Report of the HPC Correctness Summit

Jan 25–26, 2017, Washington, DC

Report of the HPC Correctness Summit
Jan 25–26, 2017, Washington, DC

REPORT AUTHORS

Ganesh Gopalakrishnan	University of Utah
Paul D. Hovland	Argonne National Laboratory
Costin Iancu	Lawrence Berkeley National Laboratory
Sriram Krishnamoorthy	Pacific Northwest National Laboratory
Ignacio Laguna	Lawrence Livermore National Laboratory
Richard A. Lethin	Reservoir Labs, Inc., Yale University
Koushik Sen	University of California, Berkeley
Stephen F. Siegel	University of Delaware
Armando Solar-Lezama	Massachusetts Institute of Technology

October 4, 2017

DISCLAIMER

This report was prepared as an account of a summit sponsored by the U.S. Department of Energy. Neither the United States Government nor any agency thereof, nor any of their employees or officers, 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 document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Copyrights to portions of this report (including graphics) are reserved by original copyright holders or their assignees, and are used by the Governments license and by permission. Requests to use any images must be made to the provider identified in the image credits (if any) or the first author.

Contents

1 Introduction

Technologies for verification and debugging have made significant strides in the context of general systems software. An investment in such technologies to make them applicable for High Performance Computing (HPC) could lead to substantial improvements in the productivity and sustainability of HPC software development. Such improvements will be essential to fully exploit new exascale computer architectures. Without such investment, there is the possibility of a substantial crisis in our ability to advance the field of HPC, as the complexity of our architectures, algorithms, and applications is moving beyond the ability of our developers. As HPC is of strategic importance to our nation, forming the bedrock of its scientific and technological capabilities, such investment is highly warranted.

1.1 Reasons for the correctness crisis

While the general correctness problem in computer science is well researched, specific reasons that cause HPC-specific correctness methods to turn into an urgent priority include the following (see Figure 1 for an overview).

Growing heterogeneity: Given the widening disparities between CPU, memory, and I/O speeds, computations will be supported by a heterogeneous architectures that include CPUs, GPUs, and special-purpose accelerators [63]. Even today, programs in this space exist around a patchwork of semantic abstractions that are poorly understood individually, and whose emergent behavior is poorly understood.

Massive scale: Attaining exascale will require proper coordination and synchronization across many tasks, threads, and processes [41]. Disciplines that guide programming in this space do not exist, nor do testing methods that unearth defects in this space. Left unchecked, this will lead to deployed systems that yield untrustable results or crash during long-running simulations. Fixing the root cause of bugs in these settings will incur huge latencies during which science domain experts may sit idle or be unproductively engaged with debugging. These costs are known to be already very high (see the sidebars for some concrete examples of costly field bugs in HPC).

Non-intuitive behaviors: The push toward significant energy savings will lead to the use of reduced floating-point arithmetic, delayed state updates across weak memory consistency models, non-determinism caused by

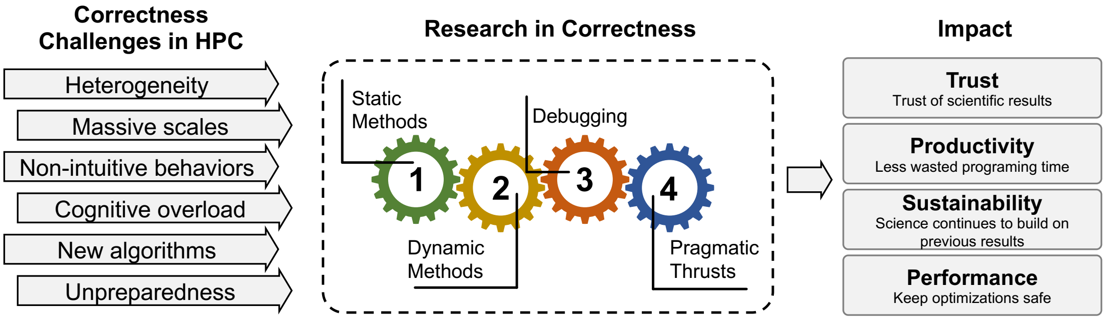

Figure 1: Overview of the existing challenges in correctness for HPC and the research areas that need extensions to address these challenges.

dynamic voltage/frequency scaling, fault recovery steps and inherent application non-determinism. It is impossible to employ ad hoc debugging methods in these situations.

Cognitive overload: The manner in which people write as well as configure full applications is evolving in a direction where human reasoning about correctness is impractical. For example, the NWChemEx computational chemistry code is bringing together SPMD and task parallelism, multiple programming models and runtime interfaces, code generation, and dynamic and adaptive selection of execution configurations. In these settings, manually reasoning about correctness of the application, or even an individual execution, has gone beyond the scope of the individual developers; the need to bring in more automated and/or formal ways has become quite apparent.

New scalable algorithms: New algorithms for numerical computing offer the potential for major improvements in the asymptotic requirements for computation [86]. Kernel-independent and generalized Fast Multipole Methods (FMM) enable matrix vector multiply to be achieved in $O(N)$ time. Support preconditioner methods will enable sparse systems of equations to be solved in near-linear time. Randomized linear algebra and compressive methods will enable systems to be simulated in sampled or compressed form. Such algorithms will bring complex new execution patterns and complex tradeoffs between precision, probability, and time. The ability to exploit these algorithms will require tools to facilitate reasoning about the correctness of their application and implementation.

Community unpreparedness: Compared to sequential programming abstractions that are more familiar to an application scientist, future HPC systems will involve a large slew of semantic abstractions (and relatively newer abstractions such as tasking) whose correct and efficient usage are not first nature to the broad application development community.

Ways to insulate application developers through domain-specific languages (DSL) are badly needed; yet, progress is lacking in this direction. A related but severe problem is that HPC application developers do not have the mindset (or the necessary common repositories) for sharing best practices (including sharing information on bugs and bug-fixes). This lack will hamper the transitioning of new verification research results into practice.

Undoing an optimization leads to a difficult bug

Tensor contraction expressions in Coupled Cluster methods involve products of multiple anti-symmetric tensors. In NWChem, these expressions are computed as a chain of pairwise tensor contraction to minimize operation count. An effort to reoptimize these contractions in the new generation of tensor contraction engine (TCE) by undoing the chaining and reoptimizing the contraction sequence in the existing TCE led to incorrect results. Isolating the source of the error was a manual process: transforming subsets of the contractions and checking for differences. After months of effort, it was found that, depending on the chaining, an additional coefficient needed to be introduced in a key intermediate step, referred to as symmetrization. Explicit specification of the optimization rules and checking whether or not the transformations satisfy the rules could have identified this bug quickly.

1.2 What is in scope, what is not

Current issues in scope: Any defect a programmer can correct by modifying and/or repairing existing programs and/or their support runtime logic are well within the scope of this report. These include the classic sequential program bugs, errors relating to concurrency (e.g., race conditions, incorrect programming under weak memory models), and numerics (e.g., errors in realizing the numerical algorithm using finite-precision floating-point numbers). Defects that may manifest only when a code is scaled up and not during lower-scale testing are also in scope.

We also consider defects that can be eliminated by disciplined code transformations from higher-level, or can be eliminated through better composition and software engineering practices. Defect prevention methods that can be incorporated into best practices, pedagogy, mineable bug repositories, expert tutorials, and IDEs that can prevent or issue warnings about possible bugs are also of great importance, and are well within scope.

When defining the correctness of HPC programs, it is important to keep in mind that the behavior of a user-written program is heavily influenced by the behaviors of the underlying libraries. Not only are the sequential (e.g., numerical and C/Fortran) libraries important, the behavior of communication and runtime libraries (e.g., MPI and OpenMP) directly impact how a user program executes and whether it even makes forward progress.

In this context, it is important to detect and eliminate erroneous arguments supplied to library functions that may cause a program crash. For example, MPI calls must adhere to conventions pertinent to the source language (Fortran or C). However, resource aspects of the runtime and communication libraries are a whole different matter. For example, it is possible to write a user program that may be perfectly correct as far as the users' mental model of an "idealized MPI library" goes, but unfortunately a given MPI library may be unable to park all the asynchronous sends that the user program has issued. Such user programs can either deadlock or crash an MPI library.

A Hard-To-Debug Large-Scale Error

A bug in ddcMD, a parallel molecular-dynamic code, manifested as an intermittent hang when run at large scale on BlueGene/L at LLNL. It took a significant amount of person hours and debugging effort to find the root cause: a message race in which a process could hang waiting for a message that was intercepted by another process. More specifically, the hang occurred when two independent instances of a user-level I/O layer were simultaneously processing two separate sets of buffers—an infrequent pattern that occurred when a small data set was written immediately after a large data set. Due to the semantics of MPI send/recv operations and the use of fixed tags, messages from a small set could be confused for those for a large set and vice versa, thus triggering the hang. Later, after the bug was fixed by the programmer, the bug was used in a blind study, in which researchers developed a tool to isolate this class of bugs without having details of the error (more in [75]). This shows that documenting bug cases can be useful in developing and testing advanced correctness tool.

Most libraries are underspecified, and their implementations often do not come with strong guarantees, such as about the amount of resources provided (e.g., amount of buffering) or whether forward progress or a response within a deadline is guaranteed. These issues are clearly also important, but must be relegated to a longer-term pursuit that involves cooperation from library and runtime designers.

In the same vein, the inability to control the evolving semantics of libraries and programming languages must be kept in mind, requiring cooperation among participant communities. For instance, if a library guarantees a certain order of accuracy for its results (e.g., “9 digits of accuracy”) for specific platforms, we may not be prepared to detect the violation of such contracts by another library on a newer platform to which the code is ported.

When More Than *print* Debugging Is Needed

A scientist experienced hangs in a laser-plasma interaction code (named PF3D) when scaling it to 524,288 MPI processes on LLNLs Sequoia BlueGene/Q system. The scientist spent months trying to debug the problem through print statements to no avail. Moreover, the scientist was unable to reproduce the hang at smaller scales where fully featured, heavyweight debuggers would be more plausible. Using STAT (Stack Trace Analysis Tool), the scientist was able to debug the problem, a race condition between two distinct but overlapping communication code regions. The bug was the result of the application migrating from one version to another more scalable but incompatible one. During migration, the application ran through a compatibility layer that introduced the race condition and ultimately caused the timing- and scale-dependent hangs (more in [76]). This case shows that, although print debugging can be a useful debugging method, the HPC community can benefit from advanced correctness methods and tools to isolate bugs that otherwise can consume months of effort and millions of CPU hours to fix.

Upcoming issues in scope during exascale: We also realize that this report is being written in a timeframe where the underlying designs of exascale systems are experiencing significant disruptive changes. In this era, hardware will often be poorly specified, particularly with regard to features related to the memory model, concurrency and synchronization. With exascale, new hardware features for controlling voltage and frequency, are appearing, as well as advanced features for task scheduling, communication, and synchronization. There will be heavy uses of heterogeneous types of memory (e.g., non-volatile, scratchpad spaces that do not provide cache coherence, etc.). Consider the behavior of an adaptive congestion algorithm [67] in the communications fabric, which may affect or permute the ordering of delivery of messages. If such features are not specified or are incorrectly specified to the runtime or MPI libraries, it will be impossible for any verification technology to guarantee correctness of the software running on it.

The performance behavior of hardware is also moving into the arena of correctness and even safety, where exascale hardware systems are likely to be over-provisioned with transistors, so that not all parts of the system can be used simultaneously while still remaining below the power and cooling limits of the facility, and within the limits for safe and correct operation of the machine. This will impose requirements on firmware, system software and applications to maintain resource usage.

All these issues clearly point to an even greater demand for formal specifications from the hardware vendors on these behaviors and requirements. It will also correspondingly demand that our formal verification and debugging tools use these hardware level formal specifications in order to provide overall correctness guarantees.

The following issues are not directly within scope: There are many issues that are important to keep in mind, but are best relegated to other pursuits that are better able to focus on them. We now mention a few examples of such issues (by no means exhaustive). HPC programs may be brought down by hardware logic errors in microprocessors, GPUs, memory subsystems, and buggy interconnect protocol implementations. Soft errors may corrupt program behavior, but are not considered human-introduced defects. Version control and security-related issues are again somewhat tangential. Finally, the numerical algorithm itself can be incorrect. For instance, errors in the design of the numerical scheme to approximate the idealized mathematics, including incorrectly scheduled coarse/fine meshing, lack of conditioning of the problems, etc., can be considered *algorithmic* defects and not software defects.

Heterogeneity-caused arithmetic divergence results in deadlock

In a recent project [84], an attempt to port some of the MPI processes to run on Xeon-Phis while leaving others running on Xeons caused a curious deadlock that took days to debug. The root cause was the Xeon-Phis calculating the number of messages to be sent (through an expression $\lfloor p/c \rfloor$) differently from how the Xeons calculated the number of messages to be received (also governed by $\lfloor p/c \rfloor$). Unfortunately, the developers had not applied due precautions to their compilation flags, resulting in 63 messages being sent but only 62 attempted to be received, which then caused the deadlock. This bug tells us that a combination of factors—processors being different, floating-point roundoff differing due to the inconsistent use of compiler optimization flags, and the delicate MPI semantics allowing the number of receives posted to exceed the number of sends posted (but not vice versa)—may lead to bugs.

1.3 Suggested research foci, targeted time-frames

We now summarize some of the key short-term, medium-term, and long-term directions identified and elaborated in the rest of this report.

1.3.1 Short-term (1-2 years)

The following short-term foci are overdue, in order to bootstrap the process and bring the community together:

- Launch efforts to apply existing best-of-breed tools to challenge problems, extend those tools, and generally work with HPC applications code as-is. These tools include existing commercial tools as well as those being developed within the research community.
- Advance these tools to address cross-cutting concerns adequately (e.g., tie-in to debuggers and formal tools, instrument existing OpenMP and MPI runtimes to produce event streams, standardize verification tool design around such event streams).
- Bringing advances from the non-HPC community to HPC. These measures could begin as modest as ensuring the capture and sharing of bugs and their fixes, and in general, incorporating lessons from Empirical Software Engineering [85].
- Learn from other communities. For example, study and adapt techniques for concurrency verification from the embedded system verification community. Also, adapt techniques for verifying numerical computations from the cyberphysical systems community.

1.3.2 Medium-term (2-5 years)

The following medium-term directions deserve significant attention:

- Correctness verification of important properties in common HPC software components, including math libraries and widely-used runtime systems such as OpenMP and MPI.
- Building infrastructure to document previously solved correctness issues in the form of bug databases, bug test cases, best testing practices, as well as lightweight mechanisms to automate the extraction of such cases in HPC centers.
- Standardize interfaces to allow composability of correctness checkers, defect isolation tools, and debuggers.
- Investments in the modeling and specification of numerical algorithms, ontologies for the mathematics of the underlying algorithms.

- Support for reasoning about statistical and randomized systems, Uncertainty Quantification and Automatic Differentiation.

1.3.3 Long-term (5 years and beyond)

Investments in these long-term directions will go a long way toward closing the gap between growing system complexity and verification capabilities:

- A few "moonshot" projects, including verification of fundamental logic and numerical properties in multi-physics applications.
- Define metrics for achievable and communicable levels of correctness, especially in simulations with geopolitical consequences, such as weather simulations.

2 Rigorous Methods for Correctness

Formal methods are rigorous mathematical techniques for exhaustively checking that the model of a system under analysis satisfies a set of desired properties. The model in question could be: (1) a piece of code (e.g., the model of a numerical routine); or (2) a set of rules or axioms describing the behavior of some aspect of the system (for example, partial orders can describe the guarantees provided by a memory model, or a set of mathematical rules can describe the behavior of floating point arithmetic). The second perspective is an example of formal methods that are "baked into" analysis procedures or developed to support specific lines of reasoning.

Following the well-known Intel Pentium fiasco, all major chip manufacturers have now adopted formal analysis to verify floating-point hardware. In a recent project at Intel [69], formal methods were deemed so successful in examining critical arithmetic units of Intel's core i7 that traditional simulation-based testing was largely eliminated.[1]

Achieving this degree of adoption of formal methods in HPC is a coveted goal. However, driving a formal methods agenda forward in HPC requires prudence, given the absence of an obvious failure cost model (as happens when chips emerge with silicon defects, where each mask re-spin costs millions of dollars), and also given the sheer complexity of HPC software. More practical are approaches where formal methods are baked into tools so that everyday users are not confronted with modeling their idiosyncratic pieces of code.

[1] In recognition of this success, the leader of this project, Roope Kaivola, won Microsoft's prestigious verified software award of 2014.

2.1 What is the Correctness Problem?

A program is *correct* when it behaves as expected on any execution. This definition begs the question, what behavior is expected? This is in general a difficult question to answer and will naturally vary from program to program. Hence the question of correctness involves two related activities: *specification*—the process of rigorously defining what a program is expected to do—and *verification*—the process of establishing that a program complies with its specification, i.e., that it is correct.

> Correctness of systems hinges on having validated specifications and verification methods that find defects. Challenges in these areas with respect to HPC include the oracle problem, nondeterminism, performance focus, concurrency, scale, domain-specific mathematical abstractions, the use of floating-point arithmetic, and issues that stem from the underlying programming language and runtime support.

2.1.1 Specification

Generic vs. application-specific properties. Certain aspects of the specification of a program come "for free." These include requirements imposed by the programming language used to develop the program. For example, any correct C program should never attempt to read or write to a memory location beyond the bounds of an object, divide by 0, or dereference a null pointer. These requirements are specified in the C Standard, and are inherited by any program written in C.

The application program interfaces (APIs) of libraries and other language extensions used by the program may impose additional requirements. The Message Passing Interface (MPI) standard, for example, requires that all processes belonging to a communicator issue the same sequence of collective calls on that communicator. The OpenMP Standard forbids data races on shared variables. As with C, violations of these restrictions lead to undefined behavior, and should never occur in a correct program.

As important as these language-level requirements are, they do not suffice for specifying correct program behavior. The C program

```
int main() {}
```

satisfies all such requirements, but will not correctly compute the solution to a partial differential equation or the effective neutron multiplication factor of a fission reactor. Clearly, additional techniques must be used to specify *application-specific* properties.

Assertions. *Assertions* are a standard way of specifying application-specific properties. An assertion specifies a boolean expression which is expected to evaluate to *true* whenever control reaches that statement. Most programming languages support assertions in some way. In C, for example, `assert` statements are checked at runtime and a diagnostic message is printed if one fails. Assertions can also be turned off to save time in production runs, but this limits their ability to establish correctness of HPC applications, since many defects appear only at large scale.

While useful for expressing certain correctness properties, assertions are limited to the primitives available in the programming language and cannot easily express relations across different states. It is difficult to assert "forall integers i, if $0 \leq i < n$ then the value of `x(i)` when control exited this function is twice the value of `x(i)` when control entered the function."

Contracts. More sophisticated specification systems such as *contract languages* overcome some of these limitations. For example, the ANSI C Specification Language (ACSL) is used to specify the behavior of C functions. The language provides first-order quantifiers ("for all", "exists") and many other primitives beyond those available in C. ACSL function contracts specify pre-conditions (conditions assumed to hold when control enters the function) and post-conditions (expected to hold when control exits); they also allow one to specify relations between the pre- and post-states.

ACSL contracts are inserted as comments in the code, so they do not impact the usual workflow of compiling and executing the program. Specialized tools (for performing verification or other tasks) use the contracts in different ways. The Frama-C platform, for example, can be used to verify ACSL function contracts using deductive (theorem proving) techniques. Contracts may also be added with respect to collective calls in programming models based on the Bulk Synchronous Parallel (BSP) Model [111].

Certificates. In certification systems, proofs or correctness can be idicated as tactics scripts [10] (e.g., written in Coq [35]). In these systems, both the proof and the imperative code that runs can be auto-generated from the tactics; this is how the certified compiler CompCert [79] and the Certified Kit Operating System CertiKOS [23] are implemented.

Golden models. Finally, sometimes the simplest way to specify an algorithm is to provide an implementation. This implementation could be a simple, inefficient sequential expression of the algorithm. It can then be used as a “golden model” against which production-quality implementations can be compared. Methods that can establish the functional equivalence of two programs could then be used to verify the production implementation.

2.1.2 Verification

It is well-known that the verification problem is undecidable: there does not exist an algorithm that can always answer correctly the question, *does a program satisfy its specification?*. But a technique does not have to be perfect to be useful, and over the years, a large number of practical verification approaches have been studied and implemented. Roughly speaking, we may divide these into two categories.

Tools in the first category attempt to *prove* that a program (with specification) is correct. If the tool succeeds, the program is guaranteed to be correct. The tool can fail to find a proof for a number of reasons: the program is incorrect, the resources required (e.g., time or memory) exceed what the user can afford, or the tool is just not capable of finding the proof. Hence these tools can sometimes show a program is correct, but cannot show a program is incorrect.

Tools in the second category attempt to find defects in programs. If the tool finds a defect, it has shown that the program is incorrect. However, such tools may fail to find existing defects—because they are not capable of finding such defects, or cannot do so within reasonable resource limits—and they may report “false alarms”—possible defects which are not actual defects. Such tools can show a program is incorrect and provide valuable debugging information, but they cannot show a program is correct.

In reality, this distinction is not black-and-white. Rather, these two categories are two extreme points on a spectrum, with most tools falling somewhere in between. For example, model checking techniques can be used to prove that a program satisfies specified properties within certain finite bounds (e.g., on the number of processes or inputs sizes) but leave open the possibility that a defect exists outside of those bounds. Contract-based techniques can show that one function in a program is correct under the assumption that other functions behave correctly. Other approaches can give probabilistic guarantees.

In what follows, we outline some of the major currents in software verification research and practice.

Testing. The most widely-used approach to the correctness problem, testing involves executing the program on a selection of inputs and examining the results. Testing has become a more rigorous discipline over the last 20 years. A variety of techniques for selecting test sets satisfying certain criteria (e.g., statement, branch, or path coverage) have been explored. Language-specific properties, assertions, and even contracts can be tested. The main limitation is that testing cannot establish the program behaves correctly on an input not in the test set. Other limitations in the HPC context are discussed in Section 2.2.

Static analysis. These automated techniques attempt to reason about a program without executing it. Compilers use static analyses to prove properties such as: a variable is never used before it is defined; a variable is only assigned a value of a compatible type; and control never reaches the end of a function body without issuing a *return* statement. The types of properties that can be proved are generally simple (see Table 1).

Dynamic analysis. In this approach, properties are checked as a program executes, or after the program stops using traces that are gathered when the program executes (see Table 1). Like testing, specific inputs are needed, but dynamic analyses can detect defects that are not normally detected by testing, such as the occurrence of a "potential deadlock" even when no actual deadlock occurred during the execution.

Deductive reasoning [57]. This family uses theorem-proving techniques to prove a program satisfies its specification. They can be fully automated or require substantial human interaction. Verification Condition Generation is one increasingly popular approach that generates a number of small theorems from a program+specification which can then be independently "discharged" (proved) using a variety of theorem provers. These approaches often require at least some help from the user, such as code annotations (e.g., loop invariants) or guidance through more difficult proofs.

Symbolic execution [73, 21, 112]. These techniques "execute" a program in an abstract sense, using symbols (X_1, X_2, $\ldots$) in place of concrete values as inputs. The "values" returned by operations are symbolic expressions (e.g., $X_1 - 2.7 * X_2$). Symbolic execution can be used to generate high-quality test sets automatically, to find bugs, and even to prove properties (usually with some restrictions such as bounds on input sizes or loop iterations).

Model checking [32]. This approach is particularly effective for checking temporal properties of concurrent systems, e.g., "no process calls function `f` until every process has exited the ghost-cell exchange." It is standard in the hardware industry and is the basis of many software verification techniques for parallel programs. Typical model checking techniques compute a set of reachable states of a finite transition system. When applied to software this usually enables exhaustive verification of properties with small bounds on the number of processes and other parameters. Model checking can be combined with symbolic execution to cover a wide range of concurrency behaviors and a wide range of inputs.

Certification. In the certification approach, proofs are constructed by the programmer along with the software. Proof assistants automate aspects of this task to multiply programmer effectiveness in generating code with associated proofs. Certifying compilers [78] preserve the proof through code optimization, to produce optimized code along with the compacted proof, in a certificate, of its correctness. The certificate can be rapidly checked against the binary, e.g., as the program starts, to ensure that the resulting binary code meets the specification.

2.2 Challenges in High Performance Computing

The correctness problem takes on a number of special characteristics in high performance computing. Here we enumerate some of the most important points. These points illustrate why specification and verification are particularly needed now in HPC, and identify specific challenges that will need to be overcome.

The oracle problem. HPC programs are often attempting to do new science, so the expected results are usually not known. This makes traditional testing techniques, in which the actual result computed by the program is compared with an expected result, impossible. (There are often specific cases in which the expected result is known, but these are exceptional.) Hence HPC requires verification approaches that do not require knowledge of all expected results. An example would be a tool that proves the functional equivalence of a complex, optimized implementation of some algorithm with a simple, trusted implementation of that algorithm.

Nondeterminism. Many HPC programs are nondeterministic. One source of nondeterminism is concurrency—varying the interleavings of actions from different threads or processes and the computed results may change. The transition to exascale is expected to lead to even more nondeterminism; hardware components will dynamically adjust their execution rates; software implementations will embrace asynchrony to save time and energy; and linear algebra libraries will increasingly employ randomized algorithm techniques to achieve asymptotic speedups [114]. Testing becomes extremely problematic for nondeterministic systems, because a correct execution for some input does not even guarantee the program will behave correctly on a second execution with the same input.

There is no one-size-fits-all approach to nondeterminism. For many programs, the final result is expected to be completely independent of the program's "internal nondeterminism." For others, the final result is expected to vary in expected ways, for example, any difference should result only from the non-associativity of floating-point operations. In addition, many HPC algorithms, such as Monte Carlo simulations, rely on randomness in an essential way. For such "externally nondeterministic" programs, new specification techniques may be needed, for example, to express correctness in terms of probability distributions.

Performance-focus. In traditional software domains, programmers try to express algorithms in the simplest and most natural ways possible. This makes code easy to understand, maintain, and modify. In HPC, there is a tension between these goals and the need for good performance. Simple algorithms that could be expressed in a few lines of code, such as matrix-matrix multiplication, are often re-written using a combination of optimizations, such as loop tiling, loop permutation, and loop unrolling. The programmer must also introduce explicit parallelism. Even though such loop optimizations and loop parallelization can be easily performed by a compiler (automatically or interactively), many HPC programmers persist in performing these optimizations manually, introducing the chance for bugs. The programs are often highly parameterized, and provide multiple implementations of many functions, since different parameters and versions are needed to obtain adequate performance on different platforms. All of these forces lead to programs that are considerably more complex than they would be if performance were not an overriding goal. The increased complexity makes defects much more likely and verification even more necessary.

Concurrency. HPC programs are parallel programs. While some of the verification techniques discussed in Section 2.1 are applicable to parallel programs, the vast majority of verification work targets sequential programs. For example, the ACSL specification language is very mature and used by a number of tools, but has no support for concurrency. Furthermore, modern HPC programs are increasingly *hybrid programs* which invoke multiple concurrency models in a layered approach. These programs are extremely difficult to reason about informally. Yet even among those verification tools targeting parallel programs, very few can be applied to hybrid programs. Finally, the use of weak shared memory consistency models—expected to increase in the exascale era, in order to hide memory latencies—adds another layer of complexity and will require new verification techniques.

Scale. Modern HPC programs are intended to run at an extreme scale, with astronomical input sizes, numbers of processes or threads, execution time, and so on. Often, defects are not observed at small scale. This makes traditional testing and debugging techniques difficult. It can be very expensive and difficult to obtain time on the machines that can support that scale. It can take a tremendous amount of time to run tests at that scale. And debugging a trace involving millions of steps and thousands of threads is an extreme sport. Traditional model checking techniques also scale poorly. Therefore HPC requires (1) verification techniques that can scale to that massive scale, (2) techniques that "downscale" programs so that defects that normally manifest only at large scale will manifest in the downscaled version, or (3) techniques whose cost is independent of scale.

Mathematical abstractions. Many HPC programs use mathematical subjects such as multivariate calculus, differential equations, linear algebra, and (directed) graphs. Specifying algorithms in these areas is extremely difficult if the specification language does not provide appropriate abstractions, such as *derivative*, *matrix*, or *strongly-connected component*. Similarly, proof systems or automated verification techniques must be developed to support those abstractions. Many libraries of this sort exist (see e.g., [40]) but there is much work to increase their adoption in the HPC community and to fill out needed gaps.

Floating-point arithmetic. Many HPC programs involve extensive floating-point computations. The notion of correctness in such programs is intimately tied up with floating-point issues, such as round-off error. Increasingly, developers are reducing floating-point precision to reduce communication costs, and the effect of these tweaks on the output is difficult to gauge.

Tools that can analyze the extent of error introduced by these tweaks and determine whether it is within safe margins for a given application are needed. However, with few exceptions, support for floating-point reasoning is very weak in existing verification systems. Floating-point arithmetic also wreaks havoc on testing-based verification, since it can be difficult to determine the magnitude of an acceptable discrepancy.

Another aspect of floating-point arithmetic is how compilers treat floating-point optimizations. All compilers support a slew of "IEEE-unsafe" floating-point optimization flags that can yield a manyfold improvement in performance, but at the expense of changing the results of floating-point calculations. The flags themselves vary from platform to platform. This aspect of floating-point result variability can render applications incorrect, especially if applied with a performance-focus alone (not minding correctness or result-reproducibility).

Programming language. Most HPC programs are implemented in Fortran or C++ (or both), while many verification tools target C or Java. While many of the ideas and even specific techniques are language-independent, significant engineering effort is required to extend existing verification tools to new programming languages.

3 State of the Art and Successes

There have been several notable successes in establishing rigorous methods in support for HPC. Many of today's successes lie in the use of static analysis, dynamic analysis, focused testing with non-determinism control, anomaly detection specific to HPC, and debuggers focused on HPC. The use of rigorous and systematic methods in many recent projects, while not as mature, has already shown considerable promise.

Todays correctness practices comprise a body of domain-specific testing, and tools and frameworks to debug, pinpoint, and fix errors that escaped the testing phases. Most of these practices are ad hoc—they often require heavy-weight program instrumentation and analysis, and are tailored to specific classes of bugs (e.g., data races), programming models (e.g., MPI), and runtime systems and platforms. In addition, they are largely not composable, and are often difficult to adopt in practice in the workflow of large scientific code bases. As a result, it is not uncommon for programmers to end up chasing elusive bugs by "printf" debugging. When an error is reproducible, parallel debugging tools can be very helpful in diagnosing an error, though this process tends to be manual and requires a significant amount of domain expertise.

We split the state-of-the-art practices into two broad categories: *testing* and *tools for bug detection and localization.*

3.1 Testing

Although testing scientific software is generally considered to be difficult [72], it is nevertheless the mainstay of today's verification approaches. Conventional testing, such as *regression* testing, *white* and *black box* testing, and *functional* testing are used to check exceptional situations and corner cases. Finer-grained levels of testing, such as *unit* testing, are however less common, specially in legacy HPC applications [60], as the effort of generating these tests is difficult to justify for domain scientists. State-of-the-art testing practices rely on the reproducibility of results under fixed inputs, and usually check domain-specific physics laws. Assertions are used to check expected behaviors and results at different code locations.

Validation through the use of analytical solutions to check results against experimental data is also employed to some degree. Verification is also supported through techniques such as methods of manufactured solutions (checking against solutions to made-up idealized cases) as well as higher level criteria such as the order of convergence.

Challenges of Testing. The main challenge to test scientific codes is the large effort in generating test cases, specially for complex multiphysics codes. Tests require data input, and in HPC applications this can be very large; thus exhaustively and manually testing every input is infeasible. An option for HPC codes is to scale down the domain, but is often infeasible to do without introducing inconsistencies. HPC codes tend to use user-defined data types and complex and long data structures, which may be passed through functions, and initializing these structures to create different test cases is a huge effort. Non-determinism and lack of tool support are other important impediments to testing.

Most testing today is limited to a small-scale setting (small number of processes and threads, and small input size). HPC resources are shared and it is practically impossible (or at least very costly for an HPC center) to perform frequent testing (e.g., nightly testing) of all applications at large scale. This limits the scope of bugs that can be covered by testing—it is expected that the behavior that is checked at small scale extrapolates to large scale, though that is often not the case in practice.

Some of the tools that are used to test HPC software include: tools to

Table 1: Some of the existing tools and frameworks to detect and localize bugs in HPC programs

	Formal Method	Static Analysis	Dynamic Analysis	Control of Non-determ.	Anomaly detection	Parallel debugging
Serial Code						
Clang Static Analyzer–static analysis bug detection in C/C++ [29]	✓	✓				
Clang Sanitizers–runtime bug detection (e.g., AddressSanitizer) [30]		✓				
Klocwork–on-the-fly, scalable static analysis [74]		✓				
Multi-threaded Code						
Valgrind–memory management error detection and threading bugs [121]			✓			
Intel Inspector–memory and threading error debugger [65]		✓	✓			✓
CUDA-MEMCHECK–memory access errors detection in GPU code [36]			✓			
ThreadSanitizer–data-race detection for multi-threaded programs [31]			✓			
ARCHER–data-race detection for OpenMP programs [5]		✓	✓			
GMRace–data-race detection in GPU programs [131]		✓	✓			
GKLEE–concolic verification GPU programs [80]	✓	✓	✓			
GPUVerify–static (SMT-based) verification of GPU programs [28]	✓	✓				
DTHREADS–deterministic execution of multi-threaded programs [81]			✓	✓		
CUDA-GDB–NVIDIA CUDA gdb-based debugger						✓
Insure++–runtime error detection [64]			✓			
Multi-process Code						
MUST–MPI deadlock detection [54]	✓		✓			
UMPIRE–dynamic error detection for MPI [123]			✓			
ISP–dynamic formal verifier for MPI [120]	✓		✓			
FlowChecker–communication errors in MPI [26]			✓		✓	
AutomaDeD–anomaly detection in parallel programs [17]			✓		✓	
Prodometer–progress-dependence analysis to diagnose hangs [88, 75]			✓		✓	
Vrisha, WuKong–scale-dependent bug detection [133, 134]		✓	✓		✓	
Scale-dependent overflows detection [77]		✓	✓			
MPIWiz–record-and-replay for MPI [128]			✓	✓		
Retrospect–deterministic replay of MPI applications [14]			✓	✓		
ReMPI–record-and-replay for MPI [102]			✓	✓		
NINJA–noise injection to make ND bugs in MPI manifest faster [103]			✓	✓		
SReplay–record-and-replay for one-sided communication [97, 96]			✓	✓		
Hybrid (multi-threaded, multi-process) Code						
CIVL–formal Verification of parallel programs [132]	✓	✓				
Relative Debugging–comparison of two program executions [39]		✓	✓			✓
STAT–stack trace analysis tool [2]		✓				✓
TotalView–parallel debugger [118]			✓			✓
DDT–parallel debugger [37]			✓			✓
LGDB, CCDB–Cray command-line parallel and comparative debuggers			✓			✓

write regression tests for numerical software, such as ATS (Automated Testing System) [4] developed at LLNL, continuous integration frameworks, such as Bamboo [3], and C/C++ testing frameworks, such as Google Tests [49], and Boost Tests [13].

3.2 Infrastructure for Bug Detection and Localization

There exists a variety of tools and techniques that have been proposed to detect and to isolate software defects in HPC applications. We categorize these frameworks in six groups: *static analysis*, *dynamic analysis*, *formal methods*, *anomaly detection*, *non-determinism control*, and *parallel debugging*. We present a short definition of each of these methods as follows, and Table 1 lists some of these tools. Note that different methods are not mutually exclusive and it is common for tools to use a combination of methods; for example, a tool may perform static analysis in one phase, and then to perform dynamic analysis or formal verification in a another phase.

3.2.1 Static Analysis

Static analysis examines the code without executing the program and it is perhaps the first line of defense against bugs for programmers. These checks are typically performed when the program is compiled and can warn the programmer of possible errors in the program. At the moment of writing, the Clang compiler has currently more than 670 diagnostic flags. Static checks are performed as well in Integrated Development Environments (IDE), which can detect errors even before the compilation phase (Eclipse [20]). Klocwork [74] is an on-the-fly static code analysis tool that is used at LLNL and other DOE laboratories to detect bugs at early stages.

More advanced static analysis tools can reason about the semantics of code and find bugs that traditional compiler warnings cannot find. These tools may use symbolic execution and abstract interpretation techniques to explore all execution paths in the program. An example in this category is the Clang Static Analyzer [29].

While compilers perform a large number of static checks, this all relies on compilers being correct themselves. However, compilers can have bugs that often arise when performing optimizations (specially under concurrency [24])—these in turn may yield application bugs in extreme cases that are very hard to isolate. The test and check of code transformations that are semantic preserving are an active area of research [78]. Commercial compiler vendors dedicate major resources to assembling test cases and regression testing and have years of experience in the engineering of compilers for correctness and performance; this is why the best commercial compilers continue to outpace their open source counterparts in correctness.

3.2.2 Dynamic Analysis

Most of the existing bug detection and localization tools for HPC perform dynamic analysis [76]. Dynamic analysis involves checking correctness by executing the program with an specific input (or a set of inputs). There are two broad categories of dynamic analysis, *online* and *offline*; in the former, checks are performed during the application's execution time, whereas in the latter the checks are performed after the application has finished execution, usually by analyzing traces of the application that were gathered during the application run. For HPC programs that run on multiple processes (e.g., MPI programs), traces are usually gathered from all processes and then aggregated for further analysis.

A large group of dynamic checkers are *memory checkers* since many bugs arise due to incorrect use of memory. The Valgrind memory checker [121], for example, supports MPI programs and can perform memory checks in all MPI processes. A subgroup of memory checkers, detects data races in multithreaded programs, including checking in heterogeneous systems with accelerators. The Intel Inspector [65] and ThreadSanitizer [31] support data-race detection of pthread programs. ARCHER [5] performs data-race detection in OpenMP programs on top of ThreadSanitizer and static analysis.

Other dynamic analysis frameworks for bug detection are tools to detect deadlocks and synchronization problems in MPI (e.g., MUST [54], ISP [120], and DAMPI [124]), tools to detect errors at the message-passing layer (e.g., FlowChecker [26]), and tools to perform progress analysis of processes to isolate the origin of hangs (e.g., Prodometer [88, 75]).

In hybrid programming models, data races occur easily and are notoriously hard to find. Conventional state-of-the-art data race detectors exhibit $10\times - 100\times$ performance degradation and do not handle hybrid parallelism. UPC-Thrille [92, 93, 91, 119] is the first complete implementation of data race detection for distributed memory programs. In benchmark programs, UPC-Thrille found all previously known data races with at most 50% overhead when running on 2048 cores.

Finally, dynamic analysis techniques have been proposed to tune the precision of floating-point programs. Precimonious [101] is a dynamic analysis approach that performs a search on the types of the floating-point program variables trying to lower their precision subject to accuracy constraints and performance goals. Blame Analysis [100] can be used to further speedup the precision tuning of Precimonious. Blame Analysis functions by executing floating-point instructions using different levels of accuracy for their operands. Evaluation on ten scientific programs shows that Blame Analysis is successful in lowering operand precision.

3.2.3 Formal Methods

Formal methods, which allow specification and verification of software, haven been used to certain degree in HPC. The SPIN model checker [58] has been used in various approaches to check properties of parallel programming models, including MPI [110] and distributed task-based models [89]. CIVL [132] is a symbolic execution-based verifier that can analyze programs using many HPC-relevant parallel programming models, including MPI, OpenMP, Pthreads, and CUDA. The ARCHER race detector [5] based on formal loop carry independence analysis and happens-before analysis detects race conditions in OpenMP programs. Verification of producer-consumer synchronization achieved through the use of named barriers is studied in [109]. Additional success cases of formal methods are listed in Section 3.4.

3.2.4 Control of Non-determinism

When debugging a parallel program, programmers must first reproduce the bug; however, because of the non-determinism that comes from parallelism and non-deterministic inputs, reproducing bugs can be a challenge. Some data- and message-race bugs, only manifest themselves one time every many (possible hundreds) runs. Thus, programmers often use tools to control the non-determinism of parallel programs when debugging. A common method is to use *record-and-replay* techniques [102] to record the execution of a program when the bug manifests, and then to replay the same execution deterministically using a parallel debugger. Other tools allow programmers to speedup the manifestation of the bug, i.e., to make it manifest with more likelihood in less runs (NINJA [103]). SReplay [97, 96] is the first software tool for deterministic record and replay for one-sided communication. A key innovation in SReplay is that it allows the user to specify and record the execution of a set of threads of interest (sub-group), and then deterministically replays the execution of the sub-group on a local machine without starting the remaining threads.

3.2.5 Anomaly detection

Anomaly (or outlier) detection—detection of behavior that is significantly different from the expected (or normal) behavior—can be used to isolate software defects. Here, *behavior* can be broadly defined in terms of performance or correctness metrics, from slower-than-usual execution times to out-of-range floating-point computations or unusual logic actions (e.g., some branches taken more often than others). Most methods in this domain use traces that are obtained under error-free runs to define *normal* behavior, and then traces that are collected when an error manifests are used to detect and localize problems. Some of the work in the area include DMTracker [44], Mirgorodskiy et al. [87], AutomaDeD [17], and Bronevetsky et al. [18]. Anomaly detection has been used as well to detect scale-dependent bugs, i.e., bugs that hide themselves at small scale but that manifest at large scale (Vrisha [133], WuKong [133]).

3.2.6 Conventional Parallel Debugging

Parallel debuggers allow programmers to control and to examine the state of threads and processes in a parallel program. These tools have advanced graphical interfaces that support a wide range of features to visualize the value of variables in the program and can operate under several parallel programming models, including OpenMP, CUDA, and MPI. Two of the most used commercial options are TotalView [118] and DDT [37].

A very effective way to debug large-scale parallel programs is stack trace analysis; the STAT [2] tool provides a lightweight method to gather and merge stack traces of parallel processes and to present them to programmers in an intuitive way. Relative Debugging [39] can assists programmers to locate errors by observing a divergence in relevant data structures between two versions of the same program as they execute, and is particularly effective when code is migrated from one platform to another.

LGDB (Cray Line Mode Parallel Debugger) is a GDB-based parallel debugger developed by Cray that is used in DOE scientific computing facilities, such as the National Energy Research Scientific Computing Center (NERSC) and Argonne Leadership Computing Facilitys (ALCF). CCDB is a GUI tool for comparative debugging that runs LGDB underneath. Its interface makes it easy for users to interact with LGDB for debugging.

3.3 Correctness through Correct-by-Construction Certification

As mentioned above, in the certification approach, a rigorous software development methodology involves writing the proof of code correctness along with the code, using a proof assistant. In some cases, one does not write the code at all – it is auto-generated, along with the proof, and the layer specifications, from a sketch written in a tactics language [126]. Recently this approach has showed great promise in the systems software field, with certifying compilers and ways of developing efficient code along with strong proofs of correctness. The certified software is engineered for proof modularity, so that independently certified parts can be linked for ensuring correctness of the overall system. The scope of proofs within this community includes reasoning about concurrency, security, storage systems and floating point correctness. A rich library of code, proof objects, and mathematical ontologies are available for developers to draw on in creating larger systems. There is great pick-up of this technology within the computer systems research community—it is now expected and rewarded by the top conferences that new systems software technologies are accompanied with the formal proofs of correctness [25]. The certification approach is also rapidly gaining traction in the embedded computing field (for correctness of systems with respect to safety requirements) and cybersecurity field (for proofs of freedom from particular vulnerabilities). Although the software engineered with this approach is highly modularized, e.g., into "Deep Specifications" [50], the certification approach does not impose significant performance penalties. Full and performant concurrent operating systems are available that are fully certified [51]. Hardware cores have been designed that export their functional properties (e.g., opcode semantics, memory model, and synchronization semantics) and are formally verified to these specifications. The specifications are exported so that the software above can be certified in the context of proved hardware semantics. Recently there has been progress in applying certification approaches to randomize algorithms [8], which might lead to ways to certify numerical methods based on statistical assumptions.

3.4 Successes due to rigorous and systematic methods

We enumerate some recent advances in the field of verification that are related to HPC. While small scale and initial prototypes, and in some cases very difficult to achieve, they indicate the feasibility of verifying HPC programs, the availability of powerful initial tools, and that the field is primed for success.

- Certification of C programs with floating point: Ramananandro et. al. [99] developed a tool VCFloat that demonstrated that floating-point computation can be verified in a homogeneous verification setting based on Coq only. Ramananandro used a new formal specification of IEEE 754 Floating point called Flocq [12].

- Verification of a C numerical solver for a wave equation. Boldo et. al. [11] used a tool Frama-C that statically analyzes a C program to produce a proof that can be checked by a range of different tools, including Coq with Flocq, and also SMT solvers.

- Rigorous mixed-precision floating-point tuning methods, such as in FPTuner [27], promise to lead to optimization methods that can reduce data movement and energy consumption, while providing rigorous absolute-error related guarantees.

- UPC-Thrille [92, 93, 91, 119] is the first complete implementation of data race detection for distributed memory programs. The implementation tracks local and global memory references in the program and it uses two techniques to reduce the overhead: 1) hierarchical function and instruction level sampling; and 2) exploiting the runtime locality specific to Partitioned Global Address Space applications.

- CIVL [132] is the first symbolic execution-based verifier that can analyze programs using many HPC-relevant parallel programming models, including MPI, OpenMP, Pthreads, and CUDA. It has also been applied to "hybrid" programs that use more than one programming model. It has found bugs in several shorter examples, including race conditions in an OpenMP code offered in tutorials.

- The ARCHER race detector [5] based on formal loop carry independence analysis and happens-before race checking helped detect a nasty race condition (that previously defied debugging for months) within HYDRA, a large multiphysics application developed at LLNL, which is used for simulations at the National Ignition Facility (NIF) and other high energy density physics facilities.

We list success cases of rigorous and systematic methods that are outside of HPC but that exemplify how these methods can successfully be applied in other complex systems:

- Since 2011, engineers at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in their systems [90]. The use of Temporal Logic of Actions (TLA+) helped Amazon find several bugs and to improve the overall confidence of their systems. This is an excellent example of how formal methods have been used in large real-world distributed systems.

- The STACK [125] analysis system uses symbolic execution to identify points in the code that can lead to errors because of undefined behavior. In particular, it has been very successful in identifying checks that could be removed by the compiler because of undefined behavior. The tool was used to find over 150 bugs that got repaired in open source programs including the Linux kernel and the Postgres database system. The program has also been used successfully at companies such as Intel [52].

- Concolic Testing (also known as DART: Directed Automated Random Testing or Dynamic Symbolic Execution) alleviated the limitations of classical symbolic execution by combining concrete execution and symbolic execution [47, 108]. Concolic testing [22] has been demonstrated as an effective technique for generating high-coverage test suites and for finding deep errors in complex software applications. The success of concolic testing in scalably and exhaustively testing real-world software was a major milestone in the ad hoc world of software testing and has inspired the development of several industrial and academic automated testing and security tools such as PEX, SAGE, YOGI, and Vigilante at Microsoft, Apollo at IBM, ConBol and Jalangi at Samsung, CATG at NTT Laboratories, and SPLAT, BitBlaze, jFuzz, Oasis, and SmartFuzz in academia. Some of these tools have been successfully applied to discover critical functional abugs and security vulnerabilities in real-world software. For example, SAGE [48] has found many new expensive security bugs in many Windows applications, and is now used daily in various Microsoft groups. SAGE found one-third of all Win7 WEX security bugs.

- In recent work, file sharing and synchronization services supported by DropBox was subject to property-based testing [62]. This shows that large scale distributed systems, developed in an ad-hoc way without formal methods, can have major holes, even when in massive use, and that formal methods can be practically used to close these gaps.

4 New Research, Impact

In this section, we identify extensions to existing state-of-the-art practices and research needed to make the correctness techniques described so far work in the context of HPC applications. Our description covers static methods §4.1, dynamic methods §4.2, debugging §4.3, and pragmatic issues §4.4.

4.1 Static methods

Static techniques for program analysis and verification could help increase confidence in HPC results, as well as reduce development time by reducing the effort needed to track down and fix bugs. This potential research agenda can be broadly divided into the following thrusts: runtime focused thrust §4.1.1, numerical algorithms thrust §4.1.2, specification thrust §4.1.3, and thrust towards verification of compilers and libraries §4.1.4.

4.1.1 Runtime system focused thrust

Many recent successes in applying formal reasoning techniques to systems software could be applied directly to the runtime systems underlying a lot of HPC codes. In particular, the MPI and OpenMP runtimes could be good targets for such an effort.

There is a wide range of approaches that could be applied to these runtimes; at one end, we could attempt to verify these runtimes for the absence of memory errors and race conditions. At the other extreme, recent successes in the development of fully verified system components, from file systems to compilers, suggest that it would be possible to develop fully verified implementations of at least some parts of the MPI or OpenMP runtimes.

> New research in support of the correctness challenges in HPC is needed in the areas of static methods, methods that have runtime system focus, numerical algorithm focus (with emphasis on floating-point usage), and focus on verifying compilers and libraries. The use of dynamic methods, debugging, and many pragmatic thrusts (even smart IDEs) can go a considerable way. Rigorous methods are essential for many aspects of concurrency, including shared memory consistency models.

Another potential target for this approach is the compiler itself; there is already significant work on verified compilers, but additional resources could help push this work to focus on verifying the kinds of optimizations that are most relevant to HPC code, such as cache optimizations.

This agenda would have the benefit of immediately eliminating entire classes of bugs from HPC systems without the need for any buy-in from the HPC application development community; application developers would just swapp one library for another.

There are a variety of techniques that can find memory errors and data-races in existing software. Applying these techniques to the runtimes that support most HPC applications could yield some insights, helping improve confidence on such systems. A more ambitious goal would be to gradually rewrite major components of the standard HPC runtime using mechanisms more suited for verification. The goal would to move beyond verifying the absence of memory errors and data races, and towards verifying important functional properties, such as ensuring that no messages will be dropped by the runtime. In addition to the implementation effort, this approach requires new research into the formalisms necessary for verifying such properties.

4.1.2 Numerical algorithms focused thrust

The second thrust would involve verifying the numerical computations themselves, under some assumption that the underlying runtime systems are correct. This has the potential for much bigger payoff relative to verifying the runtime system, given that most HPC developers run into errors of their own making more often than they run into MPI bugs.

On the other hand, this also requires significant new research around two major issues: reasoning about numerical and floating point computation, and making it possible for HPC users to write specifications.

Automated reasoning for reals and floating point Numerical computations can be analyzed at two levels; first, they can be analyzed assuming ideal, real-valued computation. In reality, however, computation involves floating-point numbers. In many settings, the assumption of real-valued computation can help uncover outright bugs in the computation, but reasoning about floating points is necessary to reason about issues such as convergence and precision.

There has been significant recent work on automated reasoning techniques for real-valued computation. For example, the dReal [45] system is able to reason about logical formulas involving transcendental functions such as *sin*, *cos*, *log* and exponents, as well as formulas involving integrals. Support for transcendental functions is provided by recent tools such as FPTaylor [113] that underlies FPTuner [27], a rigorous floating-point precision tuner. These tools are yet to be scaled to the sizes often required by DOE physics codes.

The complexity of reasoning about numerical code depends on the gap between the specification and the implementation. For example, in checking that a tiled implementation of a computation is equivalent to an un-tiled version, the gap between specification and implementation is relatively small. Such a verification problem does not require deep reasoning about the properties of real-valued or floating-point computation, since both versions are performing the same computation, only in different order, so a modern verification tool can abstract away the details of the floating-point computation and focus on verifying that the loop structures are equivalent. This level of reasoning can be achieved with existing technology, and is indeed performed by systems such as STNG [116], which reasons about the equivalence between low-level stencil implementations and their high-level specifications.

At the other extreme, reasoning about the fact that Conjugate Gradient will indeed find the solution of a linear system of equations is extremely challenging, and requires detailed knowledge of linear algebra to be encoded in the verification system. Doing so is likely quite possible and could yield substantial benefits in correctness.

For floating-point arithmetic and associated error versus performance tradeoff analysis, formal methods can provide safety nets for enabling what practitioners like to do—i.e., push on performance while skimping on precision. Formal methods are essential to define what is safe for the situation at hand (error containment, ensuring convergence), as floating-point precision tuning is often not that effective without modeling the usage context.

Unsafe optimizations for floating point: One aspect of compilation that has received very little attention is how fast the code can be made by pursuing "unsafe math" optimizations (upto 5 times faster for some codes, thus a highly tempting option), and yet, these optimizations introduce far more than the normal IEEE round-off error. A recently developed tool FLiT [43] is able to portray the number of different answers one can obtain even for a single test routine. Unfortunately, the meanings of compiler optimization flags vary across compilers. All this can lead to result variability to an uncalibrated extent, affecting both correctness and reproducibility. This is another aspect of the aforesaid error versus performance trade-off analysis that merits rigorous support.

4.1.3 Specifications Thrust

One of the major roadblocks in the adoption of verification and formal reasoning technology is the difficulty of writing formal specifications. There are two ways in which other communities have addressed this problem. First is to focus on general properties that every program should satisfy (such as memory safety or race freedom). Tools can be designed to verify such properties without the need for the tool user to have to provide individual specifications for every program.

Second is to focus on specific domains and write specification languages that are tailored to those domains. In the HPC context, there is a recent trend towards domain specific languages that has been fueled by recent successes in generating very high-performance implementations from high-level domain specific notations. Examples of such high-performance systems include TCE [9, 56], Halide [98] and Spiral [95].

A move towards domain-specific languages can help sidestep the verification problem and make it possible to introduce verification technology without burdening the user. This can be done in a few different ways. First, verification can serve as a form of translation validation. Most production compilers are developed by large organizations and used on millions of programs, so they have no obvious errors.[2](the CSmith [129] work has shown how buggy production compilers can be). Domain-specific languages are less likely to have either of these characteristics, so they are more likely to have bugs. A general verification infrastructure to guarantee that the output of the DSL compiler is consistent with its input could be very useful. Moreover, the DSL compiler could provide a trace of its derivation steps that could significantly simplify the verification task. Verification could also help in those (hopefully rare) cases where the output of the DSL compiler needs to be modified for any reason.

Recent work in the context of stencils [70] has shown that domain-specific compilers can interact with verification in other ways. In the STNG system, a low-level stencil computation is analyzed to extract a high-level specification of the computation in the Halide DSL. This automatic extraction of the specification can make it possible to leverage the power of the high-performance DSL in the context of a low-level legacy implementation.

A DSL can also be embedded in a regular "host" programming language; simple C is perfectly able to express loops for linear algebra, sten-

[2] There are two ways of constructing a piece of software: One is to make it so simple that there are obviously no errors, and the other is to make it so complicated that there are no obvious errors. - Tony Hoare

cils, and solvers in a clear and compact "textbook" way and high performance codes can be automatically generated from such specifications [82]; no DSL is needed for such domains. A Domain-Specific Embedded Language (DESL) [61] has many advantages such as avoiding a "tower of Babel" of many different DSLs, clear semantics for linking with other modules, and benefiting from ongoing research and development for the host language. Investment in optimization and verification of the host language benefits all programs in that language.

4.1.4 Verification of compilers and libraries

The compiler technology has advanced enough to automate many optimization steps: loop transformations, data layouts, vectorization, etc. HPC applications increasingly rely on optimizing compilers, auto-tuners, and optimized libraries to achieve portable performance. This trend is advantageous from a correctness perspective. Beyond verifying every manual optimization in an application, ensuring correctness of the compilers and libraries can help us ensure more parts of the software toolchain are correct.

Given the shift toward automated data layout and iteration-space optimizations achieved through portability layers such as RAJA [59] and Kokkos [42], the integrity of such "tall compilation stacks" can become single points of failure due to bugs they can introduce in all their generated code. Code generation may also be able to encompass the generation of complex data structures that are not feasible for humans to originate. On the flip side, these stacks can also serve as *single opportune points of intervention* for maximally impactful uses of formal methods.

Polyhedral optimizations involve the use of well-specified transformations implemented through complex tool chains. Whereas the test suites associated with the tool chains can catch some bugs, they can be sensitive to initialization values used for inputs [7, 104]. Verifying the code generated by a polyhedral optimizer, through a combination of verification, exhaustive testing, and certification, is an attractive yet feasible endeavor.

One possible route for a verified HPC compiler is to base it on CompCert [115]. Most commercial and open source compilers are implemented with traditional non-certified programming techniques, making their verification difficult. The route would follow the path of engineering the range of optimizations for HPC on an existing certified compiler. Once this is done, domain-specific embedded languages (DESLs, see Section 4.1.3) implemented in the certified host language and compiler would benefit from the certification capabilities.

4.1.5 Other thrusts

Formal methods based on automata-theoretic modeling can be applied to expressing component interfaces in the form of interface automata [38], or learning the behavior of code that a human expert does not understand (the latter has been successfully applied in the Android operating system context).

In the area of formal shared-memory consistency models, formal methods are the only satisfactory approach in that while ad hoc testing and manual reasoning may find missed cases, they do not help provide rigorous guarantees that cover *all possible executions* allowed by a memory model.

More importantly, formal methods can eminently point to formalized testing adaptations, as in a recent paper [127], where formalizing the underlying relations of memory models in Alloy allowed the authors to generate tests that distinguish subtly different memory consistency models.

4.2 Dynamic methods

Static methods are widely acknowledged for their soundness and precision, but face challenges when applied to large realistic code bases. Code sizes, layers of abstraction, and combinations of programming languages (e.g. C++ and Fortran) all pose problems to static methods.

In recent years, dynamic methods have emerged as a practical and powerful alternative to static approaches. Dynamic methods make inferences based on observed execution(s) of the program. While no guarantees can be provided for any other unobserved execution, the hope is these inferences are generic and useful to developers. Tools such as Valgrind, and Intel ThreadChecker have widespread adoption in the software community and have been shown to be able to handle very complicated codes, such as the Linux kernel. Compared to static approaches, dynamic methods require a guided process that invloves developer feedback and steering.

Dynamic symbolic execution (or concolic testing) [22, 47, 108, 21] is a dynamic analysis method where constraint solvers are used to steer the program execution along various feasible execution paths of a program. Though dynamic symbolic execution has been successfully applied to find subtle bugs and security vulnerabilities in sequential software, little has been done to scale it for parallel and concurrent software [106, 107]. Research is needed to combine conventional model-checking and active-testing techniques [105, 68] for concurrent programs with dynamic symbolic execution to make them work for HPC programs.

Online dynamic analysis methods have the advantage of being deployable in production environments and in conjunction with the actual libraries available on a platform. Therefore, they are practical and can provide guarantees pertinent to a particular realization. However, these approaches cannot store or process complete traces and need to minimally perturb the application. Online analysis can benefit from further research into the identification and analysis of relevant interleavings (the partial order) in the presence of multiple concurrency models. Offline dynamic trace analysis methods can afford to perform multiple potentially expensive error analysis passes on the traces from large-scale runs. These methods rely on methods to lower the tracing overhead, including the identification and discovery of relevant events to instrument so as to perform the analyses of interest.

Both online and offline analysis require research to improve their scalability with concurrency and input size. Often, traces contain low-level operations not immediately correlated with the source level. Examples include basic-block level fine-grained control-flow information or load/store information. Formal methods can help narrow the gap between low level traces and human understanding of the code. These inverse-mapping relations are crucial to explain bugs in higher level terms. Formal methods can play a significant role in critical design choices such as flowing traces into a checker, shifting between offline and online analysis, and the use of statistical (sampling) based approaches to reduce the amount of tracing done while providing probabilistic guarantees (e.g., [19]).

If support for automated code transformation is desired, research is needed in developer presentation tools that can provide reverse mappings across multiple levels of abstraction. Ideally, the tools could suggest source-level transformations to fix an identified correctness problem. In a large HPC application composed from many libraries, these analyses should be composable and not interfere in terms of correctness or performance. While they can aid in debugging HPC applications, bugs in these analyses can dissuade user adoption. A well-constructed verified toolbox of analysis can complement verified HPC runtimes in ensuring that the bugs identified are indeed from the user's application.

4.3 Debugging

Traditional debugging tools and techniques help to identify the root cause of errors by allowing programmers to control the application and to inspect the applications state (e.g., value of variables) in an execution. Parallel debuggers control and inspect the execution of many threads and processes, a task that can be computationally expensive given the high degree of parallelism in todays largest HPC systems. A disadvantage of these methods is that they are manual in nature, i.e., the programmer has to reason about the program and manually find the bug. Advanced debugging techniques and tools help programmers to automatically pinpoint bugs—some with fine granularity, e.g., lines of code. These automatic methods, however, are mostly dynamic (i.e., they can only make decisions based on a given input and execution) and may suffer from high false-positive rates. There exists complementary techniques that aid in the debugging process, such as record-and-replay techniques, which allow programmers to deterministically reproduce bugs. These techniques are of great help to isolate software defects that manifest themselves rarely or non-deterministically.

Extensions to the state-of-the-art debugging methods are required in the following areas:

- Scalable debugging tools to isolate software defects that manifest at large scale, where scale represents number of threads, number of processes, and/or input size. Two categories are important in this area: (i) scalable tools to help control and analyze a program in a large-scale execution when a bug manifests itself, and (ii) debugging tools to isolate scale-dependent bugs using small-scale runs.

- Accurate automatic debugging techniques to help programmers automatically find the origin of errors to a fine degree of granularity, such as the line of code, function, or code component. In particular, research is needed to improve the accuracy of existing techniques in this category. Metamorphic testing is promising in this regard [71].

- Methods to control non-determinism when debugging, such as record-and-replay, thread/process schedule controlling, and thread/process schedule enforcing techniques, are needed.

4.4 Pragmatic thrusts

Smart IDEs. In the recent years, Integrated Development Environments (IDEs) have gotten smarter in dicovering bugs and common programming mistakes at development time. As a programmer types her/his program, these IDEs perform on-the-fly code analysis and instantaneously report syntax errors and complex static errors. Examples of such smart IDEs include Eclipse, Intellij IDEA [66], and CLion [33]. These IDEs not only perform on-the-fly analysis and report static programming errors, they also utilize state-of-the-art program analysis techniques to help programmers with code refactoring and navigation. In practice, smart IDEs have been found to significantly improve programmer productivity. In supporting HPC software development, smart IDEs can be extended to find concurrency related bugs, such as data races, deadlocks, and atomicity violations. Existing smart IDEs cannot reason about hybrid programming models often used in HPC programs. Correctness tools and techniques can be made easily accessible to HPC programmers if the formal program analysis techniques developed for HPC programs can be integrated into these IDEs.

Software design, specification, and testing practices. An HPC correctness campaign can target a few key steps in the software development lifecycle to improve our confidence in their correctness. First, many of the target DOE applications for correctness verification are monolithic and lack formal specification. Research is needed into methods for "reverse engineering" specifications, such as the lifting technique implemented in Helium [83]. This process will be helped by the design of tools and techniques to decomposing monolithic applications into verifiable units and composing the results of verification. Second, conventional software engineering teams employ code guidelines, such as Code C++ guidelines, Google style guide, etc. to avoid common design and programming errors. Many of these coding guidelines are associated with tools that can check for conformance. The availability of such tools for HPC software (e.g., precluding the use of MPI_COMM_WORLD would help improve software quality and end-user's trust in their correctness. Third, a significant challenge in regression testing of computational science applications is assessing when a change in the program output is significant. Often, mandating that the output remain bitwise equivalent is too strong a requirement and may not be possible in the case of non-deterministic applications, but selecting an arbitrary numerical tolerance may result in missed bugs. Research is needed to adapt regression testing to applications with large amounts of floating-point arithmetic.

Bug Repositories. Many open-source projects maintain public bug tracking systems, which can be used to identify bugs found "in the wild." These repositories encourage the development of practically useful tools and to evaluate research tools on real-world bugs. While many HPC projects are open source, the use of bug-tracking systems needs to be promoted among the DOE application developers. Going beyond bug-tracking systems, developing guidelines for sharing bugs and code snippets to reproduce them can accelerate the development of tools that can handle HPC-specific correctness challenges. Ideally, tools can help automatically mine bug repositories to isolate bugs from other sources of errors (software configuration, user errors, etc.), validate the bugs, and extract relevant code harness from the patches used to close a bug. Even the study of job failures on HPC clusters and the reasons for such failures (e.g., [130]) would be valuable for the community to compile and share.

5 Next steps

5.1 ASCR Focus Areas

There are many opportunities for return on investment in HPC correctness. These returns and investments would fall into the short, medium and long term time scales.

5.1.1 Short Term

Advances for production use. Investment focused on supporting the current efforts to program and perform computational science on leadership computing facilities. Such effort would apply best-of-breed existing tools (commercial as well as those being researched by the community), extend those tools, and generally work with HPC applications code as-is. Such work would encompass extending the existing HPC tools infrastructure (debuggers, compilers, etc.) with features for larger scale verification and debugging in the more complex HPC contexts being encountered today.

> We propose many directions broken down into short, medium and long-term components. We also propose an agenda for a correctness workshop as well as a few "moonshot" projects that can bring in added creativity through added time and resource pressures.

Importing successful ideas from non-HPC domains. This effort would focus on bringing the tools and technologies currently being developed to prove correctness and safety properties of non-HPC code, to the HPC community. This would bring software currently being used to formally prove hardware, security, safety, and performance—as applicable—to HPC. Research in the formal verification of cyber-physical systems could be applied to the verification of simulated physical systems. Research in embedded computing - e.g. verification of controls, and verified hardware, e.g. flight safety, could

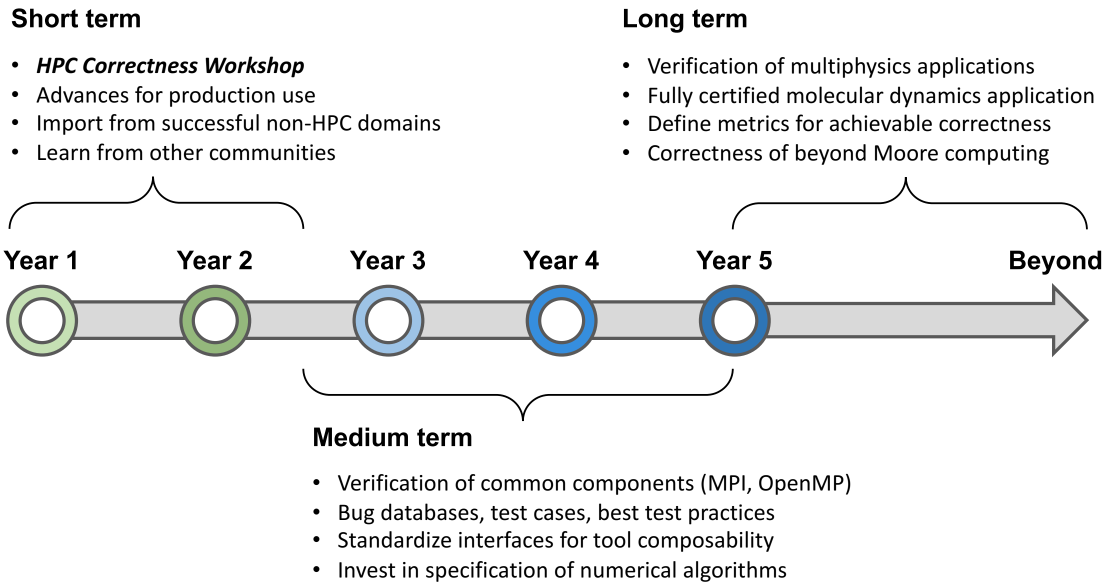
Short term
• *HPC Correctness Workshop*
• Advances for production use
• Import from successful non-HPC domains
• Learn from other communities
Long term
• Verification of multiphysics applications
• Fully certified molecular dynamics application
• Define metrics for achievable correctness
• Correctness of beyond Moore computing
Year 1
Year 2
Year 3
Year 4
Year 5
Beyond
Medium term
• Verification of common components (MPI, OpenMP)
• Bug databases, test cases, best test practices
• Standardize interfaces for tool composability
• Invest in specification of numerical algorithms

Figure 2: Short, medium, and long term goals identified to advance the field of correctness in HPC

be applied to verifying the controls for current HPC systems. Certification techniques for large scale distributed systems (e.g., Amazon, DropBox) could be brought to HPC for reasoning about large scale parallel computing systems (and HPC file systems).

5.1.2 Medium Term

Community-wide impact: Medium term focus must target achieving community-wide impact through projects that take on the challenges of verifying critical software components such as widely-used runtime systems (e.g., OpenMP and MPI implementations) and math libraries. There must also be significant emphasis placed on the collection of bug incidence reports, as well as search for past solutions that detail how these bugs were fixed. Finally, tool interfaces and runtime event collection mechanisms must be standardized to support tool composition.

Ontologies of mathematics / algorithms. This effort would focus on developing specific infrastructure for verification and debugging unique to HPC. In particular, investments in the modeling and specification of numerical algorithms, ontologies for the mathematics underlying such algorithms, and reasoning about statistical and randomized systems, would be advanced. There would likely be a close relationship between these efforts and Uncertainty Quantification and Automatic Differentiation, since such efforts are aimed at giving confidence in the results of scientific simulations. Research would be aimed at tools for automated reasoning and verification about topics in Applied Mathematics of relevance to ASCR, in particular numerical methods for scalable solving of PDEs, discretization, optimization, multiscale computing, and multi-physics. The effort would be directed toward formalizing mathematics otherwise expressed in "prose" in mathematical research papers. The effort would try to parallel the development seen in the computer systems research community, where new systems software research papers (operating systems, file systems, etc.) are now expected to come with the formal expression of the algorithm and the proof of its correctness. Such an effort would nudge and enable the engineering of these new mathematical advances to be done with the most modern software engineering practices, with the modularity, specifications, and proofs needed to achieve correct by construction HPC systems.

5.1.3 Long Term

Beyond Moore. Focus on Beyond Moore computing would clearly be an appropriate long term focus. In particular, ensuring correctness of new computing paradigms such as neuromorphic, probabilistic, and quantum computation. Application areas such as machine learning would also be a focus of the correctness research. Machine learning in particular seems to offer opportunities for new advances in correctness, particularly developing systems for proving that a ML system will work within bounds, and for explaining the conclusions or decision made by a machine learning system.

"Moonshot" projects. Longer term investments would be achieved by moonshot projects that seek to build end-to-end, demonstrable successes, with both immediate benefits and which also become the basis for advances in the field. A few potential moonshot projects are described in §5.2.

5.2 Moonshots

Well-chosen "moonshot" challenges can help increase the pace of progress and demonstrate what is possible. While these projects are ambitious, they are probably feasible with just a few years of focused effort. Consider that the first sequencing of a human genome took about ten years and billions of dollars, but now such sequencing is routinely practiced within a few hours in a doctor's office lab for a few hundred dollars. Once feasibility is established in the projects below, the engineering of tools to reduce costs and speed the results will rapidly advance. For verification in particular, consider that proof libraries and tools are cumulative, and can lead to building of capabilities. The first efforts to formalize floating point took years to achieve, but the formal specifications for floating point are now available and can be downloaded for free.

5.2.1 Project 1: Fully certified molecular dynamics simulation

Molecular dynamics (MD) packages such as Desmond [16] would serve as excellent "moonshot" projects. The tools and science of projects such as Deep Specification [94] could be applied, extending them to the special tuned number representations and operators of DESMOND, through bond models and approximations, and through the compiler, systems software, and runtime. One would certify the property of bit-reversibility through to the implementation. The components of such a project (e.g., a fully certified concurrent dynamic runtime task scheduler, certified parallel 3D FFT) could then be used in other projects.

Longer term, this technique could be used to provide certifications and verification that a long running numerical simulation running on a special purpose scientific computer *proves* a scientific result [117]. This would not just be an academic exercise; vital missions of the Department of Energy (e.g., the NNSA) depend on simulation on complex, custom constructed high performance computers to assure the safety and performance of our nation's strategic nuclear arsenal. Advances in NNSA simulations of kinetic plasma on the special Roadrunner supercomputer [15] used optimizations and coding techniques closely relate to those of DESMOND. The results of such moonshot projects could immediately carry over to kinetic plasma simulations of the type that runs on Roadrunner. Furthermore, such a certification system could be used to assess the implications of new hardware architectures (e.g., network communications protocols), representations (reducing the precision of values) and algorithms (e.g., communication avoiding or sparse high dimensional FFT) on future high performance kinetic plasma simulations.

5.2.2 Project 2: Multiphysics

Multiphysics software systems—simulations that consists of more than one component governed by its own principles—are used in many large-scale physical simulations. An impactful project would be full verification of important logic and numerical properties (e.g., energy conservation properties or others) underlying multiphysics applications. This exercise can force the examination of how individual subsystem guarantees help meet whole-system correctness goals. Some examples of HPC multiphysics software infrastructures that could be targeted are Chombo [34], PETSc [6], SUNDIALS [55], Trilinos [53], and Uintah [46].

5.2.3 Project 3: Verified Compiler/Runtime Components

Verifying some of the key software infrastructural components of an HPC system can bootstrap the development of rigorous methods that help harden support for large-scale runs of HPC simulations. Of special interest would be the verification of the MPI library, going by the MPI 4.0 standard, tracing through various device layers and ending in optimized infrastructural code that supports rapid messaging using lock-free programming methods. Similarly, verifying the polyhedral compilation toolchain and linear algebra libraries can ensure correctness of large and highly reused code bases.

5.3 HPC correctness workshop

The exercise of bringing this limited set of report authors together for sharing ideas has resulted in good cross fertilization of ideas for HPC correctness: making us aware of useful tools for our own research in HPC, and some of the larger challenges. But with the breadth of the problem, and the richness of the verification and debugging community outside, HPC, more is needed.

Advancement in this area of correctness could be facilitated by a workshop on HPC correctness that could bring together the larger community of experts on correctness techniques and tools with DOE stakeholders, especially the developers of DOE HPC applications.

Such a workshop would provide an opportunity for HPC software developers to communicate current practices and discuss the primary obstacles to achieving correctness in HPC software development and for correctness experts to identify promising research directions that offer the potential to overcoming these obstacles. A two day workshop comprising a small number of invited presentations, 5–10 minute presentations based on 2-page position papers solicited from the community, and 3–4 1-hour round table discussions to stimulate a dialogue between the correctness experts and stakeholders is recommended. We anticipate that this dialogue will reinforce many of the summit findings, possibly identify additional research opportunities, and help prioritize future research directions.

5.4 Competitions for verification of HPC software

In recent years, several verification competitions have evolved within the general software verification community. For example, the annual SV-COMP competition, currently in its seventh year [1], is a fully automated competition in which participants submit tools which are all fed a long series of C programs with corresponding properties that are expected to hold or fail. The VerifyThis competition [122], in its sixth year, is an interactive competition in which participants are given a set of problems which they are expected to implement and verify over the course of a day using any tools they desire. These competitions have had several beneficial impacts: they provide objective and consistent comparisons of tools, they provide a recognized measure of the current state-of-the-art, and they have created large verification benchmark suites that are widely used beyond the competition itself.

As discussed above, HPC software has many specific characteristics, and these are not covered in the existing general-purpose software competitions. Therefore, a HPC-specific verification competition could be held, say, during the course of one day at SC17 with multi-agency sponsorship. Similar to VerifyThis, participants could be given a set of programs of increasing complexity, together with written specifications of expected behavior, and asked to formally specify and verify as much as they can, using any tools they desire. A panel of judges would examine and evaluate the results. Participants could also be given an opportunity to present their solutions.

References

[1] Annual SV-COMP competition. https://sv-comp.sosy-lab.org/2017/.

[2] Arnold, D. C., Ahn, D. H., De Supinski, B. R., Lee, G. L., Miller, B. P., and Schulz, M. Stack trace analysis for large scale debugging. In *IEEE International Parallel and Distributed Processing Symposium* (2007), IEEE, pp. 1–10.

[3] Bamboo. https://www.atlassian.com/software/bamboo.

[4] Automated testing system (ATS). http://computation.llnl.gov/research/mission-support/WCI/automated-testing-system.

[5] Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D. H., Laguna, I., Schulz, M., Lee, G. L., Protze, J., and Müller, M. S. ARCHER: effectively spotting data races in large OpenMP applications. In *2016 IEEE International Parallel and Distributed Processing Symposium,* (2016), IEEE, pp. 53–62.

[6] Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. Petsc. *See http://www.mcs.anl.gov/petsc* (2001).

[7] Bao, W., Krishnamoorthy, S., Pouchet, L., Rastello, F., and Sadayappan, P. Polycheck: dynamic verification of iteration space transformations on affine programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016* (2016), pp. 539–554.

[8] Barthe, G., Espitau, T., marco Gaboardi, Gregoire, B., Hsu, J., and Strub, P.-Y. Formal certification of randomized algorithms.

[9] Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., Krishnan, S., chung Lam, C., Lu, Q., Nooijen, M., Pitzer, R. M., Ramanujam, J., Sadayappan, P., and Sibiryakov, A. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE 93*, 2 (2005), 276–292.

[10] Bertot, Y., and Castéran, P. *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[11] Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., and Weis, P. Wave equation numerical resolution: a comprehensive mechanized proof of a c program. *Journal of Automated Reasoning 50*, 4 (2013), 423–456.

[12] Boldo, S., and Melquiond, G. Flocq: A unified library for proving floating-point algorithms in coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on* (2011), IEEE, pp. 243–252.

[13] Boost test library. http://www.boost.org.

[14] Bouteiller, A., Bosilca, G., and Dongarra, J. Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting* (2007), Springer, pp. 297–306.

[15] Bowers, K. J., Albright, B. J., Yin, L., Daughton, W., Roytershteyn, V., Bergen, B., and Kwan, T. Advances in petascale kinetic plasma simulation with vpic and roadrunner. In *Journal of Physics: Conference Series* (2009), IOP Publishing.

[16] Bowers, K. J., Chow, E., Xu, H., Dror, R. O., Eastwood, M. P., Gregersen, B. A., Klepeis, J. L., Kolossvary, I., Moraes, M. A., Sacerdoti, F. D., et al. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006), ACM, p. 84.

[17] Bronevetsky, G., Laguna, I., Bagchi, S., de Supinski, B. R., Ahn, D. H., and Schulz, M. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2010), IEEE, pp. 231–240.

[18] Bronevetsky, G., Laguna, I., de Supinski, B. R., and Bagchi, S. Automatic fault characterization via abnormality-enhanced classification. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* (June 2012), pp. 1–12.

[19] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010* (2010), J. C. Hoe and V. S. Adve, Eds., ACM, pp. 167–178.

[20] Burnette, E. *Eclipse IDE Pocket Guide*. O'Reilly Media, Inc., 2005.

[21] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation* (2008).

[22] Cadar, C., and Sen, K. Symbolic execution for software testing: Three decades later. *Communications of the ACM 56*, 2 (February 2013), 82–90.

[23] Certikos: Certified Kit Operating System. http://flint.cs.yale.edu/certikos/.

[24] Chakraborty, S., and Vafeiadis, V. Validating optimizations of concurrent c/c++ programs. In *Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on* (2016), IEEE, pp. 216–226.

[25] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 18–37.

[26] Chen, Z., Gao, Q., Zhang, W., and Qin, F. FlowChecker: Detecting bugs in MPI libraries via message flow checking. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010), IEEE, pp. 1–11.

[27] Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., and Rakamaric, Z. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), pp. 300–315.

[28] Chong, N., Donaldson, A. F., Kelly, P. H. J., Ketema, J., and Qadeer, S. Barrier invariants: a shared state abstraction for the analysis of data-dependent gpu kernels. In *28th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)* (10 2013), pp. 605–622.

[29] Clang static analyzer. https://clang-analyzer.llvm.org/.

[30] Clang sanitizers. https://clang.llvm.org/docs/.

[31] Threadsanitizer data-race detector. https://clang.llvm.org/docs/ThreadSanitizer.html.

[32] Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, Cambridge, 1999.

[33] CLion. https://www.jetbrains.com/clion.

[34] Colella, P., Graves, D., Ligocki, T., Martin, D., Modiano, D., Serafini, D., and Van Straalen, B. Chombo software package for amr applications-design document, 2000.

[35] The Coq Proof Assistant. https://coq.inria.fr/.

[36] Nvidia CUDA-MEMCHECK correctness checking tool. http://docs.nvidia.com/cuda/cuda-memcheck/.

[37] DDT debugger. https://www.allinea.com/products/ddt.

[38] de Alfaro, L., and Henzinger, T. A. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001* (2001), pp. 109–120.

[39] DeRose, L., Gontarek, A., Vose, A., Moench, R., Abramson, D., Dinh, M. N., and Jin, C. Relative debugging for a highly parallel hybrid computer system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 63:1–63:12.

[40] Ding, C., and Mondet, S. A curated list of awesome coq frameworks, libraries and software. https://github.com/uhub/awesome-coq/blob/master/README.md.

[41] The exascale computing project. https://exascaleproject.org/exascale-computing-project/.

[42] Edwards, H. C., Trott, C. R., and Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput. 74*, 12 (2014), 3202–3216.

[43] Floating point litmus testing (flit) framework. www.cs.utah.edu/fv/FLiT/Analysis.html.

[44] Gao, Q., Qin, F., and Panda, D. K. DMTracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (2007), ACM, p. 15.

[45] Gao, S., Kong, S., and Clarke, E. M. dReal: An SMT solver for nonlinear theories over the reals. In *Proceedings of the 24th International Conference on Automated Deduction (CADE)* (2013), pp. 208–214.

[46] Germain, J. D. d. S., McCorquodale, J., Parker, S. G., and Johnson, C. R. Uintah: A massively parallel problem solving environment. In *The Ninth International Symposium on High-Performance Distributed Computing, 2000. Proceedings.* (2000), IEEE, pp. 33–41.

[47] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (2005), pp. 213–223.

[48] Godefroid, P., Levin, M. Y., and Molnar, D. A. SAGE: whitebox fuzzing for security testing. *Commun. ACM 55*, 3 (2012), 40–44.

[49] Google Tests. https://github.com/google/googletest.

[50] Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X. N., Weng, S.-C., Zhang, H., and Guo, Y. Deep specifications and certified abstraction layers. In *ACM SIGPLAN Notices* (2015), ACM.

[51] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. Certikos: an extensible architecture for building

certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[52] Hardesty, L. Dude, wheres my code? http://news.mit.edu/2013/system-flags-useful-code-compilers-might-discard-1016.

[53] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS) 31*, 3 (2005), 397–423.

[54] Hilbrich, T., Schulz, M., de Supinski, B. R., and Müller, M. S. MUST: A scalable approach to runtime error detection in MPI programs. In *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[55] Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS) 31*, 3 (2005), 363–396.

[56] Hirata, S. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A 107*, 46 (2003), 9887–9897.

[57] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM* (1969), 576–580.

[58] Holzmann, G. J. The model checker spin. *IEEE Transactions on software engineering 23*, 5 (1997), 279–295.

[59] Hornung, R. D., and Keasler, J. A. The raja portability layer: Overview and status. Tech. rep., 2014.

[60] Hovy, C., and Kunkel, J. Towards automatic and flexible unit test generation for legacy hpc code. In *Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering* (2016), IEEE Press, pp. 42–49.

[61] Hudak, P. Domain-specific languages. *Handbook of Programming Languages 3*, 39-60 (1997), 21.

[62] Hughes, J., Pierce, B. C., Arts, T., and Norell, U. Mysteries of Dropbox: Property-based testing of a distributed synchronization service. In *International Conference on Software Testing, Verification and Validation (ICST)* (Apr. 2016).

[63] Hwu, W.-M. What is driving heterogeneity in hpc? https://bluewaters.ncsa.illinois.edu/documents/10157/169216/hwu_heterogeneity.pdf, 2016.

[64] Insure++. https://www.parasoft.com/product/insure/.

[65] Intel inspector. https://software.intel.com/en-us/intel-inspector-xe.

[66] IntelliJ IDEA. https://www.jetbrains.com/idea.

[67] Jiang, N., Kim, J., and Dally, W. J. Indirect adaptive routing on large scale interconnection networks. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 220–231.

[68] Joshi, P., Naik, M., Park, C.-S., and Sen, K. An extensible active testing framework for concurrent programs. In *21st International Conference on Computer Aided Verification (CAV'09)* (2009), vol. 5643 of *Lecture Notes in Computer Science*, Springer, pp. 675–681.

[69] Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodova, A., Taylor, C., Frolov, V., Reeber, E., and Naik, A. Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation. In *CAV* (2009), pp. 414–429.

[70] Kamil, S., Cheung, A., Itzhaky, S., and Solar-Lezama, A. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, ACM, pp. 711–726.

[71] Kanewala, U. https://www.cs.montana.edu/upulee.kanewala/research.html. Application of Metamorphic Testing to HPC: Some examples.

[72] Kelly, D., and Sanders, R. The challenge of testing scientific software. *CAST 2008: Beyond the Boundaries* (2008), 30.

[73] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[74] KLOCKWORK Static Analyzer. http://www.klocwork.com/.

[75] Laguna, I., Ahn, D. H., de Supinski, B. R., Bagchi, S., and Gamblin, T. Probabilistic Diagnosis of Performance Faults in Large-scale Parallel Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), PACT '12, ACM, pp. 213–222.

[76] Laguna, I., Ahn, D. H., de Supinski, B. R., Gamblin, T., Lee, G. L., Schulz, M., Bagchi, S., Kulkarni, M., Zhou, B., Chen, Z., and Qin, F. Debugging high-performance computing applications at massive scales. *Communications of the ACM 58*, 9 (Aug. 2015), 72–81.

[77] Laguna, I., and Schulz, M. Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE Press, p. 19.

[78] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[79] Leroy, X., et al. The compcert verified compiler. *Development available at http://compcert. inria. fr 2009* (2004).

[80] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., and Rajan, S. P. Gklee: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 215–224.

[81] Liu, T., Curtsinger, C., and Berger, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 327–336.

[82] Meister, B., Vasilache, N., Wohlford, D., Baskaran, M., Leung, A., and Lethin, R. R-stream compiler. In *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011, pp. 1756–1765.

[83] Mendis, C., Bosboom, J., Wu, K., Kamil, S., Ragan-Kelley, J., Paris, S., Zhao, Q., and Amarasinghe, S. P. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), pp. 391–402.

[84] Meng, Q., Humphrey, A., Schmidt, J., and Berzins, M. Preliminary experiences with the uintah framework on intel xeon phi and stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE)* (2013), pp. 48:1–48:8.

[85] Menzies, T. Understanding software: Recent lessons from empirical software engineering. https://figshare.com/articles/Understanding_Software_Recent_Lessons_from_Empirical_Software_Engineering/4680961, 2017. Talk presented at the SI2 PI meeting, Arlington, February 21.

[86] Messina, P., and Lee, S. Exascale computing project – software. https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/201704/ECP_Update_ASCAC__20170419.pdf, 2017. ASCAC Meeting, Arlington, VA.

[87] Mirgorodskiy, A. V., Maruyama, N., and Miller, B. P. Problem diagnosis in large-scale computing environments. In *Proceedings of the ACM/IEEE SC 2006 Conference* (Nov 2006), pp. 11–11.

[88] Mitra, S., Laguna, I., Ahn, D. H., Bagchi, S., Schulz, M., and Gamblin, T. Accurate Application Progress Analysis for Large-scale Parallel Debugging. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 193–203.

[89] Murthy, K., Paul, S. R., Meel, K. S., Cogumbreiro, T., and Mellor-Crummey, J. Design and verification of distributed phasers. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 405–418.

[90] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[91] Park, C.-S., Sen, K., Hargrove, P., and Iancu, C. Efficient data race detection for distributed memory parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011), ACM, p. 51.

[92] Park, C.-S., Sen, K., and Iancu, C. Scaling data race detection for partitioned global address space programs. In *27th International Conference on Supercomputing (ICS'13)* (2013), ACM, pp. 47–58.

[93] Park, C.-S., Sen, K., and Iancu, C. Scaling data race detection for partitioned global address space programs (short paper). In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), ACM, pp. 305–306.

[94] Pierce, B. C. The science of deep specification, Nov. 2016. Invited keynote at *SPLASH / OOPSLA*.

[95] Puschel, M., Moura, J. M., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE 93*, 2 (2005), 232–275.

[96] Qian, X., Sen, K., Hargrove, P., and Iancu, C. Opr: Deterministic group replay for one-sided communication. In *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2016), ACM. Poster Paper.

[97] Qian, X., Sen, K., Hargrove, P., and Iancu, C. Sreplay: Deterministic group replay for one-sided communication. In *30th International Conference on Supercomputing (ICS'16)* (2016), ACM.

[98] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. P. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013* (2013), pp. 519–530.

[99] Ramananandro, T., Mountcastle, P., Meister, B., and Lethin, R. A unified Coq framework for verifying C programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (2016), ACM, pp. 15–26.

[100] Rubio-Gonzalez, C., Nguyen, C., Mehne, B., Sen, K., Demmel, J., Kahan, W., Iancu, C., Lavrijsen, W., Bailey, D. H., and Hough, D. Floating-point precision tuning using blame analysis. In *38th International Conference on Software Engineering (ICSE'16)* (2016), IEEE.

[101] Rubio-Gonzalez, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonius: Tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)* (November 2013), ACM.

[102] Sato, K., Ahn, D. H., Laguna, I., Lee, G. L., and Schulz, M. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), ACM, p. 62.

[103] Sato, K., Ahn, D. H., Laguna, I., Lee, G. L., Schulz, M., and Chambreau, C. M. Noise Injection Techniques to Expose Subtle and Unintended Message Races. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), ACM, pp. 89–101.

[104] Schordan, M., Lin, P.-H., Quinlan, D., and Pouchet, L.-N. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2014), Springer, pp. 493–508.

[105] Sen, K. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)* (2008), ACM, pp. 11–21.

[106] Sen, K., and Agha, G. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)* (2006), vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 419–423.

[107] Sen, K., and Agha, G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)* (2006), vol. 4383 of *Lecture Notes in Computer Science*, Springer, pp. 166–182.

[108] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (2005), ACM, pp. 263–272. ACM SIGSOFT Distinguished Paper Award.

[109] Sharma, R., Bauer, M., and Aiken, A. Verification of producer-consumer synchronization in GPU programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), pp. 88–98.

[110] Siegel, S. F. Model checking nonblocking MPI programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation* (2007), Springer, pp. 44–58.

[111] Siegel, S. F., and Zirkel, T. K. Collective assertions. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings* (2011), pp. 387–402.

[112] Siegel, S. F., and Zirkel, T. K. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science 5*, 4 (2011), 395–426.

[113] Solovyev, A., Jacobsen, C., Rakamarić, Z., and Gopalakrishnan, G. Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In *Proceedings of the 20th International Symposium on Formal Methods Formal (FM)* (2015), pp. 532–550.

[114] Spielman, D. A., and Teng, S.-H. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing* (2004), ACM, pp. 81–90.

[115] Stewart, G., Beringer, L., Cuellar, S., and Appel, A. W. Compositional compcert. In *Proceedings of the 42Nd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA, 2015), POPL '15, ACM, pp. 275–287.

[116] Stng: Automatically leverage gpus for your high-performance computation with verified lifting. http://stng.uwplse.org/.

[117] Sussman, G. J., and Wisdom, J. Chaotic evolution of the solar system. Tech. rep., DTIC Document, 1992.

[118] Totalview debugger. http://www.roguewave.com/products-services/totalview.

[119] UPC Thrille. http://upc.lbl.gov/thrille.shtml.

[120] Vakkalanka, S. S., Sharma, S., Gopalakrishnan, G., and Kirby, R. M. ISP: a tool for model checking MPI programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (2008), ACM, pp. 285–286.

[121] Valgrind instrumentation framework. http://valgrind.org/.

[122] VerifyThis competition. http://www.verifythis.org.

[123] Vetter, J. S., and De Supinski, B. R. Dynamic software testing of MPI applications with Umpire. In *ACM/IEEE 2000 Conference of Supercomputing* (2000), IEEE, pp. 51–51.

[124] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., De Supinski, B. R., Schulz, M., and Bronevetsky, G. A scalable and distributed dynamic formal verifier for MPI programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for* (2010), IEEE, pp. 1–10.

[125] Wang, X., Zeldovich, N., Kaashoek, M. F., and Solar-Lezama, A. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 260–275.

[126] Weng, S.-C. *DeepSpec: Modular Certified Programming with Deep Specifications*. PhD thesis, New Haven CT, 2016.

[127] Wickerson, J., Batty, M., Sorensen, T., and Constantinides, G. A. Automatically comparing memory consistency models. In

Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017 (2017), pp. 190–204.

[128] Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., and Voelker, G. MPIWiz: Subgroup Reproducible Replay of Mpi Applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 251–260.

[129] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and Understanding Bugs in C Compilers. In *Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)* (June 2011).

[130] Yuan, Y., Wu, Y., Wang, Q., Yang, G., and Zheng, W. *Computers & Mathematics with Applications 63* (Jan. 2012), 365377.

[131] Zheng, M., Ravi, V. T., Qin, F., and Agrawal, G. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems 25*, 1 (2014), 104–115.

[132] Zheng, M., Rogers, M. S., Luo, Z., Dwyer, M. B., and Siegel, S. F. Civl: Formal verification of parallel programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2015), pp. 830–835.

[133] Zhou, B., Kulkarni, M., and Bagchi, S. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 85–96.

[134] Zhou, B., Too, J., Kulkarni, M., and Bagchi, S. WuKong: automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing* (2013), ACM, pp. 131–142.