Towards Richer Challenge Problems for Scientific Computing Correctness

Matthew Sottile Mohit Tekriwal John Sarracino

Lawrence Livermore National Laboratory, USA

{sottile2, tekriwal1, sarracino2}@llnl.gov

Correctness in scientific computing (SC) is gaining increasing attention in the formal methods (FM) and programming languages (PL) community. Existing PL/FM verification techniques struggle with the complexities of realistic SC applications. Part of the problem is a lack of a common understanding between the SC and PL/FM communities of machine-verifiable correctness challenges and dimensions of correctness in SC applications.

To address this gap, we call for specialized challenge problems to inform the development and evaluation of FM/PL verification techniques for correctness in SC. These specialized challenges are intended to augment existing problems studied by FM/PL researchers for general programs to ensure the needs of SC applications can be met. We propose several dimensions of correctness relevant to scientific computing, and discuss some guidelines and criteria for designing challenge problems to evaluate correctness in scientific computing.

1 Introduction

Scientific computing and high performance computing has long relied upon benchmark suites to give computer science researchers target challenge problems that are known to be relevant to end-users in the sciences. These have most often been successful in the area of performance benchmarks that have driven compiler, systems, and parallel computing research. For example, the NAS Parallel Benchmarks (NPB) [11, 10] were created to be representative of highly parallel HPC problems that arise in aerospace applications, specifically in fluid dynamics simulations. The Mantevo problems [13] (and their predecessors such as the Salishan problems [18]) provide similar problem collection representative of those of interest to the broad HPC community within the US Department of Energy. A related effort in the area of programming languages and compilers is the Shonan challenge [24] proposed in the early 2010s.

The target challenge problems shared by the HPC community have grown in sophistication over time: early targets were as simple as Linpack [16] and related linear algebra and numerical primitives, leading later to small algorithmic benchmarks, up to the current day suites that represent larger kernels and full miniature applications. We expect that similar growth in sophistication will also be necessary for correctness challenges, where we start small but seek to create challenges that get ever closer to full applications as our ability to reason about challenge problems improves.

A key focus when constructing the performance-oriented benchmark suites was to adequately cover the set of known computer and software design dimensions that a system exhibits that influence performance. In high performance computing platforms this includes fine-grained instruction performance, cache and memory hierarchy effects, vector or SIMD instructions, instruction-level parallelism features within CPUs, communication layers, and so on. Without care, one could build benchmarks that are redundant with respect to what they measure. When considering memory (or more broadly, communication) for example, focusing on dense problems (e.g., dense linear algebra, regular stencil solvers, etc.)

Submitted to:
VSS 2025

would miss the very different properties of sparse problems. Similarly, both dense and sparse linear algebra problems may exhibit a high degree of regularity of memory access patterns that do not resemble less structured patterns present in some problems (e.g., unstructured mesh data structures), or even random-like patterns that arise in others (e.g., graph problems). Similar issues arise when looking at other aspects of a system: instruction mix, control flow patterns, different forms of concurrency or parallelism, etc.

In the performance of scientific computing applications, the dimensions that describe different applications are relatively well understood and have motivated most of the benchmark suites we see today. The open question that we are interested in here is focused not on performance but correctness: *what suite of challenge problems exhibits the diversity of algorithmic, data structure, and computational models that are found in real-world scientific computing workflows*?

Existing challenge problems in the formal verification community are a good starting point. For example, many of the programs in the Gallery of Verified Programs [19, 2] are drawn from challenges like VerifyThis [23], the NSV-3 benchmarks [8, 17], and various verification competitions. These have gaps though: graph challenges typically focus on traversals, whereas in scientific codes graphs often require algorithms that aren't easily expressible as traversals (e.g., partitioning and coloring). Matrix algorithms are typically dense, and sparse problems often overlook the differences that arise in different representation suitable for different target architectures. In linear algebra, we are missing important operations like matrix factorization methods, eigensystem calculations, and linear solvers. While the numerical verification community has developed ground-breaking verified numerical algorithms for decades (such as coq-num-analysis [6] and the Mathematical Components Library [4]), these individual research efforts have not yet been condensed into a general set of grand challenges relevant to the scientific computing community. Building from the existing formal verification challenges that set a good foundation would be extremely useful to pull formal verification techniques closer to the kinds of algorithms and programs that one encounters in practice in modern scientific computing.

1.1 Challenge problems vs benchmarks

There is a distinction between challenge problems and benchmarks. Challenge problems define a target to reach for, while benchmarks provide an objective set of metrics that assess the relative quality between different solutions. Current scientific computing benchmark suites have relatively narrow scopes of correctness: floating point benchmark suites [14] focus on numerical issues, while suites such as DataRaceBench [25] focus on specific classes of concurrency control problems. These are not complete because they focus on low-level properties far from the scientific domain and are in and of themselves not specific to scientific computing. Floating point correctness and data race freedom are generic problems that span computing domains. It is unsurprising that current suites have this form, as programming language and formal methods experts generally lack context in domains of scientific computing. As a consequence the suites represent the topic areas where scientific computing domain knowledge is not necessary to make positive progress. Overall, there is not yet sufficient coverage of correctness criteria in scientific computing to form a comprehensive benchmark suite.

1.2 Contributions

In this position paper, we advocate for a set of *correctness* challenge problems for scientific computing. We develop two distinct categories of insights: 1) a set of *Dimensions of Correctness* that correctness challenge problems should encompass in Section 2; and 2) a set of *Pitfalls* that correctness challenge problems should *Avoid* to ensure interest and relevancy to scientific computing practitioners in Section 3.

Our recommendations are meant as a starting point to a broader discussion between the scientific computing and formal methods communities.

2 Dimensions of correctness

What are correctness dimensions that are analogous to the performance dimensions classically used to select components of benchmark suites? Scientific programs straddle different levels of the computer abstraction hierarchy: at the lowest level we have instances of numerical calculations and data in memory, while at higher level we encounter structures that connect to the models that represent the physical systems and solution methods for their governing equations. It is important to not focus solely on dimensions that are most familiar and easy to verify, such as low-level features like numerical operations, memory management, and concurrency/parallelism. Those considerations, while important for scientific programs, are arguably already well covered in existing general purpose benchmark suites (e.g., FPbench [14], DataRaceBench [25], SPEC [15], etc.). The most impact on scientific computing will be to fill the gaps between those low level aspects of a system and the high-level applications that are built atop them. At the end of the day, one can still write a bad scientific application that correctly implements floating point, avoids memory safety issues, and is free of concurrency control errors. We propose the following metrics to capture correctness criteria, which includes both low-level and high-level application correctness.

Numerics Scientific computing is fundamentally about numerical calculations and correctness often is defined as a correct correspondence between mathematical operations and their implementation in hardware or software. This is most apparent when considering floating point operations that are key to nearly all scientific applications. An additional challenge is the advent of *reduced precision* floating-point formats. With increasing hardware support for reduced precision, realistic benchmarks must necessarily incorporate reduced precision and mixed precision algorithms.

Data structures Scientific codes typically must manage a great deal of data in support of the calculations they perform. While many standard data structures are heavily used (e.g., containers from the C++ Standard Template Library), it is not uncommon for applications to implement modified or hybrid data structures to allow for finer grained performance tuning. For example, a program may “transpose” a data structure (such as a struct-of-arrays (SOA) to array-of-structs (AOS)) during different phases of the code to optimize for different memory traversal patterns. Correctness in that case not only means that each instance of the structure (SOA, AOS) is correct, but the two structures are semantically identical modulo physical layout and performance changes. Similarly, priority queues, heaps, and other structures also form key parts of codes such as discrete event simulations. These data structures typically have well understood correctness properties (e.g., invariants, memory safety, etc.).

Domain-modeling structures A specialized class of data structure that is worth calling out independently are those used for domain modeling. Often these resemble generic data structures, but carry with them additional constraints on correctness that derive from the physical system that they are modeling. These range from grids and meshes that represent a discretized view of the subject of the model (e.g., an airplane wing), to data structures that support efficient calculations (e.g., a space partitioning data structure used for collision detection). These data structures range from the simple (regular n-dimensional arrays) to very complex (unstructured grids, graphs, trees, and sometimes a hybrid of all three). Correctness of these data structures not only include generic properties (e.g., memory safety, invariants, structural assertions) but also must include high-level application domain invariants (e.g., conservation laws).

Differential equations: Scientific computing applications mostly concern with modeling physics using partial differential equations (PDEs) and then solving those PDEs approximately. For a numerical solver to emit a correct physical solution, consistency of the PDEs with respect to the physics being modeled must be formalized. Thus, a realistic benchmark should contain consistent formal approaches or abstractions to model these PDEs and associated correctness criteria such as well-posedness, boundary condition compatibility, numerical stability, etc. Such correctness metrics are well-studied in the numerical methods communities and must necessarily be part of the correctness of a scientific computing challenge problem.

Concurrency and parallelism Scientific computing is rarely a single CPU, single computer activity - it is the home of some of the most complex parallel programs around. Reasoning about correctness of parallel applications is challenging because often scientific applications combine different parallel programming and concurrency control models in a single application. Reasoning about correctness requires considering a combination of shared memory, vector, and distributed memory parallelism: often implemented by different libraries that are not necessarily aware of each other. Even worse, in high performance scientific computing we are not just concerned with correctness, but with performance - so it is necessary to reason about potentially complex algorithms that attempt to minimize synchronization in the interest of performance (at the cost of analytical complexity).

Approximation schemes Scientific programs implement models of phenomena of interest that often rely upon some approximation methods. These are chosen for a variety of reasons: incomplete understanding of the system of interest, infeasibility of running a full fidelity model, or limited availability of necessary computational resources. Challenge problems that allow us to reason about approximations are important to allow people to reason about the validity or quality of approximation methods. It is important to consider approximation schemes separately from numerical methods, which are a very specific class of approximation that is driven by our implementation of numbers in computers. Approximation schemes often can be reasoned about using their definition over the mathematical reals without needing to complicate matters with numerical precision considerations when considering challenge problems. Ultimately that wholistic view is necessary, but at the full application level.

3 Pitfalls to avoid

We next focus on the gap between traditional software verification correctness challenge problems and their relevancy to realistic scientific computing applications. When considering performance benchmarks, one goal often was diversity of implementation: a good benchmark suite allowed for implementation in multiple languages and environments to ensure that they were representative of the community. A benchmark tuned to only be suitable for a single system or language often is less useful to the community. For example, a challenge problem specifically designed to test the ability of a functional language compiler to desugar a high-level syntax to a lower level form is not useful to a language developer working with a classical imperative language - thus it is not a generalizable challenge problem.

Avoid Over-specialization In the correctness domain it is useful to ensure that correctness challenges also avoid over-specialization to specific techniques or systems used to reason about correctness. A problem that is highly specialized to SMT solvers or dependently typed proof assistants is better suited for those communities than it is to the broader scientific computing community. Such problems are more suitable to challenges such as those studied as part of the SAT competition [7, 26] or static analysis test suites such as Juliet [3]. We should instead focus on creating challenge problems that are sufficiently high level that they can be mapped onto different methods for reasoning about them. It may be the case that some problems map poorly onto some methods, but that is perfectly reasonable for challenge problems as it provides something for practitioners to aim at.

Avoid "Toy" Problems It is also important to consider benchmarks that are realistic. Many running examples of "scientific applications" are not well aligned with the current state of the art. Either they represent extremely dated methods that are now considered defunct, originate from introductory textbooks with the intention of teaching foundations, or they come from contexts outside the practice of domain sciences where efficiency, accuracy, and precision are not critical. For example, a common running example is a direct $O(n^2)$ solver for an n-body model, but in practice more efficient methods based on space partitioning trees (e.g., the Barnes-Hut method [12]) or multipole methods [21] are used. These methods bear little similarity to the simpler direct method.

Similarly, there are often cases where algorithms appear to be "scientific", but are actually intended for non-science purposes. A good example of this are the fluid models based on Stam's "Stable Fluids" work [28], which provides fluid-like visualizations for games and video purposes, but exhibit known bad properties (e.g., numerical dissipation) that make them scientifically less valuable. We must consider the validity and utility of any benchmark with respect to the domain that it intends to serve, which may require us to deprioritize algorithms that may be more appealing from a verification perspective due to their simplicity.

Focus on correctness issues unique to scientific applications Finally, good challenge problems for scientific computing focus on the issues unique to scientific computing. While broad application of challenge problems is useful, general purpose challenges often fail to focus on the correctness dimensions that address issues unique or of unusually high importance to scientific computing compared to other domains. For example, many domains outside of SC and HPC require bit-level correctness (e.g., cryptographic algorithms), or a very relaxed level of correctness where differences are imperceptible (e.g., visual effects). Scientific computing brings in richer requirements that often fall somewhere in between: preservation of a conservation law or non-violation of a bound on the rate by which information can propagate (e.g, the CFL condition). Integrating these kinds of physical correctness criteria into a benchmark suite will be very valuable in connecting what the benchmark evaluates to the constraints that are relevant to scientific applications.

Separate underlying mathematics from the implementation The fundamental underlying mathematics for various numerical approximation schemes remains the same irrespective of their implementation. For instance, the Gram-Schmidt algorithm fundamentally computes an orthogonal set of vectors given a linearly independent set of input vectors. The implementation of Gram-Schmidt algorithm has two well-known variant: classical and modified, where the latter has better numerical properties when implemented in finite precision. However, both these variants need to satisfy the mathematical invariant or a fundamental characteristic of Gram-Schmidt method, which is the orthogonality of the vectors. As the scientific libraries are ported and rewritten to cater to the performance benefits of evolving hardware, we need to be able to verify the key mathematical invariants of the numerical schemes with respect to their well-known classical counterparts. Challenge problems should identify numerical approximation schemes which are sensitive to the evolving software and hardware landscape in scientific computing, to aid the development of and evaluate formal tools which can verify key mathematical invariants irrespective of the low level implementations.

Allow unchecked assumptions Often programs state a set of assumptions they make about their inputs that are necessary for the program to function correctly. For example, a function may assume all inputs are positive, or a simulation may assume that an input triangular or tetrahedral mesh avoids angles below a certain threshold. Given that scientific computing programs often emphasize performance, it is not uncommon for code to make these assumptions but not waste compute time checking them. We may discover that challenge problems for scientific problems will carry more unchecked assumptions about their inputs than we may find in other domains simply because these checks impede performance or interfere with algorithms and data structures tuned to be performant.

4 Conclusion and final thoughts

Correctness fundamentally requires a clear definition of what constitutes "correct", but precisely what aspects of correctness matters varies. A program may be correct with respect to memory safety, yet absolutely incorrect with respect to a higher level physical model that it is intended to implement. A set of challenge problems specifically aimed at scientific computing should focus first and foremost on correctness at the scientific domain and the modeling techniques used to encode the domain problem in some computational form that can be implemented.

Unlike performance challenges, defining correctness constraints often requires contextual information that may not directly manifest in the implementation or its execution context. These may include known bounds imposed by the physical world (e.g., hard limits like the speed of light), dimensional analysis, units of measure, physical constants, and so on. Furthermore, there may be constraints best expressed in quantities that are derived from other properties (e.g., dimensionless measures describing fluid flow regimes): the relationship of these derived quantities to their components must be captured as part of a specification. Physical models often exploit or assume symmetries or conservation laws, and these laws must also be expressed as part of a specification as they are a fundamental assumption that the model rests upon.

In this position paper, we advocate for a generalizable set of *correctness* challenge problems for scientific computing applications. The intention is to help bridge the gap between existing formal verification techniques and practical scientific codes by providing a realistic set of compelling "grand challenges" for scientific correctness. Such a challenge set must span the computer abstraction boundaries used in scientific computing, incorporating both low-level correctness issues (such as memory and numerical safety) as well as high-level physical correctness criteria. Moreover, we discuss several pitfalls that a correctness challenge set should avoid. Our recommendations aim to start a dialogue between formal methods and scientific computing domain experts, motivating a common set of correctness challenges that will enable more practical and powerful formal verification techniques within the scientific computing community.

Acknowledgements

This manuscript has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA2 7344 with the US. 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 non-exclusive, 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.

References

[1] *Archive of Formal Proofs*. https://www.isa-afp.org/.

[2] *Gallery of verified programs — toccata.gitlabpages.inria.fr*. https://toccata.gitlabpages.inria.fr/toccata/gallery/.

[3] *Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset — samate.nist.gov*. https://samate.nist.gov/SARD/test-suites/112.

[4] *Mathematical Components*. https://math-comp.github.io/.

[5] *Mathematics in mathlib*. https://leanprover-community.github.io/mathlib-overview.html.

[6] *Micaela Mayero / Numerical Analysis in Coq · GitLab*. https://depot.lipn.univ-paris13.fr/mayero/coq-num-analysis/.

[7] *SAT Competitions — satcompetition.github.io*. https://satcompetition.github.io/.

[8] (2010): *NSV-3: Third International Workshop on Numerical Software Verification.* https://www.lix.polytechnique.fr/Labo/Sylvie.Putot/NSV3/.

[9] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo & Maurice Yarrow (1995): *The NAS parallel benchmarks 2.0.* Technical Report, Technical Report NAS-95-020, NASA Ames Research Center.

[10] David H. Bailey (2011): *NAS Parallel Benchmarks*, pp. 1254–1259. Springer US, Boston, MA, doi:10.1007/978-0-387-09766-4_133. Available at https://doi.org/10.1007/978-0-387-09766-4_133.

[11] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber et al. (1991): *The NAS parallel benchmarks—summary and preliminary results.* In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165.

[12] Josh Barnes & Piet Hut (1986): *A hierarchical O (N log N) force-calculation algorithm.* nature 324(6096), pp. 446–449.

[13] Paul Stewart Crozier, Heidi K Thornquist, Robert W Numrich, Alan B Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M Willenbring, Douglas W Doerfler & Michael Allen Heroux (2009): *Improving performance via mini-applications.* Technical Report, Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ....

[14] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern & Zachary Tatlock (2016): *Toward a standard benchmark format and suite for floating-point analysis.* In: *International Workshop on Numerical Software Verification*, Springer, pp. 63–77.

[15] Kaivalya M Dixit (1991): *The SPEC benchmarks. Parallel computing* 17(10-11), pp. 1195–1209.

[16] Jack J Dongarra, Piotr Luszczek & Antoine Petitet (2003): *The LINPACK benchmark: past, present and future. Concurrency and Computation: practice and experience* 15(9), pp. 803–820.

[17] Georgios Fainekos, Eric Goubault, Sylvie Putot & Stefan Ratschan (2011): *Mathematics in Computer Science: Numerical Software Verification*. 5(4), Springer International Publishing. Available at https://link.springer.com/journal/11786/volumes-and-issues/5-4.

[18] John T. Feo (1992): *Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Inc., USA.

[19] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 — Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 125–128.

[20] Maya Gokhale, Ganesh Gopalakrishnan, Jackson Mayo, Santosh Nagarakatte, Cindy Rubio-González & Stephen F Siegel (2023): *Report of the DOE/NSF workshop on correctness in scientific computing, june 2023, orlando, FL*. *arXiv preprint arXiv:2312.15640*.

[21] Leslie Greengard (1988): *The rapid evaluation of potential fields in particle systems*. MIT press.

[22] Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Perez & Pablo Ascariz (2012): *PVS linear algebra libraries for verification of control software algorithms in C/ACSL*. In: *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings 4*, Springer, pp. 147–161.

[23] Marieke Huisman, Raúl Monti, Mattias Ulbrich & Alexander Weigl (2020): *The VerifyThis Collaborative Long Term Challenge*, pp. 246–260. Springer International Publishing, Cham, doi:10.1007/978-3-030-64354-6_10. Available at https://doi.org/10.1007/978-3-030-64354-6_10.

[24] Oleg Kiselyov, Chung-chieh Shan & Yukiyoshi Kameyama (2012): *Bridging the theory of staged programming languages and the practice of highperformance computing*. Technical Report, Technical Report 2012-4, National Institute of Informatics, Japan, 2012. URL . . . .

[25] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan & Ian Karlin (2017): *DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools*. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14.

[26] Laurent Simon, Daniel Le Berre & Edward A Hirsch (2005): *The SAT2002 competition*. *Annals of Mathematics and Artificial Intelligence* 43, pp. 307–342.

[27] Jos Stam (1999): *Stable fluids*. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., USA, p. 121–128, doi:10.1145/311535.311548. Available at https://doi.org/10.1145/311535.311548.

[28] Jos Stam (2023): *Stable Fluids*, 1 edition. Association for Computing Machinery, New York, NY, USA. Available at https://doi.org/10.1145/3596711.3596793.

[29] Gregory V Wilson & Henri E Bal (1996): *Using the Cowichan problems to assess the usability of Orca*. *IEEE Parallel & Distributed Technology: Systems & Applications* 4(3), pp. 36–44.