Top Ten Ways to Make Formal Methods for HPC Practical

Ganesh Gopalakrishnan and Robert M. Kirby
School of Computing, University of Utah, Salt Lake City, UT 84112
http://www.cs.utah.edu/fv – {ganesh,kirby}@cs.utah.edu

ABSTRACT

Almost all fundamental advances in science and engineering crucially depend on the availability of extremely capable high performance computing (HPC) systems. Future HPC systems will increasingly be based on heterogeneous multicore CPUs, and their programming will involve multiple concurrency models. These developments can make concurrent programming and optimization of HPC platforms and applications very error-prone. Therefore, significant advances must occur in verification methods for HPC. We present ten important formal methods research thrusts that can accelerate these advances.

1. FORMAL METHODS AND HPC

High performance computing (HPC) is one of the pillars supporting virtually all of science and engineering. For the long term viability of this area, it is absolutely essential that we have HPC platforms that are easy to program and come with incisive verification tools that help application developers gain confidence in their software. Unfortunately, the current situation is far from these ideals. In § 2, we propose ten research thrusts that are essential to *avert a debugging crisis in HPC*, and substantiate our remarks. In the remainder of this section, we describe the context and our motivations.

The FV and HPC Communities Ignoring Each Other: A significant number of HPC applications are written using the Message Passing Interface (MPI, [1]) and are run on message passing distributed memory machines. MPI is the *lingua franca* of parallel computing in HPC. Its amazing popularity is attributable to its thoughtful design that involved a collaboration between machine vendors and application developers. The designers of MPI made sure that MPI programs will run well on a huge variety of platforms and also that a variety of parallel applications can be efficiently coded up in it. However, MPI programming is very error prone because every efficiency-oriented MPI construct can also be easily misused, resulting in nasty code level bugs such as deadlocks and resource leaks. Barring a few exceptions [2], debugging challenges associated with MPI programming have not been very much discussed in the formal verification literature. The primary debugging approach for MPI is still one of running an MPI application on a specific platform, feeding it a collection of test inputs, and seeing if anything goes wrong. This approach may seem to work — largely due to the uniformity of the platform hardware, libraries, and programming styles employed in this area. By avoiding aggressive styles of MPI programming and staying within safe practices, today's HPC application developers can often compensate for the lack of rigorous testing and reasoning approaches.

At present, HPC application developers and the Computer Science research community are relatively unaware of what is going on each others' research areas. HPC application writers can simply learn MPI and a few programming languages, then spend most of their time focusing on their application-area sciences (such as Physics and Chemistry). For them, even the word "verification" often means *does one's algorithm and its encoding match Physics?* — *i.e,* code correctness is often not the primary question. The "CS side" also goes about developing formal methods for programs more familiar to it – such as device drivers, flight software, and floating point hardware. Bugs faced by HPC platform and application developers are often attributed to the use of baroque libraries such as MPI and languages like C and Fortran. People expressing these views seem not to realize that the large size of libraries such as MPI is due to its careful design to run on a huge variety of platforms and applications. While formal methods for verifying unmodified concurrent programs were being demonstrated in the late 90s (*e.g.,* Verisoft [3] by Godefroid), none of these ideas have influenced MPI program verification tools.

The consequences of this isolation are quite shocking. We once encountered a bug that got triggered only when our program was configured for exactly 128 processes – not on lower or higher process counts. Using crude debugging methods, we finally traced this bug to the MPI library. In another instance, we applied decades old model checking methods to a published locking protocol and unearthed two serious flaws [4]. Other MPI programmers have similar (or worse) incidents to report. Considering that Petascale machines being built all around the world will consume about a million dollars worth of electricity a year – not to mention other costs (personnel, opportunity costs of delayed simulations) that are even higher – the detrimental effects of these *ad hoc* debugging methods become all too apparent.

SDP Workshop 2010, Santa Fe, NM

Neglected Collaboration Coming Home to Roost: A sea change awaits HPC system designers, application developers, and CS researchers. There is growing demand for HPC capabilities coming from every conceivable scientist and engineer: *e.g.,* biologists conducting cellular level studies, engineers designing new aircraft, and physicists testing new theories. Future HPC systems will be required to deliver amazing computational (FLOP) rates at low energy-delay product values never before attained. They have to do so on multi-core platforms that have to be programmed properly in order to deliver higher overall performance. Unfortunately these platforms will be comprised of *highly* heterogeneous ensembles of CPUs and GPUs which will demand inordinate amounts of programming effort. HPC application developers – once happy in their "MPI world" and focused on their area sciences – will now have to employ combinations of libraries including OpenMP, MPI, and CUDA/OpenCL [5, 6]. Unfortunately, there are no good methodologies that help guide hybrid programming of this nature.

If CS researchers had any excuses to ignore HPC, even these must now prove to be baseless because HPC has crept into mainstream CS through a growing list of application areas and devices such as computer games, iPhones with GPUs, and desktop supercomputers. CS researchers cannot declare that "ad hoc combinations of MPI, OpenMP, and OpenCL" must cease to exist. These well proven APIs with a large user base and proven efficacy will continue to be used in various combinations. The severe weaknesses associated with the conventional "run and see" kind of testing can no longer be covered up through safe practices because of the large variety of APIs in vogue. All the missed dialog between HPC researchers and CS researchers must therefore occur within a short span of time. All the coding bugs that were quietly patched up during HPC projects must now become parts of bug databases and challenge problems to help bootstrap the necessary formal verification enterprise.

Basis for Our Position Statement: We embarked on applying FV methods to HPC problems about five years ago, and to our pleasant surprise, we were met with warm reception! We found that HPC researchers were eager to help us demonstrate that new algorithms designed using formal methods can be correct, and also provide insights to outperform *ad hoc* algorithms [4]. Based on our experience, we can say that there are still much 'low-hanging fruits' to be picked in this area. We have shown in [7] that the core semantics of complex APIs such as MPI can be formalized in sufficient detail. In [8, 9], we report on our dynamic verifier for MPI called ISP and show that specialized dynamic partial order reduction algorithms for MPI can scale well and help locate bugs in very large realistic MPI programs.

There are a few others who pioneered work in applying FV to HPC problems even before us (*e.g.,* [2]). Yet, in the grand scheme of things, *there is an absolutely alarming shortage of researchers interested in FV and HPC.* These facts are painfully apparent in many ways. For example, there are hardly any papers on formal methods for HPC problems in today's leading FV conferences; the vast majority of papers are confined to trendy topics such as *Java Concurrency* or *Transactional Memories*). Also, FV researchers wanting to publish in leading HPC conferences find very few conferences set up to handle technically deep presentations. Clearly, we need to do far better if we are to urgently build a community of FV and HPC researchers who collaborate.

Based on our experience, we can now summarize *ten important thrusts* in § 2.1 through § 2.10 that, if not undertaken urgently, will prove detrimental to growth in HPC. In each section, we present evidence to back up our statements through our recent projects.

2. OUR POSITION STATEMENTS

2.1 Support FV around well-established APIs

Formal methods researchers tend to prefer simple parsimonious APIs, dismissing well established APIs such as MPI and OpenMP as overly rich and poorly designed (*i.e.*, "hairy"). The real danger of going this route are several. First, it would be nearly impossible to find large meaningful examples that employ these minimalist APIs. Without demonstrating FV results on large problems, the HPC community will not be swayed in favor of FV methods. They may also get thoroughly disenchanted about the reach of FV methods and the unhelpful disposition of FV researchers.

Second, dealing with real APIs such as MPI actually enriches formal verification methodologies and algorithms. Modern APIs such as the multicore communications API (MCAPI, [10]) have, for instance, begun to incorporate many of the same constructs pertaining to message passing. *True innovation is almost always sparked by complexity born out of genuine need than artificially mandated parsimony.*

Justification: We started our work on formalizing MPI unsure of how far we could progress. Over a progression of versions, we managed to produce a comprehensive formal semantics for 150 of the over 300 MPI-2 functions [7] in TLA+. This semantics proved useful for putative query answering (users asking questions about scenarios of usage of the API [11]) but not to design a dynamic verification algorithm. Our breakthrough came when we understood how to formulate the *happens-before* relation underlying MPI. ISP and its entire analysis – including how it models concurrency/resource interactions is based on this happens-before model. Recently, we have been able to apply these lessons while building a verifier similar to ISP for MCAPI [10].

A strong caveat while programming in a multi-API world is that designers will do anything that is not explicitly prohibited. Recently, a group reported to us of a mysterious "hang" that they experienced when they invoked kernels on multiple GPUs using OpenMP threads. It was lucky that they encountered this problem; it could have been worse if they found things "seemingly working," only to hand a failure to someone else who later ports the code. These facts strongly suggest the need for modeling well-established APIs as well as their interactions through formal methods.

2.2 Devise point solutions

While there have been some attempts at standardizing formal verification tools around common intermediate forms (*e.g.,* tools such as CHESS [13] can verify all programs that bind to the .NET API), in general we believe that point tools are necessary for different APIs. Codes involving some APIs are best modeled and handled using symbolic methods (*e.g.,* our PUG tool [14] for CUDA) while others are best handled using dynamic verification methods and specialized search methods (*e.g.,* the ISP tool). Even for dynamic verification tools, one has to engineer different dynamic partial order reduction methods to realize a significant amount of interleaving reduction.

Justification: We have recently contributed "PUG," the first comprehensive formal analysis tool for CUDA based on SMT solving [14]. The approach taken in PUG (symbolic analysis) and that taken in ISP (dynamic reduction based verification) are examples of different solutions working well for different APIs. Likewise, dynamic interleaving reduction algorithms that best handle MPI and those that best handle shared memory threads (*e.g.,* our tool Inspect [15])are quite a bit different. The efficiency gains due to point-solutions that handle particular APIs well are well worth the effort of developing separate tools.

2.3 Design API implementations supporting FV

Tools such as Verisoft, CHESS, and ISP must be able to transfer control to the verification scheduler when API calls are invoked, examine the execution history thus far, and compute as well as enforce essential interleavings that have not been manifested thus far. Unfortunately, many API designs and implementations make these steps extraordinarily difficult. While many high-efficiency OpenMP implementations are being developed, few of them provide hooks to control the underlying threads/tasks created by the runtime. As other examples, we had to employ many complex (and error-prone) techniques such as dynamic API call rewriting [16] or putting in *probe loops* [10] in our past work. These steps could easily have been avoided by exporting a few more *verification oriented calls* in these APIs. Dynamic verification tools built using these extra API calls will also prove to be more portable across machines. Another very important API-related fact is that *it is insufficient to have just one API implementation.* From the point of view of FV tool developers, high performance API implementations seldom offer support for the construction of error monitors because these implementations do not maintain very much state history. Also these implementations perform very few (if any) runtime error checks, thus allowing illegal API call arguments to go undetected. These can easily lead to inconsistent executions. From the point of view of practitioners, an FV oriented API alone is insufficient because of its overheads.
Justification: Taking MPI non-deterministic receives as an example, there are no MPI runtimes that help exercise control over the MPI sends that these receive commands can match. By providing extra arguments to an MPI wildcard receive, it would become possible to influence these decisions. Also, building complementary API implementations is an important requirement: a high performance API for final production use, and a verification-oriented API based on the formal API semantics for use during verification.

2.4 Avoid cross-API interleaving product

While traditional methods such as dynamic partial order reduction have helped conquer complexity by analyzing only *representative* interleavings, it is not clear how these ideas will transfer over to programs that employ *multiple concurrency APIs.* For example, if OpenMP code blocks and MPI code blocks are used in a program, the interleavings in these respective sections – already individually exponential – can multiply out. While static analysis methods may often help determine where interleavings matter, maintaining static analysis tools that handle hybrid concurrency models costs considerable engineering time.
Justification: With hybrid concurrency models becoming the norm, formal methods researchers must now develop good heuristics to avoid interleaving multiplication across different APIs. Isolating different code blocks and verifying them separately is also not an option because of the broad interfaces presented by each such block.

2.5 Unify handling of concurrency and resources

It is common to treat correctness and performance separately, and to assume that the amount of resources available only affects performance and not correctness. This is not true in general: in MPI for example, having *extra capacity to buffer messages* can often *cause additional deadlocks* (of course, in other situations, less buffering increases the number of deadlocks). Given that concurrent protocols will be deployed across systems that vary widely in terms of resource availabilities, it is crucial to understand such unintended dependencies.
Justification: In recent work [17], we show how having the single unifying model of *MPI happens-before* enables us to analyze concurrent behaviors in the presence of resource induced behavioral variations. We can for instance analyze whether a given MPI program is vulnerable to deadlocks when one tries to increase its performance by increasing buffering. Now, with hybrid concurrency models resulting from the use of multiple APIs, it is important to have a good understanding of how the concurrency space changes with resources. Verification methods must incorporate such 'resource awareness' into their analysis.

2.6 Evolve distributed verification algorithms

For a number of reasons (beyond those mentioned in § 2.5), it is important to formally verify designs configured at higher ends of the problem scale. For many MPI programs it is simply impossible to load-up a problem within a uniprocessor, or expect to obtain enough computing power to run these problems. Thus we must develop truly distributed verification algorithms where we can use multiple (*e.g.,* thousands) of computing nodes, and conduct large-scale HPC application verification on such systems. We must also develop effective methods to conduct coverage/scalability tradeoffs.
Justification: Recently, we have developed a variant of the ISP algorithm where, instead of a centralized scheduler that computes the MPI happens-before and bases its scheduling on it, we let MPI processes run on their own nodes, and employ a distributed algorithm based on logical clocks. We are developing this algorithm in the context of the tool Distributed Analyzer of MPI (DAM) in collaboration with LLNL [18]. Our experience strongly supports the need for distributed verification methods that provide coverage/performance tradeoffs.

2.7 Develop formal equivalence criteria

One of the biggest "surprises" during parallelization is the mismatch between answers obtained in a sequential (but unacceptably slow) implementation versus that obtained from a parallel implementation.
Justification: When one imagines problems being coded up in a mixture of MPI and CUDA/OpenMP/OpenCL, the number of sources of uncertainty in terms of numerical precision go up. Even when the compilers are correct and user code is non-buggy (big "if"s), different answers will arise due to word width restrictions and associativity changes during parallelization (in floating-point arithmetic, $a + (b + c) \neq (a + b) + c$). Another source of bugs in compilers and hand-

optimized codes is the omission of `memory fence` instructions. This can introduce very subtle bugs in iterative algorithms which may yield plausible-looking answers, but are still inherently flawed due to their use of values that are a few more time steps behind than intended (example due to Gropp). Such dangers underscore the importance of having equivalence criteria between sequential and parallel programs, some of which are discussed in [19].

2.8 Develop model-specific reductions

Techniques that help limit state explosion during verification are best tailored for the concurrency model at hand. For example, the preemption bounding approach used in CHESS [13] is ineffective for MPI because it heuristic is designed to alter the shared memory effects (lost atomicity, wrong updates of globals, *etc.*) – not the critical message matching steps occurring within the MPI runtime. With hybrid programming becoming the norm, suitable bounding methods must be devised for each concurrency model.
Justification: In recent work [18], we have developed a technique that we call *bounded mixing* to contain dynamic analysis complexity. The success of this method depends on the fact that the control flow effects of an MPI non-deterministic receive do not last beyond a handful of succeeding operations. Bounded mixing is able to turn a large exponential space into the summation of much smaller exponential spaces by exploring combinations of non-deterministic receives only within small *sliding windows.* This bounding heuristic is tailored for message passing APIs.

2.9 Distribute well-integrated tools

FV researchers must release their tools well integrated into widely used tool integration frameworks.
Justification: Through collaboration with IBM, we have released a front-end for ISP called "GEM" (Graphical Explorer for Message passing) [12] well integrated into the Eclipse Parallel Tools Platform (PTP) Version 4.0. We plan to integrate PUG also within PTP. This situates ISP, PUG (and soon DAM) within easy reach of real designers who will also find other tools (*e.g.,* performance analyzers) well integrated within PTP for concerted use. Our ongoing experience with PTP supports our position on tools.

2.10 Teach using formal tools

Concurrency textbooks must emphasize the conceptual basics of various concurrency models. Unfortunately, most existing books almost entirely rely on examples where *ad hoc* experiments and their execution outcomes are listed.
Justification: As an example, a scenario pertaining to MPI's *non-blocking wildcard probe*s ("one can probe one MPI send; but match yet another one") is discussed almost entirely based on examples in the existing MPI documentation. Our GEM tool integrates a happens-before viewer for MPI, and in our EuroMPI 2009 tutorial, we demonstrated that this scenario can be deduced as a formal consequence of happens-before. We can rein in the complexity of concurrency education only if we choose to emphasize such fundamental deduction rules – as opposed to encouraging programmers to memorize seemingly disparate facts.

3. CONCLUDING REMARKS

The future success of concurrent system design depends on the development of formal analysis tools that can handle hybrid concurrency models. We presented ten field-proven steps to accelerate this research so that we can place future HPC system design on a rigorous footing.
Acknowledgments: We are grateful for the timely research funding from Microsoft through their HPC Institutes program, and NSF CNS 0824021. Discussions with Gropp, Lusk, Thakur, and de Supinski are gratefully acknowledged.

4. REFERENCES

[1] MPI 2.1 Standard. MPI Standard 2.1, `http://www.mpi-forum.org/docs/`.
[2] Stephen F. Siegel. Model checking nonblocking MPI programs. VMCAI 2007, pages 44–58, 2007.
[3] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97*, pages 174–186.
[4] S. Pervez, G. Gopalakrishnan, R.M. Kirby, R. Thakur, and W. Gropp. Formal methods applied to high-performance computing software design. *Software: Practice & Experience*, Vol 40, 23-43.
[5] OpenCL: `http://www.khronos.org/opencl`.
[6] CUDA: `http://gpgpu.org/isc2009`.
[7] Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of MPI 2.0. *Science of Computer Programming*, 2009. Accepted.
[8] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations. In *CAV08*, pages 66–79, 2008.
[9] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, , and Rajeev Thakur. Formal verification of practical MPI programs. In PPoPP, pages 261–269, 2009.
[10] Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. MCC - a runtime verification tool for MCAPI user applications. In *FMCAD*, pages 41–44. IEEE, November 2009.
[11] Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. An approach to formalization and analysis of message passing libraries. FMICS best paper, pages 164–181, 2008.
[12] Alan Humphrey, Christopher Derrick, Ganesh Gopalakrishnan, and Beth Tibbitts, GEM: Graphical Explorer for MPI Programs, Parallel Software Tools and Tool Infrastructures (ICPP workshop), 2010. `http://www.cs.utah.edu/fv/GEM`
[13] CHESS: Find and reproduce Heisenbugs in concurrent programs. Microsoft Research.
[14] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*, 2010. Accepted.
[15] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert Kirby. Runtime model checking of multi-threaded C/C++ programs. TR UUCS-07-008.
[16] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. PADTAD, Seattle, WA, July 2008.
[17] `http://www.cs.utah.edu/~sarvani/dissertation.html`. Some parts accepted in EuroMPI 2010.
[18] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky, A Scalable and Distributed Dynamic Formal Verifier for MPI Programs, Supercomputing (SC), 2010, Accepted. `www.cs.utah.edu/fv/DAM/sc10.pdf`.
[19] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM TOSEM*, 17(2):Article 10, 1–34, 2008.