organizational structure has enabled emphasis to be placed on quality *in addition to* the pure functionality of the software products. The ECP has also established a project team (known as IDEAS) to examine the ways in which the application and software teams could be equipped with the tools and knowledge necessary to lead them to improve the quality and practices of their projects. There have been two significant outcomes of this. First, the application and software teams' awareness of the importance and impact of quality has been raised (this on its own has resulted in increased use of platforms such as GitHub and the tools that are part of these ecosystems). Second, considerable thought has been put into the differences between scientific and commercial software development and how to create tools and practices tailored to the former so that the latter's benefits can be realized.

A second (possibly unintended) consequence of the ECP is that the scientists and developers working on the numerous projects have taken this increased understanding and awareness back to their home institutions (at least those of the authors and author's colleagues) which has precipitated discussion and dialog on the value of scientific software *in its own right.* The possibility that this could lead to a cultural change at these organizations in the appreciation of software and its role in science is a monumental shift if it could be achieved. As the ECP draws to a close, it seems like the time is ideal to capitalize on the knowledge and understanding that we have gathered and use this to empower the next generation of software that will be used to continue our quest for scientific discovery.

References

[1] `https://github.com/coreinfrastructure/best-practices-badge`

[2] `https://github.com/ossf/scorecard`

[3] `https://software.ac.uk/resources/online-sustainability-evaluation`

[4] `https://rateyourproject.org`

[5] `https://e4s-project.github.io/policies.html`

[6] `https://www.acm.org/publications/policies/artifact-review-and-badging-current`

[7] `https://joss.theoj.org`

Building Human-Centered Computing Knowledge of Scientific Software Development

Jason Wiese and Mary Hall, University of Utah

Challenge

High-Performance Computing (HPC) has made incredible advances to expand the boundaries of what is computationally possible, providing a foundation upon which numerous scientific advances are made across fields ranging from predicting the path of hurricanes to discovering therapeutics for Covid-19. Yet, too often HPC tools research takes an “if you build it, they will come” approach to innovation. In the worst case, this results in tools that have little use beyond demonstrating that an idea works and getting a paper published. In the best case, this approach yields a tool that works for some end-users, but does not serve the broad needs of the scientists working in the application domains where these tools could have even greater impact.

Three decades of knowledge amassed in Human-Centered Computing (HCC) suggests a process for building software that seeks to gain a deep understanding of who is going to use it, the context that they are going to use it in, and the other workflows, systems, and processes with which it is integrated. HCC is an umbrella term we use to refer to the broad and overlapping set of user-focused expertise often referred to as Human-Computer Interaction (HCI), User-Centered Design (UCD), or User Experience (UX), among others. Unfortunately, HCC tells us that there is no silver-bullet solution here, no single checklist or set of guidelines to ensure that the software we create is both *useful* and *usable*. Instead, HCC takes the position that the way to create effective software systems is to follow a process of understanding user needs, exploring many possible solutions in the design space, and converging on a solution before beginning any real development. This process is distinct from the more formulaic Software Engineering approach to *requirements gathering*, which assumes that we have already decided what to build. This HCC approach is tried and true in both research and practice, however HCC has been noticeably absent from the development of scientist-facing HPC research.

Opportunity

While there isn’t a silver bullet solution, the broad HCC research literature demonstrates through many examples that there is value in building up domain-specific knowledge and insights to better facilitate the development of software in those domains. For example, HCC literature on medical informatics, educational technology, social computing, and crisis informatics have all developed deeper insights, design patterns, best practices, and guidance for developing effective software in their respective domains. As this knowledge amasses, the quality of software in these domains improves, and it becomes less resource-intensive to design high-quality software as these designs become more broadly applicable. There is a ripe opportunity to develop similar insights in a new and novel HCC domain that bridges the gap between HPC innovation and the effective application of that innovation.

One example of this opportunity is our recent human-centered research investigating the ways that researchers currently use the CloudLab[1] experimental cloud computing infrastructure.

---

[1] https://cloudlab.us/

CloudLab gives researchers access to physical hardware to prototype and test cloud computing architectures and other innovations, rather than a virtualized environment. In this work, we have engaged a range of current CloudLab users with *contextual inquiry*[2], an HCC research method where the HCC researcher takes on the role of an apprentice, learning from the CloudLab user how they do their work with CloudLab through live demonstration. This method is effective at getting users to express what they *really* do, rather than an idealized or aggregate version of what they do that can miss critical details.

While this work is ongoing, it has already led to valuable insights into how CloudLab users engage in their work. For example, we have found that CloudLab users who have participated in our formative work have reported that early on in their work on a project, they are not thinking about the reproducibility of that project but rather "just want to get it working." However, once they do get it working they encounter an inflection point where, rather than taking additional steps to facilitate future reproducibility, they instead run with the success of their experiment and press ahead, even though some simple steps at the right moment could save significant future effort. This insight suggests that there is a design opportunity for building in better support for capturing state and facilitating reproducibility in an automated fashion as the project progresses, rather than retrospectively at the end of a project. Given the broad importance of reproducibility in scientific research, we anticipate that the future insights from this process will yield insights that might also be applicable beyond the specific context of computer systems researchers.

The opportunity to build deep insights into how the cutting-edge HPC technology we create can be crafted into software artifacts with real impact is, or at least should be, as important to this research community as the technology itself. Further, because of the novelty of the technology and the unique problems that it surfaces, this is necessarily a research endeavor.

Timeliness

The complexity and diversity of today's supercomputing architectures has pushed developers towards low-level architecture-specific software. This trend has in turn created significant concerns about whether the HPC workforce will be prepared to migrate applications from one platform to another. Instead of architecture-specific software development, there are many domain-specific programming systems which automate the lowering from high-level specification to architecture-specific code. While a popular research solution to migrating to diverse platforms, domain-specific solutions have not seen the impact on real-world applications we might expect. What is limiting their adoption?

This whitepaper proposes applying contextual inquiry to evaluating domain-specific programming frameworks, similar to the approach in the work on CloudLab. We consider both successful domain-specific frameworks, such as TensorFlow used in machine learning, and those in the scientific computing domain, such as the large number of domain-specific frameworks for stencil computations arising in PDE solvers. What distinguishes a widely-used domain-specific framework from those that have yet to gain traction, and how might the HPC community draw the necessary lessons to build a sustainable ecosystem?

---

[2] Holtzblatt, K., & Beyer, H. (1997). *Contextual design: defining customer-centered systems*. Elsevier.

A Testing Strategy that Supports Scientific Software Sustainability

James M. Willenbring - jmwille@sandia.gov
Sandia National Laboratories, North Dakota State University*

1. The Challenge of Testing a Scientific Software Ecosystem

Testing scientific software packages is challenging. Many test suites that were not designed with modern systems in mind, and therefore are fragile. Some packages have system-level tests, but lack unit and integration testing. Several scientific software projects have insufficient testing of any kind. These kinds of problems are not new and are broadly recognized.

More recently, it has become clear that while testing a scientific software package is difficult, testing an ecosystem of scientific software is far more complicated. Achieving a single state where all of the packages in the ecosystem can coexist (for example, two packages do not require different versions of the same dependency) is challenging. Maintaining this state in the face of new development is much harder. There are two fundamental ways that new development can be dealt with.

The first option is to test release versions of each package. The advantages to this approach are that the software is generally not changing quickly, and release versions tend to be fairly stable. One disadvantage of this approach is that changes come in large batches and it can be time consuming to identify which change led to a failure. After a problem is identified, fixes might be complicated because significant development may have occurred since the change that caused the failure. A second disadvantage of this approach is developers or users may want to use a pre-release version of one or more packages.

The second option is to test development versions of each scientific software package. This approach allows for the flexibility of testing changes individually or in small batches, but requires more complicated communication patterns, policies, and workflows to execute effectively. If one development version changes in a way that breaks another, should the test be allowed to fail, or should the version not be updated until all builds and tests pass? What if a test is failing on only one platform? What if a change to one package reveals a bug in another? What if the team behind one package fails to respond to bug reports? These questions are representative of the many decisions that must be made in calibrating the orchestration of testing for an entire scientific software ecosystem.

2. Designing a Testing Strategy

The ultimate solution includes both release and strategic development testing. While creating a complete strategy for testing is complicated, defining the policies, workflows, and funding model

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the US Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. SAND2021-14650 C

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

to support the strategy is even more challenging. However, until the testing is in place, it is hard to evaluate policies and workflows. Over the past few years, a lot of progress has been made towards improving the stability and sustainability of the scientific software ecosystem via better testing. At the ecosystem level, the Extreme-Scale Scientific Software Stack (E4S) is leading the way with a growing validation test suite that includes sanity tests for dozens of scientific software packages [1]. The most recent release includes 91 scientific software packages [2]. The current model for E4S is to test release versions of packages together, and new versions are incorporated as they are released and tested. The development of E4S and its infrastructure has been transformational, yet much more can be done to find breakages in compatibility sooner than package release time.

Two ECP Software Development Kits (SDKs), those in the Math Libraries (called the xSDK [3]) and Data and Visualization areas, have stood up significant testing at the SDK level, which are subsets of the software packages comprising E4S. While testing development versions of the entire E4S right now is not practical, testing very small subsets is showing promise. Case in point, the xSDK previously tried to stand up testing for development versions of all of the packages comprising the xSDK, and that wasn't sustainable - the workflows and communication patterns necessary to deal with the errors efficiently are currently infeasible. However, attempting smaller subsets of software has been more successful. Using these small cases as building blocks for improved communication, workflows, and funding models it may be possible to extend this kind of testing to the full SDK level.

It may not make sense to try to extend the development version testing the E4S level. It depends on whether or not the increased stability and sustainability achieved at the SDK level is sufficient to promote development version workflows where needed at the E4S level. Note that using the development version of everything at the scientific software ecosystem level is likely not necessary. For example, an application may want to evaluate a change to a linear solver or a development tool, but not necessarily at the same time.

3. Conclusion

Improving the testing of the scientific software ecosystem gives users greater confidence in using scientific software packages, and makes the ecosystem and its components more maintainable for scientific software developers. Great strides have been made in recent years with E4S and the math libraries and data and visualization SDKs, but more work is necessary to improve testing robustness and efficiency and to determine more optimal workflows and policies. Fortunately, we are now in a position to learn from and build on these efforts.

References

[1] E4S 2021.11 Release Notes. https://e4s.io/talks/E4S_21.11.pdf.
[2] E4S Validation Test Suite. https://github.com/E4S-Project/testsuite.
[3] xSDK Homepage. https://xsdk.info/.

Title: Engineering Scientific Workflows for the Scientists of the Future
Authors: Greg Eisenhauer (Georgia Tech, eisen@gatech.edu), Jeremy Logan (ORNL, lot@ornl.gov), Patrick Widener (Sandia, pwidene@sandia.gov), Matthew Wolf (ORNL, wolfmd@ornl.gov; corresponding author)

Challenge: The scientific software ecosystem has become radically decentralized. Construction of scientific software, long reliant on monolithic (and potentially closed source) infrastructures, now is based on collections of rapidly evolving open source software solutions. This changed ecosystem provides many opportunities for customization, but in doing so it has abandoned much implicit management of technical debt. The software engineering practices which allow you to develop HPC and data analysis codes on your laptop and deploy them to a leadership-class machine are not suited to the evolving practice of collaborative computational science. In the modern ecosystem, scientific software systems interconnect: codes are coupled, workflows are streamed, and data connects with computation in many other ways that use the broad label 'workflow'. Considerable investment in a well-developed and well-understood spectrum of software engineering tools ushered the scientific community to this point. Widespread application of those tools to such connected software (and connected software development, and connected software developers) amplifies the accumulation of software technical debt. This debt in turn increases the friction associated with collaboration and workflow reuse.

For example, even our small team has seen examples where such workflow engineering issues have arisen in fields as diverse as exascale computational fusion, computational biology, extreme-scale material science ensembles, and national security engineering computations. As an exemplar, consider the machine-learning-enhanced computational biology workflow from Jacobson *et al* [Genes2019]. The end goal is to use the ML algorithms to generate best-in-class knowledge graphs of the complex biological pathways that can lead to drug discovery, pathology, and more. As a workflow software engineering task, that means devoting substantial graduate student and professional time to seeking out each new best-in-class genomic labeling scheme, incorporating it into the pre-ML labeling workflow, and then fixing any performance or functionality bugs that arise from the idiosyncrasies of data layout, format, scalability, etc. before being able to run the workflow. The management of that technical debt in reuse of others' workflow components is a substantial distraction from the science focus that could otherwise be occurring [Cluster2021]. We need frameworks and ecosystems built around the way that such sharing and reuse works for real science codes.

Opportunity: Scientific software is a realization of an experimental hypothesis, and reuse of scientific workflows is a realization of the experiment/evaluate-hypothesis/share-results loop. The utility of models for individual researchers can/should be extended to address the needs of the larger scientific community. Scientific technical debt is under-managed, in part, because the approaches we take do not address root causes, including the different incentives which apply to science's use of software versus commercial software development processes. Planning, communications, expectations of reproducibility and transportability of results, and transparency of process are different in science. The sociological structures of science communities are different from those of commercial or even other non-profit sorts of

structures, and there needs to be more investigation of how those differences do, could, and should impact our scientific workflow development processes.

Scientific software frequently, if not always, is written with a larger workflow pattern in mind. Going forward, the opportunity is to use such patterns to help entangle software and workflow development, and to manage both explicitly. Without such explicit management, developers find themselves in an all-too-common situation, similar to that discussed in the bioinformatics challenge above, of knowing that one has a license, source code, and data sets from a workflow, and yet having no guarantee that 3 post-docs won't need a year and half to get it working. We need to better understand how to make the ecosystem of software and workflow development tools interoperate so that build requirements, runtime constraints, and data provenance can all be managed in an automated or at least semi-automated manner. Better collection and maintenance of metadata of the software development process, and of the roles and targets for the larger workflow, can enhance our understanding of the real practices of scientific workflows.

As a specific technical opportunity, model-driven approaches to code generation can reduce technical debt in code artifacts and lead to more reusable workflow components. Model-centric approaches allow users to drive such components through domain-specific models designed to allow users to describe what needs to be done without needing to provide the programming that determines how it is to be done. Discovering techniques that allow creation of such models and corresponding templates that can be used for code generation will require a shift of mindset for scientific developers, as well as further study of this relatively new approach. To best take advantage of model-driven techniques, some rethinking of our traditional software engineering roles is needed. For example, the process of generalizing an existing workflow component may be of little benefit to a developer who wishes to use it for a single purpose, but of great value to future users with other purposes in mind.

Timeliness: Our current approach to science workflows is untenable and causes great friction in the practice of science. The science of scientific workflow engineering is in its infancy, even compared to the state of scientific software engineering. The state-of-the-art is undergoing rapid and increasing change, driven by unpredictable factors such as the degree to which the goals of industry players overlap with those of researchers. Thus, further research and care now is important to assure that the science workflows of the future can be built so as to leverage the way that our scientists collaborate, share, and reuse results.

References:

[Cluster2021] M. Wolf *et al*., "Reusability First: Toward FAIR Workflows," 2021 IEEE International Conference on Cluster Computing (CLUSTER), 2021, pp. 444-455, doi: 10.1109/Cluster48925.2021.00053.

[Genes2019] A. Cliff, *et al*., "A high-performance computing implementation of iterative random forest for the creation of predictive expression networks", Genes, vol. 10, no. 12, pp. 996, 2019.

Software Monsters: Quantifying, Reporting, and Controlling Composite Applications

Justin M. Wozniak, Computer Scientist
Data Science and Learning Division
Argonne National Laboratory
(630) 252-3351 — woz@anl.gov

Challenge

Exciting new scientific applications that are rapidly developed to attack new, critical, and dynamic application spaces are *predominantly* large composite applications (or workflows) that integrate a great deal of software together. New computational paradigms, such as the prevalence of maching learning techniques, uncertainty quantification, and design optimization add to the importance of programming at this level. Applications thus face challenges when integrating the significantly different paradigms of high-performance computing (HPC), big data analysis, and the machine learning toolboxes emerging today. We propose that fundamental software metrics can be brought into innovative programming models to address the construction and execution of scientific applications.

Consider the recent (2020-2021) Gordon Bell finalists for the COVID-19 Special Prize. These applications are generally phrased as workflows, as they combine ensembles of simulations coupled with learning, analysis, and visualization. One such application with an complex machine learning component [1] utilized a custom R installation to perform a novel multi-objective optimization scheme; the R library size to perform these optimizations contains 112 packages totaling 321 MB, and this was for software generally understood and directly used by the developers. More opaque packages such as TensorFlow reside in Python installations totalling 20 GB or more.

These *monstrous* software dependencies and integrations pose a critical challenge to forward progress in scalable scientific applications. They create difficulties maintaining and validating scientific results over non-trivial time scales, and must be better understood.

Opportunity

"If you can't measure it you can't manage it."

The first step toward a scientific study of changes in culture affecting scientific software development and use is quantifying those changes. We propose that a software **score report** can be used to rapidly convey key statistics about the software used by a composite application. But the key observation here is not the static report itself but the *changes over time* to capture developments that involve new risk to an application. Such reports could be automatically generated by a system that understands software dependencies (e.g., Spack [2] or an extension to it).

Traditional software metrics include lines of code, code points, number of packages, the compiled size of packages, and code coverage metrics. Key questions that can be addressed herein include:

1) How many lines of code are written by the user?
2) How many interface points are there between user code and library code?
3) What is the code coverage of the user and library code?
4) What is the code coverage of the user and library code on the current hardware?
5) What is the code coverage of the user and library code on the current application problem?

The measured numbers, however, should not be immediately interpreted as good or bad but monitored for trends over time. These can be collected with respect to changing application structure as components are added or removed and compared against, for example, test suite results. We envision that they could be captured by automated continuous integration tools (e.g. GitLab).

"What just broke?"

A second step toward is the construction of software **structure graphs** using AI-based techniques. Deep learning systems have been used for graph analytics for some time, and can be used to expose underlying features in structure and key latent variables. [3] The graph structure of dependencies and calls in a composite application could reveal metrics for change over time, while also exposing structural qualities (broad vs. deep structures) and the encounter of new structures not seen before, revealing the associated risks and opportunities. Structure graph fingerprints could thus be compiled and exchanged. These graphs allow software metrics from software dependencies to be immediately made available to higher-level packages, thus making score reports easy to generate and understand as software changes.

Software structure graphs are a first step toward AI-driven development tools and best practice identification for scientific software development. Structure graphs collected from ECP-related applications along with user-labeled tags could be analyzed to rapidly reveal software structural practices that are likely to produce good outcomes. At the lowest level, the graph would contain entries for vendor-provided tools for messaging, operating system services, and so on. Structure graphs could be extended to include hardware components, including accelerators or other exotic devices.

Structure graphs could also be used to assist in the automatic generation of auto-tuning schemes for performance, accuracy, or multi-objective schemes [4]. Identifying the most critical and/or sensitive parameters in library software is challenging. Composite application tuning is thus a structured optimization problem in which some subproblems have already been (partially?) solved. If these parameters and knowledge bases could be exposed to higher-level packages, auto-tuning problems could be automatically generated from the structure graphs, while gaining the acceleration from prior dependency optimization runs.

We stress that this kind of reporting could be seen as time-consuming and tedious. To bring AI into the software analysis loop, however, we see that some fundamental metrics must be exposed for automated analysis, and this workshop could be critical in identifying those.

TIMELINESS

"Given enough eyeballs, all bugs are shallow."

This type of software analysis is needed immediately as the rapid adoption and longevity of advanced machine learning libraries developed by small disparate teams and/or corporate giants become critical to scientific applications. These projects must be manageable, and fundamental to that is understanding application structure.

The expected outcome of this approach will ease the development and validation of complex studies that integrate domain science with data science and machine learning techniques. Multiple proposed computer science investigations will result, including fundamentals in AI understanding and information extraction from reports and graphs, as well as software challenges in producing the raw data from existing package managers. The work proposed here will lay the groundwork for future programming models and methods that will allow a wide range of application domains to rapidly adopt hierarchical programming and reduce the complexity and of application-specific testing and evaluation.

This level of programming will become critically important as complex model exploration studies and deep-learning-infused workflows are deployed on exascale systems. In the absence of a comparable systems and methodologies, researchers will have difficulty developing and deploying composite applications. Developments in community software packages will be unmanageable. Software management for advanced scientific computing will be left in the human bandwidth-limited status quo.

REFERENCES

[1] J. Ozik, J. M. Wozniak, N. Collier, C. M. Macal, and M. Binois, "A population data-driven workflow for COVID-19 modeling and learning," *International Journal of High Performance Computing Applications*, 2021.

[2] T. Gamblin, 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 *Proc. SC*, 2015.

[3] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk," *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug 2014. [Online]. Available: http://dx.doi.org/10.1145/2623330.2623732

[4] T. Shu, Y. Guo, J. M. Wozniak, X. Ding, I. Foster, and T. Kurc, "Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications," in *Proc. SC*, 2021.

Beyond Open Source: A Call to Rethink Policies and Incentives for Sustainable Scientific Software Development

John Wu, LBNL, KWu@lbl.gov

Much of the high-performance computing (HPC) software ecosystem is built on top of free open-source tools (1). For example, many of the HPC systems use variants of Linux as their operating systems, Lustre as their file systems, GNU C compiler suites as their compilers, and LAPACK as their core math libraries. Aside from these foundational tools with wide usage outside of HPC systems, there are many more specialized tools critical for DOE scientific researches. These tools include high-performance simulation tools for physics, chemistry, biology, materials science, as well as data analysis tools for processing images from astronomical observations, x-ray light sources, neutron sources, and so on. These software tools are critically important for recent scientific discoveries including gravitational waves (2) and Higgs boson (3). These tools also rely on a large number other tools for their operations. A recent accounting of Spack packages (4) from the Exascale Computing Project (ECP) indicates that a handful of ECP simulation tools depend on thousands of other packages, most of which are open-source. The success of DOE science rely on these software packages. Continuing the development and maintenance of these tools require a thriving community of tools and developers. Open Source as a development paradigm has served us well so far. However, the developers of these Open Source Software usually could not depend on their software to make a living and therefore have to find ways to support themselves, which creates tension in the development community. This position paper is a call for more flexibility in valuing scientific software, which we believe would lead to more ways to support scientific software development.

Software Sustainability: State of Art

Innovation in scientific software is highly valued, in contrast, the maintenance efforts are not receiving nearly as much attention. However, both development and maintenance efforts are critical to a thriving science community. In the above mentioned example from ECP, where ECP funds the work of a couple of dozen of software products, however, these software products relies on thousands of supporting packages. As these packages are used on new HPC systems and in new application scenarios, these supporting packages would require updates, corrections, and modifications. The key software sustainability challenge is "how to support these maintenance efforts required for these thousands of supporting packages?"

Software Sustainability is a long-standing research topic (5). Much of the published literature focuses on how to architecture the software system and similar technical issues. At the same time, there are also a number of studies pointing to the importance of non-software issues such as license and citation. For example, Midha and colleagues found that license choice affects both technical successes as well as market successes of an open source software project (6), and a number of authors have written about the importance of citation and credit of software tools (5). Another broad categories of issues sometimes discussed is community building (7). However, we believe the following observation from the Software Sustainability Institute really captures the core issue, "reward" (8):

> Many researchers are yet to be convinced of the importance of developing well-engineered software. Although we might disagree with this viewpoint, it's an understandable one, because the research community provides little - if any - reward for producing such software. A lack of reward leads researchers to choose quick fixes over a more considered, maintainable approach to development.

Therefore, our main thesis is to propose more research into different "reward" mechanisms.

Exchange Platform and Evaluation of "Reward" Mechanisms

The existing software sustainability research touches on a number of aspects related to rewards. For example, license could be used to gain monetary rewards and citations is a form of reputation reward. Given that much of the scientific research are supported by grants with a limited ability to pay for software tools, there is strong bias preferring free open source license with no virial restrictions. However, this is not always the best option, even for esoteric software packages specifically designed for molecular dynamic simulations. There are many open source versions of molecular dynamics simulation packages (9), but also a number of commercial packages that perform the same tasks (10). In fact, we are aware of a senior scientist who recently resigned from a large open source development effort to start a commercial venture to market a set of molecular dynamics software. We take this as an evidence of the tension between divergent choices available at the moment. We don't believe the currently available choices are the only ones. Could we devise additional choices that might better serve the scientific community?

Based on the theoretical framework developed by Morgan et al (11), we propose to explore different reward mechanisms by conducting large-scale simulations. The simulator will track the operations of a large software trading/exchange platform with the ability to employing different utility assignment algorithm for software packages (12). The ideal outcome would be to develop a reward mechanism to enable the software community to self-sustaining. Most likely we might identify a number of mechanisms that might be able to achieve this ultimate goal, it would then be necessary to design a real software exchange to share the software development expertise and the resulting software tools. As with those platforms for the sharing-economy, we anticipate this exchange mechanism to help us further explore the reward mechanism for a self-sustaining scientific software community.

References Cited

1. Bangerth W, Heister T. What makes computational open source software libraries successful? Computational Science & Discovery. 2013;6(1):015010.
2. Barish BC, Weiss R. LIGO and the detection of gravitational waves. Physics Today. 1999;52:44-50.
3. Spira M, Djouadi A, Graudenz D, Zerwas R. Higgs boson production at the LHC. Nuclear Physics B. 1995;453(1-2):17-82.
4. Gamblin T, LeGendre M, Collette MR, Lee GL, Moody A, Supinski BRd, et al. The Spack package manager: bringing order to HPC software chaos. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; Austin, Texas: Association for Computing Machinery; 2015. p. Article 40.
5. Katz DS, Choi S-CT, Lapp H, Maheshwari K, Löffler F, Turk M, et al. Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1). arXiv preprint arXiv:14047414. 2014.
6. Midha V, Palvia P. Factors affecting the success of Open Source Software. Journal of Systems and Software. 2012;85(4):895-905.
7. Boettiger C, Chamberlain S, Hart E, Ram K. Building software, building community: lessons from the rOpenSci project. Journal of open research software. 2015;3(1).
8. Crouch S, Hong NC, Hettrick S, Jackson M, Pawlik A, Sufi S, et al. The Software Sustainability Institute: Changing Research Software Attitudes and Practices. Computing in Science & Engineering. 2013;15(6):74-80.
9. Jing X, Troullier NR, Dean D, Binggeli N, Chelikowsky JR, Wu K, et al. Ab Initio molecular dynamics simulations of Si clusters using the higher-order finite-difference-pseudopotential method. Physical Review B. 1994;50(16):12234-7.
10. Hafner J. Ab-initio simulations of materials using VASP: Density-functional theory and beyond. Journal of Computational Chemistry. 2008;29(13):2044-78.
11. Morgan L, Feller J, Finnegan P. Exploring value networks: theorising the creation and capture of value with open source software. European Journal of Information Systems. 2013;22(5):569-88.
12. Chli M, Wilde PD. The Emergence of Knowledge Exchange: An Agent-Based Model of a Software Market. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans. 2008;38(5):1056-67.

Metaverse for Urban Informatics: A Solution towards Participatory Smart City Applications

Haowen Xu (ORNL, xuh4@ornl.gov), Andy Berres (ORNL, berresas@ornl.gov),
Srikanth Yoginath (ORNL, yoginathsb@ornl.gov), and Jibonananda Sanyal (ORNL, sanyalj@ornl.gov)

Challenge - The design, development, and use of scientific software in the urban science domain often face sociotechnical challenges. The goal of building smart and sustainable cities is to improve the overall life quality of citizens (residents) in urban spaces. It is critical to embrace the social dimension, which focuses on the interaction between human society and the changing urban environment (Calzada & Cobo 2015; Nasirzadeh et al., 2020). Smart city apps are software tools developed to address theoretical understanding, operations and management, long-range decision-making, and urban policy assessments of various complex urban sub-systems (e.g., transportation, buildings, health, water, and disaster management) within the same system (Thakuriah et al., 2017). However, many smart city apps are developed in a way that separates domain experts and developers from citizen users (Moulaert, 2013). This prevents an open, collaborative, and participatory software environment that allows citizens to dynamically connect, co-create, and share visions with planners, policymakers, scientists, and developers (Calzada & Cobo 2015). This dilemma is common in urban and environmental management applications, where the consent and opinion of citizens are essential to the decision-making processes (e.g., the management of urban disaster and land requires support from local landowners). It limits the acceptance, usage, and life cycle of many smart city apps among citizen users (Yeh, 2017). Thus, many urban science software tools remain at the proof-of-concept level without creating broader impacts and application values. Although cities worldwide have proposed visions and initiatives to increase citizen participation and collaboration in decentralized smart city governance (Simonofski et al., 2021), very few studies have come to effective technical and methodological strategies to incorporate the social dimensions in smart city apps (White et al., 2021). Existing software practices in urban informatics only offer very limited capability to involve citizen participation and interaction in specific sectors of city management.

Opportunity - We believe the recent emerging "Metaverse" concept could be integrated with smart city apps to overcome sociotechnical challenges, and improve the development and use of urban science software. The concept is a combination of "meta" (meaning beyond) and the stem "verse" from "universe", denoting a hypothetical synthetic environment linked to the physical world (Lee et al., 2021). In the context of smart city apps, this synthetic environment could be fully immersive, allowing the citizens to freely explore and interact with complex urban science dynamics and concepts, which may affect their future life, through intuitive 3D scientific visualizations and visual analytics in virtual spaces. For instance, citizens could view models of proposed construction of new buildings, roads, or parks; they could see visualizations of invisible variables such as air quality, wind, or weather predictions. A human-centered and decentralized system design will create opportunities for the computational science community to study and evaluate the use and effectiveness of their urban science software. The new approach can monitor how the citizen user's avatars interact with the smart city services and respond to hypothetic scenarios simulated in digital twin cities. The user-input data naturally reflect users' opinions regarding the design of smart city apps and their underlying urban management decisions (e.g., plans of new roads and traffic rules, demands of water distribution systems, and new designs of street landscape). Through avatars, citizens could also directly communicate and closely interact with urban planners and scientists, delivering their' feedback, emotions, evaluations, and visions for improving the design of the physical city, its operations, and the smart city app itself. A detailed example could be the request to create or improve science-based smart city services to optimize urban mobility or building energy use. Citizens could view and interact with different road layouts or building designs, and provide their feedback to the city. This practice creates a dynamic and transparent bridge of trust, positive relationships, and communication between end-users and smart city builders (urban planners, policymakers, scientists, and app developers) throughout the life cycle of a smart city app. In this setting, the developers could invite end-users to co-develop and evaluate features of urban science software, guiding the scope of smart city apps to maximize the interest and satisfaction of their citizen users. The practice aims to improve the acceptance and trust of smart city apps among citizens.

Benefits - The participatory nature of the metaverse concept allows its technologies to be seamlessly synchronized with many well-established urban research and management practices, such as shared-vision planning, social learning, serious gaming, and neighborhood games. This leads to a new software development method based on the scientific gamification concept to translate complex and elusive science components into attractive and self-educating gaming components through a metaverse environment. The method can effectively promote the social dimension in smart city apps. The gaming components employ visually explicit and abstracted representations, easy-to-understand rules of user interactions, and extended reality technologies. They thereby facilitate users' human reasoning and cognitive capability, allowing them to understand complex urban science concepts and to engage in multi-sector decision-making workflows. A small-scale real-world prototype of the described vision is tested by Dembski et al. (2020) for a small German town. The prototype's features can be further evolved using the metaverse concept to benefit major metropolitan areas and position the United States in a leading role in next-generation smart city applications.

Timeliness and Maturity - The metaverse has been a hot topic recently among major Information Technology (IT) and Artificial Intelligence (AI) companies. The concept captures the leading tech industry's imagination and investments, and has been predicted as a major trend of the next-generation internet. As the industrial sector drives the implementation of many previous smart city applications, the metaverse trend will eventually transcend from the industry to urban science and informatics sectors in academic environments. Additionally, the vision of metaverse is catalyzed by emerging technologies, such as 5G networks, edge-computing paradigm, Internet of Things (IoT)-connected infrastructure, and AI, paving the way to future digital transformations in our cyberspace. Major blockers towards a metaverse-powered smart city are the availability of 3D urban data, virtual/augmented reality hardware, and high-speed internet. We believe that these blockers will be resolved in the near future with more industry investments in metaverse technologies and infrastructure investments throughout the United States following the recent bipartisan infrastructure bill.

References

1. Calzada, I., & Cobo, C. (2015). Unplugging: Deconstructing the smart city. Journal of Urban Technology, 22(1), 23-43.
2. Dembski, F., Wössner, U., Letzgus, M., Ruddat, M., & Yamu, C. (2020). Urban digital twins for smart cities and citizens: The case study of Herrenberg, Germany. Sustainability, 12(6), 2307.
3. Lee, L. H., Braud, T., Zhou, P., Wang, L., Xu, D., Lin, Z., ... & Hui, P. (2021). All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda. arXiv preprint arXiv:2110.05352.
4. Moulaert, F. (Ed.). (2013). The international handbook on social innovation: collective action, social learning and transdisciplinary research. Edward Elgar Publishing.
5. Nasirzadeh, F., Ghayoumian, M., Khanzadi, M., & Rostamnezhad Cherati, M. (2020). Modelling the social dimension of sustainable development using fuzzy cognitive maps. International journal of construction management, 20(3), 223-236.
6. Simonofski, A., Hertoghe, E., Steegmans, M., Snoeck, M., & Wautelet, Y. (2021). Engaging citizens in the smart city through participation platforms: A framework for public servants and developers. Computers in Human Behavior, 106901.
7. Thakuriah, P. V., Tilahun, N. Y., & Zellner, M. (2017). Big data and urban informatics: innovations and challenges to urban planning and knowledge discovery. In Seeing cities through big data (pp. 11-45). Springer, Cham.
8. White, G., Zink, A., Codecá, L., & Clarke, S. (2021). A digital twin smart city for citizen feedback. Cities, 110, 103064.
9. Yeh, H. (2017). The effects of successful ICT-based smart city services: From citizens' perspectives. Government Information Quarterly, 34(3), 556-565.

AI-assisted Collaboration between Domain Scientists and Research Software Engineers

Chen Zhang

November 2021

Scientific software is a critical aspect of modern scientific research as most domains have incorporated scientific computing as part of the standard research practice. However, most of domain scientists are struggling with finding their domain specific software to facilitate their on-going research, forcing them to write one-off software packages themselves or to hire graduate students and post-docs to carry out the development. Due to the extremely narrow application and the project-driven development process, these software packages are often orphaned after the parent project ends or the original developer leaving due to graduation or job switching. The situation described above leads to the current state of scientific software where countless new ones are being produced and published on open platforms like Github, the majority of which will be abandoned within a year.

For domain scientists, the current state is an unfortunate but necessary compromise as their priority should always be carrying out research, not publishing production level software that can be maintained for long term usage. Furthermore, most domain scientists, including their graduate students and post-docs, are not professionally trained to write software in an easy-to-use and maintainable way. Consequently, the resulting software often has a steep learning curve and can only be updated or modified by a handful of people.

As an attempt to resolve this dilemma, domain scientists are starting to reach out to professional scientific software developers, also known as research software engineers (RSE), to help with the corresponding software development. This leads to some successful scientific software that are still being used today. However, the collaboration is not always a smooth process due to the knowledge gap between the domain scientists and the RSE. Generally speaking, it is very rare to find an RSE that has the same level of domain knowledge as the scientist. Therefore the software developed by the RSE might not be what the domain scientist has in mind due to communication issue. Additionally, the standard development framework and cycle adopted by most RSE often goes against the tight time table most domain scientists are facing, make an already complicated process even worse.

The obvious long-term solution to the problem described above would be having universities and research facilities continuously producing a steady stream

of RSE with necessary programming skills as well as sufficient background knowledge in various fields. However, it would require years of program planning and execution at the national level to achieve such state whereas the demands are in the present. One possible short term solution would be utilizing commercially available AI based toolkit to facilitate the collaboration between domain scientists and RSE. More specifically, AI-based code generator like Github's co-pilot can create sample code snippet based on comment (via natural language processing) as well as existing code. If the co-pilot type assistant can be re-trained to specialized in different domain, it is possible to revamp the collaboration between domain scientists and RSE into the following workflow:

- Domain scientists uses AI assisted toolkit to generate sample code snippets
- RSE help domain scientists to further develop these code snippets into prototyping scripts that are capable of producing required data
- The domain scientists can use the prototype to continue with research while the RSE can use the prototype as base to start a proper software development cycle.
- As new needs arise during the research process, domain scientists can create new snippets that RSE will help to convert into executable prototype. While the domain scientists continue their research with the new prototype, RSE can update the software development objective so that the new feature can be integrated into the development cycle.
- Once the software development reaches a certain milestone, RSE will help domain scientists to switch from prototype to the actual software.

The proposed workflow above essentially make the research (carried out by domain scientists) and the software development (carried out by RSE) into asynchronous processes, allowing both to run in parallel without too much interference. However, there are still several logistical issues need to be resolved for this process:

- Who is the ultimate owner of the final software? The owner is often responsible for maintaining and offering technical support for users, and this kind of work requires both deep knowledge of the software as well as continuous funding support.
- What is the proper copyright for code that is generated by the AI code generator, especially when the code could end up in the final product?

Despite the unresolved issues above, the AI assisted collaboration does offer a viable temporary solution to continue supporting the scientific research with quality scientific software until sufficient amount RSE with wide range of domain knowledge entering the research fields.

Exploring Generic Metaprogramming to Manage Complexity

Hui Zhou
Mathematics and Computer Science Division
Argonne National Laboratory

November 2021

1 The Science of Scientific-Software Development

There are plenty of science in the subject of software development itself. For example, programming language theory, compiler theory, or dependency management. We, in fact, have been paying great attention to the science part of the software development, as in computer science. However, the practice of software development is ultimately an art. As an art, it is the domain of science of human perception and creativity, which, in my opinion, has been largely neglected. So this paper focuses on the latter, specifically, how to manage complexity.

2 Challenge

The complexity in scientific software is rapidly increasing. When we grow our project and continuously optimizing our software, each new feature and optimization we add injects complexity. Moreover, the system environment for running scientific software is getting more heterogeneous. In today's leadership computing facilities, CPU+GPU hybrid architecture become common. Performant scientific software needs to pay attention to many hardware details—from CPU vectorization and memory locality to accelerator offloading and multiple node scalability. Even with adopting runtime systems and performance-portable frameworks, which themselves have been growing more diverse, we are still facing more complexity rather than reducing it.

Two aspects of complexity deserve particular attention. First, seeking a solution that automatically generates complexity cannot be the sole direction. For example, compiler innovation can remove some lower-level complexity, but much of the complexity that addresses the problems in a research software projects are often unique and necessary that cannot be covered by a general solution. Second, the problem of complexity is not really about the amount of complexity; rather, it is about how developers comprehend and manage the complexity. Developers can deal effectively with complexity only one part at a time and one layer at a time. Each part inherits context that we already comprehended, and each layer provides context to tackle the next layer.

Today's software projects are often not human friendly. If we sample some source code from a random project, we often see functions with hundreds of lines of code with dozens of variables, or we see dozens of small functions with little context about the purpose. In either case, such functions cannot be readily comprehended even with sufficient domain experience. To effectively work on a software project today often requires sufficient experience with the code itself. It has become increasingly difficult as complexity grow.

3 Opportunity

Today's software practice is overwhelmingly bottom-up driven. This is true even when we try to plan from the top down. For example, as soon as we decide to use MPI, the central concern becomes how to partition the problem and how to communicate between partitions. A domain scientist will not recognize equations when the code is dealing with partitions and communications everywhere. Even with a math library such as PETSc, the top-layer code is likely dealing with partitions with many PETSc-specific terms. A numerical expert may feel helpless without being very familiar with the particular library the project has adopted.

What if we can keep the library-specific terms to the bottom layers, separating e.g. the partition problems from the concepts and equations, so that the domain scientists and performance engineers can each focus at

their respective familiar layers? Or, what if scientists can start with their equations, learn some basic parallel performance concept, move on to the next layer, learn some more, and move farther down?

Clearly needed is a generic metaprogramming framework. For an early example of generic metaprogramming we can look at Donald Knuth's literate programming [1]. However, literate programming puts too much emphasis on explaining the logic using natural language. This places a heavy burden on programmers, who often are more adept at explaining logic directly with code—albeit often as pseudo code. An alternative solution is MyDef, a project that I have been developing for a while. MyDef focuses on organizing the code according to how our comprehension flows. With MyDef, we can write code one layer and one part at a time, replacing complex details with vocabulary that is delegated at separate layers. The framework then weaves all the parts and layers together and reorders the code in the order that the compiler mandates, sometime supplying the missing syntactical ingredients that are noisy during composition but necessary for the compiler to compile.

Here is a simplified example:

```
program: main
    $call init
    $call compute
    $call output

subcode: init
    int r = 10;

subcode: compute
    int area = 3.14 * r * r;

subcode: output
    printf("The area of a circle with r = \%f is \%f\\n", r, area)
```

Just with the `subcode` facility we are able to write the code in 4 contexts, each of which deals only with limited ideas. The very top layer—the main layer—is in fact language neutral. The context is inherited automatically similar to C macros, but it is conveyed more as vocabulary, e.g. `init`, `compute`, and `output`. Imagine that we can keep adding complexity—each `subcode` can spawn more `subcode`s—but the top-layer code remains comprehensible with its complexity contained. In addition, each `subcode` can be nested or rearranged in arbitrary ways, allowing us to arrange the code according to the needs of logic and flow.

4 Timeliness and Maturity

I have been developing MyDef [2] and used it for most of my projects for many years. The framework is usable as a thin wrapper that programmer can still write code in the conventional way. But to reap the potential, it gradually requires a new perspective. For example, programmers are more expected to read code like pseudocode in that some of the details are expected to be supplied either from upper layer context or by lower layer implementations. In practice, especially during debugging, it is common to work with both the source in MyDef and its final form, for example, in C, side by side. The former provides higher-level understanding, and the latter provides feedback and interface with language specific debugging tools. Unlike many other metaprogramming frameworks, MyDef aims at non-compromising target code. The target code ideally will be the same code one would write without using MyDef, including styles, thus still suitable for human peruse. It is simple to configure code editing environment, e.g. vim, to have both views synchronized within a key press. This is, in fact, a common practice that fits how our mind works. For example, thinking of the concept "apple" and feedback with an actual "green apple." The former focuses on the essential idea, where the color matters little. The latter provides feedback and will alert us when the color does rise to some significance.

MyDef currently is still a research prototype with very limited adoption. It is used in developing and maintaining MyDef itself. It also has been used in developing an in-house FDTD optical simulation software used in my previous research at NIST [3]. While I am convinced of its potential in managing software complexity, it remain to be tested by larger software projects.

In a much larger-scale software project, the need to manage complexity is, in fact, more imperative but will also face steep adoption challenges. Today, with rapidly mounting complexity, and as we search for novel solutions, I believe it is a good time to seriously explore the idea of generic metaprogramming.

References

[1] Donald E. Knuth. "Literate Programming." In: *Comput. J.* 27.2 (1984), pp. 97–111. URL: http://dblp.uni-trier.de/db/journals/cj/cj27.html#Knuth84.

[2] Hui Zhou. *MyDef - A Generic Preprocessor*. URL: https://github.com/hzhou/mydef.

[3] Bryan M. Barnes et al. "Assessing Form-Dependent Optical Scattering at Vacuum- and Extreme-Ultraviolet Wavelengths of Nanostructures with Two-Dimensional Periodicity". In: *Phys. Rev. Applied* 11 (6 June 2019), p. 064056. DOI: 10.1103/PhysRevApplied.11.064056. URL: https://link.aps.org/doi/10.1103/PhysRevApplied.11.064056.

Progress in the Study of Scientific Software Requires a Rigorous Validated Taxonomy

Sagi Zisman (sagi.zisman@nrel.gov), Robert White (robert.white@nrel.gov),
Jordan Perr-Sauer (jordan.perr-sauer@nrel.gov)
National Renewable Energy Laboratory

1 Challenge

Software Engineering (SE) is a young field with constantly evolving ecosystems, processes, and best practices. As a result, industry adoption and ad-hoc experimentation tends to drive activities and decisions. Positively, this tendency has created one of the fastest adoption rates for any technological achievement in the history of humanity which has led industry champions such as Marc Andreessen to claim that "software is eating the world" [2]. A downside of this phenomenon is a culture around software development that seems hesitant to adopt evidence-based practice. Surprisingly, one study shows that a majority of scientists believe that formal education in SE is relatively unimportant [4].

Scientists are increasingly developing software as tools for their studies [4]. This requires domain scientists to become software *practitioners*. Ideally the practice of SE follows researched recommendations in SE (evidence-based practice) in applicable areas, such as requirements or testing, to ensure the highest-quality reproducible *domain* research is performed. As each type of scientific software may have different requirements, a starting place for implementing best practices is to identify, enumerate and classify the types of software subject matter for effective adoption of relevant research. Without a classification scheme, it is difficult to see how the varied and complex SE subareas are connected which hinders our understanding and, thereby, presents a fundamental challenge for the adoption of evidence-based SE.

We define a taxonomy as a methodology and tool suited for classifying knowledge. Classification schemes support the development of scientific theories and are important in the conceptualization of a body of knowledge [5]. The benefits of developing a taxonomy are threefold: researchers can identify gaps in research and understand the generalizability of their research, funding agencies can assess where to focus research funds, and practitioners can identify and adopt applicable best practices [9]. We believe that as a maturing, dynamic field, the SE knowledge base needs a taxonomy specific to scientific software engineering. ***Our position is that different types of scientific software need different validated taxonomies and are essential for the study of scientific software engineering***.

1.1 Challenges in Scientific Software Engineering Classification

Scientific software covers a variety of areas with recent focus tending to be on data analysis. But there are other key areas of scientific software that can play a critical role by their direct effect on the accuracy and validity of the data produced. As an example, instrument control software can directly affect the data quality that feeds analysis software. In many cases each of these software areas may have specific taxonomies based on the type of work performed. For instrumentation control, a taxonomy could be quite complex since it requires the integration of several different software elements, and thus, requires an integration of several taxonomies (e.g., Safety monitoring, command and control, and data security). A non-exhaustive look at some of the types of scientific software that may need to interact can be broken down into several general categories listed below:

- Instrumentation command and control
- Integrated safety monitoring
- Laboratory Information Management Systems (LIMS)
- Data security and management
- Data analysis and visualization

1.2 Demonstrating the Complexities of the Software Taxonomies

Taxonomies can be complex and difficult to construct, and we provide an example of developing such a taxonomy: an experimental apparatus through four steps as outlined by Usman, Britto, Börstler, and Mendes [8]. In the *Planning* step, a researcher and a software engineer need to establish clear requirements, architectural view, methods, techniques, and limitations of the software. Then by *Identification and Extraction*, taxonomy can be mined based on the foundation from

Planning and expressing other elements based on research requirements. With an experimental apparatus there could be ancillary information based on required communication protocols, safety requirements, and physical apparatus behaviors. With *Design and Construction*, the complexity of the experimental apparatus taxonomy becomes more clear. While many of these processes fall into basic categories of Harvesting, Computation, and Control functionality, these aspects can be required to integrate across a series of multiple collaborating or competing hardware systems. It is critical to understand how the taxonomy maps across each of these components. *Validation* can occur at many stages throughout the development and can help identify assets and issues by demonstrating the taxonomy and utilization through subject-matter experts. Techniques for the validation of a taxonomy are highlighted in Šmite, Wohlin, Galviņa, and Prikladnicki [6] and include checking orthogonality of attributes, benchmarking against other taxonomies, and showing its utility, all of which improve the taxonomy and move towards wider adoption. Often this should lead to further iterations of the taxonomy, thereby strengthening it and also increasing the likelihood of further adoption and utilization. Despite the complexity, by establishing a thorough taxonomy, there is less room for failure to meet requirements that would cost time in additional iterative development cycles.

2 Opportunity: Towards a Scientific Software Engineering Taxonomy

The idea of grouping the knowledge areas of SE has existed since at least the 1990s [3] with many taxonomies proposed since. Recently, an open source scientific software specific taxonomy was started through GitHub [7]. While a start, it is in an embryonic phase, needing more specificity, a classification method, and to follow best practices in general. A set of these best practices was recently developed based on a systematic mapping study [8]. It is thus feasible to craft taxonomies specific to types of scientific software. We briefly discuss desirable taxonomy properties that can be readily applied.

Many taxonomies developed for SE suffer from issues that limit their usefulness, including not explicitly describing how to classify subject matter instances and the majority being qualitative [8]. In particular, a faceted classification structure is recommended for evolving fields of research and allows for the greatest flexibility of integrating new knowledge [5]. Advances in the nascent field of software metrology [1] will allow quantitative schemes to be developed. Such quantitative methods admit a mathematical description of subject matter instances as points in a space whose dimensions are the descriptive bases of classification [10]. Modern unsupervised machine learning techniques may be able to find hidden patterns and define latent classes.

3 Time is Now: Taxonomies as the Foundation of Research

We believe that an evidence-based approach to scientific software is necessary to ensure research is high-quality, valid, productive, and makes an impact. A fundamental first step in maturing scientific software engineering is the organization and classification of the types of software and associated subject matter into validated taxonomies. These results could support and be adopted into some of the ISO/IEC standards (e.g., ISO/IEC 15504-1). With software rapidly permeating through scientific fields, the moment is now to organize our knowledge.

References

[1] Alain Abran. *Software metrics and software metrology*. John Wiley & Sons, 2010.

[2] Marc Andreessen. Why software is eating the world. *Wall Street Journal*, August 20, 2011.

[3] Pierre Bourque, Robert Dupuis, Alain Abran, James W Moore, and Leonard Tripp. The guide to the software engineering body of knowledge. *IEEE software*, 16(6):35–44, 1999.

[4] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009.

[5] Barbara H Kwasnik. The role of classification in knowledge representation and discovery. *Library Trends*, 48(1):22–47, 1999.

[6] Darja Šmite, Claes Wohlin, Zane Galviņa, and Rafael Prikladnicki. An empirically based terminology and taxonomy for global software engineering. *Empirical Software Engineering*, 19(1):105–153, 2014.

[7] Vanessa Sochat. rseng/rseng: Research software encyclopedia criteria and taxonomy v0.0.18. https://github.com/rseng/rseng/.

[8] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43–59, 2017.

[9] Sira Vegas, Natalia Juristo, and Victor R. Basili. Maturing software engineering knowledge through classifications: A case study on unit testing techniques. *IEEE Transactions on Software Engineering*, 35(4):551–565, 2009.

[10] George R Wheaton. Development of a taxonomy of human performance: A review of classificatory systems relating to tasks and performance. Technical report, American Institutes for Research, Washington DC, 1968.