The Many Faces of the Productivity Challenge in Scientific Software

Hal Finkel (ANL/LCF)

2020-06-28

What we know: It takes far longer than we would like to develop, tune, and debug scientific software. We also know that maintenance is frustratingly time consuming and workflow integration seems more difficult and more fragile than it should. All of these aspects of our current development experience affect our overall productivity. All of these aspects present challenges and opportunities as we look toward the future. In this white paper, I'll attempt to outline a taxonomy underlying the overall productivity challenge in scientific software and suggest some specific opportunities for future work. This taxonomy roughly follows the software life cycle, but real software life cycles involve aspects of all of these items at many points in time.

Concept → Design (Planning)

Before a piece of software is written, some planning should be done. Can the productivity of the planning process be improved? One very-important aspect of the planning process is *searching for information on how similar problems were solved in the past.* In the context of scientific software, this knowledge is generally found in two places: First, the scientific literature contains papers, dissertations, and other written descriptions of past work. Second, repositories hold the source code of past projects. Have you ever tried to search for previous work relevant to a particular mathematical equation? It's nearly impossible. Unfortunately, current search technology for papers and code is both mathematics and code-structure unaware, treating everything as a glorified stream of works and links, making searching a very iterative and very manual process. For the purpose of answering queries, little relationship exists between code and its description in the literature. As machine-learning-based search technologies and relevant database technologies become more sophisticated and reliable, significant opportunities exist to improve this process:

- *Recommendation:* We need better ways to search both scientific literature and scientific source code. Systems that understand the correspondence between source code and its description in the literature would be helpful. Similarly, systems that understand code semantics and structure[1][2], understand how comments in the code relate to that structure, and allow for structure-based searching, would be helpful. Code and literature searching that does a better job of understanding mathematical equations and notation would be helpful[3].

After understanding, to the extent practical, our past collective experience, a number of options will likely remain. These options might include different potential numerical discretization schemes, data structures, and decomposition techniques. How can these choices be made? In part, these decisions are made by constructing prototypes. Prototyping is interesting in this regard because the construction of the prototype is a microcosm of the entire development process, but with less stringent requirements in some respects (e.g. performance, scale, portability, maintainability). As a result, no special consideration will be given to prototyping itself except to note that the most productive programming environments for prototyping might be different than the ones for production development.

Part of the planning process includes an assessment of the capabilities needed for the software. Perhaps a particular kind of non-linear solver is needed, or perhaps we will require a large class of solvers to be available for runtime selection. Perhaps FFTs are needed, or BLAS operations, or integration with a particular API. A number of questions need to be answered with respect to these capabilities: What capabilities are needed? What libraries and/or programming environments can provide those capabilities? How difficult will it be to acquire, learn, and adopt those components? Can the transitive closure of the dependency graph be understood? Are there particular risks associated with any of these dependencies? Have the components, and component combinations, been tested at the requisite scale? Will the components support the required hardware platforms? Especially if the software being planned will be open source, are there any licensing concerns with those dependencies? Some of these questions are currently answerable by looking at metadata from package managers (e.g., Spack[4]), but especially for data on future development, these questions can be answered only by a manual process of searching documentation, mailing lists, forums, and similar, or communicating directly with developers.

- *Recommendation*: We need more-comprehensive software metadata capturing not only build dependencies and licensing, but information on testing, development practices, anticipated funding stability, and planned hardware-support timelines. Having this metadata integrated with the package manager such that it can be collected for transitive dependencies would be helpful. Being able to find metadata describing at-scale testing of particular component combinations, and particular configurations, would be helpful.

Design → Implementation (Authorship)

Translating the planned design into an actual implementation accounts for a significant portion of the overall software-development effort. This process is often complicated by the fact that the requirements and/or plan might change during the implementation process. Nevertheless, the productivity of this process is impacted by the ability to maximize code reuse and minimize required boilerplate, having good documentation on the libraries and programming environment in use, using tools with which the developers are experienced, and so on. As program synthesis (a.k.a. machine programming) methods mature, intelligent programming assistants can play an important role in increasing programmer productivity by generating code based on input specifications (either in natural language or mathematics), examples of correct behavior, and provided solution templates. Prototypes created during the design process can form an important potential input source for synthesis systems. In other domains, we're already seeing ML-driven code-completion editor plugins that integrate library documentation, code context, and so on to provide useful suggestions (e.g., Kite[5], TabNine[6]). As far as synthesis goes, this is still rudimentary, but even these systems do not yet exist for C++ or Fortran to have the same impact on scientific code development. Domain-specific languages (e.g., Halide[7], TACO[8], Unified Form Language[9] as used by FEniCS[10]/PyOP2[11]) can play an important role for productive programming in certain kinds of application domains.

- *Recommendation*: We need more-intelligent programming tools to assist with the generation of code[12][13] for scientific applications. In addition, we need to work on driving down the amount of boilerplate necessary to implement common scientific-programming tasks. This can be due to better library design, improvements to C++ or other relevant general-purpose programming languages, or the use of domain-specific languages. It is, as in nearly every domain, critical that libraries and programming environments have good documentation with tutorials and examples.

One important aspect of the programmer-productivity challenge is compilation/iteration time. If it takes many minutes, or even hours, to rebuild an application in order to test a change, that is a problem. On top of that, if making sure the change is sensible requires execution via a queuing system with a non-trivial waiting time (e.g., queuing a job on a machine that runs many MPI ranks or runs with particular accelerator hardware), this can also significantly decrease developer productivity. These challenges can, in part, be addressed by a more-rigorous code-testing regimen (e.g., unit tests, mocks, stand-alone component drivers), but the creation and maintenance of the necessary tests and infrastructure are currently very time consuming.

- *Recommendation*: We need improved software analysis and compilation turn-around time in order to increase developer productivity. In addition, we need better tools to help programmers of scientific software create and maintain tests, not only to ensure correctness, but also to improve basic development iteration times on large projects. This might include specialized test/mock capabilities representing hardware accelerators, networking configurations, etc. that allow for local emulation of capabilities normally available only on shared systems.

Implementation → Correct Implementation (Debugging)

An implementation cannot be considered complete until it has been shown to be correct. This includes both verification and validation, and critically, both processes are difficult. Tracking down places where insufficient numerical precision is causing problems[14], or even where run-of-the-mill programming errors are causing crashes or inconsistencies, is frustratingly difficult. Instrumentation-based debugging tools have been developed for many of these classes of errors (e.g., LLVM Sanitizers[15][16]), but often these are difficult to use at large scale. With the integration of machine-learning-based technologies, we're seeing increasing use of data-driven techniques in scientific software development. This brings its own set of opportunities and challenges in all aspects of V&V. Nevertheless, even in cases where the correct numerical output is known, tracking down the cause of a bug is often hard. A move toward programming systems that enforce more constraints statically, thereby reducing the prevalence of bugs in well-formed programs, may be helpful, assuming such systems are sufficiently easy to learn and adapt.

- *Recommendation*: Debugging tools, especially high-performance, instrumentation-based tools, should be enhanced to work at large scale in HPC environments, and on large code bases. Similarly, static-analysis tools need improvements to work on large code bases in C++, Fortran, and other relevant programming languages. Debugging methodologies need to be enhanced to incorporate ML-based components, and intelligent, data-driven debugging of procedural code should be enabled by a new class of AI-driven technologies.

Implementation → High-Performance Implementation (Tuning)

The transformation of a working piece of software to increase its runtime performance is often time consuming and requires specialized skills. Tools already exist to help with application profiling, both from vendors (e.g., VTune) and from the HPC community (e.g., HPCToolkit[17], TAU[18]). However, these tools only help with a critical, but small, slice of the overall problem. First, modeling is needed to understand how well a particular algorithm on a particular hardware architecture might perform. Second, the profiling tools can help compare that performance to performance actually observed (and, in addition, help pinpoint potential performance problems such as load imbalance or cache misses). Third, the programmer needs to understand the cause of the inefficiencies observed, and to the extent that these inefficiencies can be characterized as abstraction penalties, whether these can be mitigated without breaking through otherwise-helpful layers of abstraction. The developer then needs to implement any necessary changes, test those changes, and so on. The developer must do all of this while being mindful of the risk of unintentionally creating larger problems (e.g., for the long-term maintenance of the software) than the performance problem being solved. Of course, to the extent that the programming environments (i.e., compilers, runtime libraries, and so on) can produce high-performance code, this overall problem can be mitigated. Compilers, for example, require specific capabilities to generate high-performance code even in the face of complex abstraction layers. Scalable link-time-optimization (a.k.a. whole-program optimization) helps with this[19], as does scalable profiling and profile-guided optimizations. Compilers cannot evaluate perfect hardware models when making code-generation decisions, as that would be impractically expensive, and often lack knowledge of dynamic factors (e.g., problem sizes) that have significant performance impacts and are unavailable until program execution. Auto-tuning has proven successful in addressing these kind of challenges[20], but remains largely unused, and for some cases, just-in-time compilation has proven very effective, although that is also little used in HPC (exceptions include in Julia[21], OCCA[22], libxsmm[23], and some research work, e.g., ClangJIT[24]). It is noted that tuning sometimes involves other factors than execution speed, for example, power usage, storage usage, or memory footprint.

- *Recommendation*: Better tools need to be developed to help developers model ideal performance and scaling on platforms of interest such that these models can be compared to observed performance. Compilers need to be improved to generate better code even in the face of complex abstraction layers, sometimes requiring language enhancements so that the compiler can get needed information from the library/application developer. Programming and execution environments need to support scalable profiling and link-time optimizations. Additionally, programming environments need to be enhanced to provide intelligent performance-relevant feedback to developers and easily support autotuning to improve on what internal models can provide. Just-in-time compilation should be provided to adapt to different inputs and solution regimes in order to increase application performance.

Implementation on One Architecture → Implementation on Another Architecture (Portability)

Portability of scientific software is a particular concern, given the large, diverse use base of many scientific applications and the diverse set of hardware available to those users. Portability is accomplished via abstraction, either using high-level/domain-specific languages or abstraction libraries (e.g., Kokkos[25]/RAJA[26]). Compilers and runtime libraries must be designed with these abstractions in mind, but critically, effective abstractions must exist in the first place. Fortunately, significant progress has been made on this front in recent years, with C++ libraries such as Kokkos, programming models such as OpenMP[27] and SYCL[28], languages such as Fortran and Julia, and systems such as Halide. These systems have developed methods for expressing data-parallel algorithms (and, to a lesser extent, task-parallel algorithms) in a portable fashion across modern CPUs and GPUs. With the increasing prevalence of hardware optimized for machine learning, and the general increase in in-system heterogeneity, these portability mechanisms will need to continue to evolve. It seems likely that the level of expression will need to evolve toward being higher level than it is commonly today. It is important to note that evolution is not just occurring in compute technology, but also in memory and longer-term storage as well.

- *Recommendation*: Portability-focused abstraction layers and programming models will need to continue to be developed, and moreover, evolve to target, not only CPUs and GPUs, but also AI-focused accelerator hardware and other aspects of increasingly-heterogeneous systems.

Isolated Implementation → Implementation As Part of a Larger Project (Integration)

Integration of scientific software into a larger workflow can be challenging for several reasons. One, the integration (e.g., between two different physical models) often needs to be fine grained, and as a result, the integration method has a significant affect on performance. Techniques used in other contexts (e.g., the microservices model commonly used in web-application development) are ruled out by performance-related design constraints. Scientific applications, however, often end up needing to integrate components programmed in different languages (e.g., C++, Fortran, and Python). The lack of standardized interchange formats also hinders the ability to integrate different applications, and thus developer productivity. In part, this is a natural consequence of the fact that scientific software sits on the edge of what is known, and so bespoke data formats are created, as nothing quite like what is being done has been done before. However, this proliferation of data formats and APIs means that an unfortunate amount of developer time is spent on writing glue code and data-processing code.

- *Recommendation*: Work needs to continue on tools that generate high-performance interfaces between components written in a variety of relevant programming languages. Similarly, tools that generate code for, or otherwise handle, data in a variety of customizable formats need continued development to increase programmer productivity.

Integration of scientific software can also provide difficult when the different components lack an ability to coordinate their use of shared resources. This commonly occurs when managing hardware threads on CPUs and accelerators, but also occurs with more mundane resources such as system memory and file handles. For example, if part of an application uses OpenMP to manage threads, and another part uses Kokkos, and one is called from within the other, will the composite system make sensible use of the hardware? Maybe not.

- *Recommendation*: Runtime systems, and operating systems, need enhancements to ensure that application components can communicate to collectively manage system resources. Only if components are composable can programmers choose the most-productive implementation technologies for each part of a larger project and/or reuse components created by others.

Implementation → Reproducible Results (Provenance)

While important everywhere to some extent, it is critical to the scientific process that the results of running scientific applications are reproducible (at least for some period of time), and moreover, that output data from the application can be associated with a particular application configuration and its transitive chain of dependencies. In current systems, however, actually accomplishing this task of recording high-quality provenance information is very hard. Code versioning helps, static linking and library-version tracking can help, techniques such as storing application-configuration information with all outputs can help, but creating a high-quality system in this regard is difficult and time consuming. Moreover, the lack of high-quality provenance information ends up harming developer productivity, in part because it leads to more-difficult debugging tasks. This really shouldn't be hard, but we need provenance concerns to permeate the entire stack in a way that they currently do not.

- *Recommendation*: Programming environments need to make it easy to collect, statically and dynamically, dependency, configuration, and version information for all applications. Application frameworks need to have common ways to extract relevant configuration values so that they can be stored with output data and restored as required.

Implementation → Updated Implementation (Maintenance)

A lot of developer time is sunk into maintenance tasks: Updating code to work with newer versions of programming environments and libraries, comply with updated coding guidelines, handle new usage scenarios, and so on. Often, these processes, even if they could be automated in theory, are done manually, because insufficient automation tools exist in practice. Sometimes, the tools exist, but are so unproductive to use that it's faster to do the work manually (e.g., creating a clang-tidy[29] plugin is powerful but is non-trivial and requires specialized skills). In some sense, these problems are not specific to scientific programming, but the necessary tooling needs to productively exist for the languages and libraries used for developing scientific applications.

- *Recommendation*: Better tooling needs to be developed in order to handle language upgrades and perform API migration. These tools might benefit from the use of ML technology. It is important that the tools are easy to use, likely example driven.

Initial Implementation Developers → Replacement/Additional Implementation Developers (Training)

An important consideration for all aspects of the productivity challenge for scientific applications is the reality that training is a challenge. A tool, language, etc. can be very productive for a specialist, but if it takes someone new to the team many years to learn how to use the tool effectively, the continued staffing of the development team will be difficult.

- *Recommendation*: It is important that all capabilities created to address productivity challenges place an emphasis on being easy to learn (or, at least, it should be easy to get started).

Implementation → The Next Implementation (Knowledge Transfer)

No particular implementation will remain useful for ever. Eventually, new requirements will overtake the ability of the software to be adapted by mutation, and a new implementation must be created. This process happens for small components and also for large code bases. The productivity challenge here is to prepare: Not only must the code itself be well documented; the rationale for the current design must be well documented (and current, as the design has likely changed over time).

- *Recommendation*: Systems must be developed, perhaps making use of ML technology, to make it easy to keep both detailed and high-level documentation in sync with the code and complete[30]. This includes dealing with mathematical descriptions that commonly occur in scientific applications.

1 Mining Framework Usage Graphs from App Corpora, Sergio Mover, Sriram Sankaranarayanan, Rhys Braginton Pettee Olsen, Bor-Yuh Evan Chang, IEEE International Conference on Software Analysis, Evolution and Reengineering, 2018 : https://github.com/cuplv/biggroum
2 Kashyap, Vineeth, David Bingham Brown, Ben Liblit, David Melski, and Thomas Reps. "Source forager: a search engine for similar source code." arXiv preprint arXiv:1706.02769 (2017).
3 Pineau, Deanna C. "Math-Aware Search Engines: Physics Applications and Overview." arXiv preprint arXiv:1609.03457 (2016).
4 Gamblin, Todd, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. "The Spack package manager: bringing order to HPC software chaos." In SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12. IEEE, 2015.: https://spack.io/
5 Kite: https://kite.com/
6 TabNine: https://www.tabnine.com/
7 Ragan-Kelley, Jonathan, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." Acm Sigplan Notices 48, no. 6 (2013): 519-530.: https://halide-lang.org/
8 Kjolstad, Fredrik, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The tensor algebra compiler." Proceedings of the ACM on Programming Languages 1, no. OOPSLA (2017): 1-29.: http://tensor-compiler.org/
9 Alnæs, Martin S., Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. "Unified form language: A domain-specific language for weak formulations of partial differential equations." ACM Transactions on Mathematical Software (TOMS) 40, no. 2 (2014): 1-37.
10 Alnæs, Martin, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. "The FEniCS project version 1.5." Archive of Numerical Software 3, no. 100 (2015).: https://fenicsproject.org/
11 Rathgeber, Florian, Graham R. Markall, Lawrence Mitchell, Nicolas Loriant, David A. Ham, Carlo Bertolli, and Paul HJ Kelly. "PyOP2: A high-level framework for performance-portable simulations on unstructured meshes." In 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1116-1123. IEEE, 2012.: https://github.com/OP2/PyOP2
12 Murali, Vijayaraghavan, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. "Neural sketch learning for conditional program generation." arXiv preprint arXiv:1703.05698 (2017).
13 Component-Based Synthesis for Complex APIs. Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, Thomas W. Reps. POPL 2017.: https://github.com/utopia-group/sypet
14 Rubio-González, Cindy, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. "Floating-point precision tuning using blame analysis." In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 1074-1085. IEEE, 2016.
15 Serebryany, Konstantin, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker." In Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), pp. 309-318. 2012.: https://clang.llvm.org/docs/AddressSanitizer.html
16 Serebryany, Konstantin, and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice." In Proceedings of the workshop on binary instrumentation and applications, pp. 62-71. 2009.: https://clang.llvm.org/docs/ThreadSanitizer.html
17 Adhianto, Laksono, 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, no. 6 (2010): 685-701.: http://hpctoolkit.org/
18 Shende, Sameer S., and Allen D. Malony. "The TAU parallel performance system." The International Journal of High Performance Computing Applications 20, no. 2 (2006): 287-311.: http://tau.uoregon.edu/
19 Johnson, Teresa, Mehdi Amini, and Xinliang David Li. "ThinLTO: scalable and incremental LTO." In 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 111-121. IEEE, 2017.: https://clang.llvm.org/docs/ThinLTO.html
20 Tiwari, Ananta, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. "A scalable auto-tuning framework for compiler optimization." In 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1-12. IEEE, 2009.
21 Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A fresh approach to numerical computing." SIAM review 59, no. 1 (2017): 65-98.: https://julialang.org/
22 Medina, David S., Amik St-Cyr, and Tim Warburton. "OCCA: A unified approach to multi-threading languages." arXiv preprint arXiv:1403.0968 (2014).: https://libocca.org/

23 Heinecke, Alexander, Greg Henry, Maxwell Hutchinson, and Hans Pabst. "LIBXSMM: accelerating small matrix multiplications by runtime code generation." In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 981-991. IEEE, 2016.: https://github.com/hfp/libxsmm
24 Finkel, Hal, David Poliakoff, Jean-Sylvain Camier, and David F. Richards. "Clangjit: Enhancing c++ with just-in-time compilation." In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 82-95. IEEE, 2019.: https://github.com/hfinkel/llvm-project-cxxjit/wiki
25 Edwards, H. Carter, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." Journal of Parallel and Distributed Computing 74, no. 12 (2014): 3202-3216.: https://github.com/kokkos/kokkos
26 Beckingsale, David, Richard Hornung, Tom Scogland, and Arturo Vargas. "Performance portable C++ programming with RAJA." In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 455-456. 2019.: https://github.com/LLNL/RAJA
27 Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering 5, no. 1 (1998): 46-55.: https://www.openmp.org/
28 SYCL: https://www.khronos.org/sycl/
29 clang-tidy: https://clang.llvm.org/extra/clang-tidy/
30 Wong, Edmund, Jinqiu Yang, and Lin Tan. "Autocomment: Mining question and answer sites for automatic comment generation." In 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 562-567. IEEE, 2013.