Formal Analysis of MPI-Based Parallel Programs: Present and Future*

Ganesh Gopalakrishnan[1] Robert M. Kirby[1] Stephen Siegel[2] Rajeev Thakur[3]
William Gropp[4] Ewing Lusk[3] Bronis R. de Supinski[5] Martin Schulz[5] Greg Bronevetsky[5]

[1]University of Utah [2]University of Delaware [3]Argonne National Laboratory
[4]University of Illinois at Urbana-Champaign [5]Lawrence Livermore National Laboratory

Abstract The Message Passing Interface (MPI) is the dominant programming model for high-performance computing. Applications that use over 100,000 cores are currently running on large HPC systems. Unfortunately, MPI usage presents a number of correctness challenges that can result in wasted simulation cycles and errant computations. In this article, we describe recently developed formal and semi-formal approaches for verifying MPI applications, highlighting the scalability/coverage tradeoffs taken in them. We concentrate on MPI because of its ubiquity and because similar issues will arise when developing verification techniques for other performance-oriented concurrency APIs. We conclude with a look at the inevitability of multiple concurrency models in future parallel systems and point out why the HPC community urgently needs increased participation from formal methods researchers.

1 MPI and Importance of Correctness

The Message Passing Interface (MPI) Standard was originally developed around 1993 by the MPI Forum, a group of vendors, parallel programming researchers, and computational scientists. The document is not issued by an official standards organization, but has become a *de facto* standard through near universal adoption. Development of MPI continues, with MPI-2.2 released in 2009. The standard has been published on the web [1] and as a book, and several other books based on it have appeared (e.g., [2, 3]). Implementations are available in open source form [4, 5], from software vendors such as Microsoft and Intel, and from every vendor of high-performance computing systems. MPI is widely cited; Google scholar returned 28,400 hits for "`+MPI +"Message Passing Interface"`" in September, 2010. MPI is ubiquitous: it runs everywhere, from embedded systems to the largest of supercomputers. Many MPI programs represent dozens, if not hundreds, of person-years of development, including their calibration for accuracy, and performance tuning. MPI was designed to support highly scalable computing; applications that use over 100,000 cores are currently running on systems such as the IBM Blue Gene/P (Figure 1) and Cray XT5. Scientists and engineers all over the world use MPI in thousands of applications, such as investigations of alternate energy sources and weather simulations. For high-performance computing, MPI is by far the dominant programming model; most (at some centers, all) applications running on supercomputers use MPI. MPI is considered a requirement even for Exascale systems, at least by application developers [6].

Unfortunately, the MPI debugging methods available to these application developers are wasteful, and ultimately unreliable. Existing MPI testing tools seldom provide coverage guarantees. They examine essentially equivalent execution sequences, thus reducing testing efficiency. These methods fare even worse at large problem scales. Consider the costs of HPC bugs: (i) a high-end HPC center costs hundreds of millions to commission, and the machines become obsolete within six years, (ii) in many of these centers, over $3 million dollars are spent in electricity costs alone each year, and (iii) research teams apply for computer time through competitive proposals, spending years planning their experiments. In addition to these development costs, one must add the costs to society of relying on potentially defective software to inform decisions on issues of great importance, such as climate change.

Because the stakes are so high, formal methods can play an important role in debugging and verifying MPI applications. In this paper, we give a survey of existing techniques, clearly describing the pros and cons of each approach. These methods have value far beyond MPI, addressing the general needs of future concurrency application developers, who will inevitably use a variety of low-level concurrency APIs.

Overview of MPI: Historically, parallel systems have used either *message passing* or *shared memory* for communication. Compared to other message passing systems noted for their parsimony, MPI supports a large number of cohesively engineered features essential for designing large-scale simulations. MPI-2.2 [1] specifies over 300 functions. Most developers use only a few dozen of these in any given application.

MPI programs consist of one or more threads of execution

*Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, CCF-0903408, CCF-0953210, CCF-0733035, and DOE ASCR DE-AC02-06CH11357. Accepted 6/1/11 to appear in CACM.

Figure 1. The Intrepid Blue Gene/P Open Science machine at Argonne with 163,840 cores, half a Petaflop (peak), consuming 1.26MW

with private memories called "MPI processes" and communicate through various types of message exchanges. The two most commonly used types are point-to-point messages (e.g., sends and receives) and collective operations (e.g., broadcasts and reductions). MPI also supports *nonblocking operations* that help overlap computation and communication, and *persistent operations* that make repeated sends/receives efficient. In addition, MPI allows processes and communication spaces to be structured using *topologies* and *communicators*. MPI's *derived datatypes* further enhance the portability and efficiency of MPI codes by enabling the user to communicate noncontiguous data with a single MPI function call. MPI supports a limited form of shared-memory communication based on one-sided communication. A majority of MPI programs are still written using the "two-sided" (message passing oriented) constructs, and *we shall focus on these in the rest of this paper*. Finally, MPI-IO addresses a whole series of issues pertaining to portable access to high-performance input/output systems.

Nature of MPI Applications: MPI applications and libraries are written predominantly in C, C++, or Fortran. Languages that use garbage collection or managed runtimes (e.g., Java and C#) are rarely used in HPC. These linguistic choices are driven by pre-existing libraries, maturation of compilation technology, and the need to efficiently manage memory allocation. Memory is one of the most precious resources in large-scale computing systems: a common rule of thumb is that an application cannot afford to consume more than one byte per FLOP. Computer memory is expensive and increases cluster energy consumption. It is widely believed that computer scientists using multiple cores even in traditional shared-memory applications will have to work with low amounts of cache coherent memory per core, and manage data locality—something MPI programmers are used to doing. It is also becoming clear that future uses of MPI will be in conjunction with shared-memory libraries and notations such as Pthreads [7], OpenMP [8], CUDA [9], and OpenCL [10], in order to reduce message-copy proliferation and exploit future commodity CPUs that will increasingly support shared memory concurrency.

While some MPI applications are written from scratch, many are built on top of user libraries, typically also written using MPI. Many such finely crafted libraries exist, including medium-sized libraries such as ParMETIS [11], used for parallel hypergraph partitioning, ScaLAPACK [12] for high-performance linear algebra, and PETSc [13], for solving partial differential equations.

Verification and Debugging Methods: MPI processes execute in disjoint address spaces, interacting through API functions involving deterministic and non-deterministic features, collective operations, and non-blocking communication commands. Existing (shared memory concurrent program) debugging techniques do not efficiently carry over to MPI, whose operations typically match and complete out of program order according to an MPI-specific *matches-before order* [14, 15]. The overall behavior of an MPI program is also heavily influenced by how specific MPI library implementations take advantage of the latitude provided by the MPI standard.

An MPI program bug is introduced while modeling the problem, while approximating the numerical methods, or while coding, including whole classes of floating-point issues [16]. While lower level bugs such as deadlocks and data races are serious concerns, their detection requires specialized techniques of the kind described here. Since many MPI programs are poorly parameterized, it is not easy to downscale a program to a smaller instance and locate the bug. For all these reasons, we need a variety of verification methods, each narrowly focused on subsets of correctness issues and making specific tradeoffs. Our main focus in this paper is formal analysis methods for smaller scale MPI applications and semi-formal analysis methods for the very large scale. For detecting MPI bugs in practice, formal analysis tools must be coupled with run-time instrumentation methods found in tools such as Umpire [17], Marmot [18], and MUST [19]. Much more research is needed in such tool integration.

Dynamic Analysis: MPI provides many nondeterministic constructs. These free the runtime system to choose the most efficient way to carry out an operation, but also mean that a program can exhibit multiple behaviors when run on the same input, posing well-known verification challenges. An example is a communication race arising from a "wildcard" receive, an operation that does not specify the source process of the message to be received, leaving this decision to the runtime system. Many subtle program defects are revealed only for a specific sequence of choices. While random testing *might* happen upon one such sequence, it is hardly a reliable approach.

In contrast, dynamic verification approaches control the exact choices made by the MPI runtime, and use this control to explore methodically a carefully constructed subset of behaviors. For each behavior, a number of properties may be verified, including absence of deadlocks, assertion violations, incompatible data payloads between senders and receivers, and MPI resource leaks. Using a formal model of the MPI semantics, a dynamic verifier can conclude that *if* there are no violations on the subset of executions, *then* there can be no violation on *any* execution. If even this reduced subset cannot be exhaustively explored, the user can specify precise coverage criteria and obtain a lesser (but still quantifiable) degree of assurance.

This approach has been realized in the shared memory concurrency domains in tools such as VeriSoft [20], Java Pathfinder [21], and CHESS [22]. In §2, we present two MPI-specific dynamic verifiers, ISP and DAMPI. Such tools offer distinct advantages: they do not require any modification to the program source code, compiler, libraries, or the software stack; they are language-agnostic; they can scale to relatively large process counts by exploiting MPI semantics; and they are fully automated.

Full-Scale Debugging: Traditional "step-by-step" debugging techniques are untenable for traces that involve millions of threads. In §3, we describe a new debugging approach that analyzes an execution trace and partitions the threads into equivalence classes based on their behavior. Experience on real, large-scale systems shows that typically only a small number of classes emerge, and the information provided can help a developer to isolate a defect. While this approach is not comparable to the other approaches discussed here, in that the focus is on the analysis of one trace rather than reasoning about all executions, it clearly has an advantage in sheer scalability.

Symbolic Analysis: The approaches discussed above are only as good as the set of inputs to which they are applied. Defects that are only revealed for very specific input or parameter values may be difficult to discover with those techniques alone. *Symbolic execution* [23] is a well-known approach for dealing with this problem, and in §4 we describe how this approach can be successfully applied to MPI programs. The TASS toolkit [24] uses symbolic execution and state enumeration techniques to verify properties of MPI programs, not only for all possible behaviors of the runtime system, but for all possible inputs as well. It can even be used to establish that two versions of a program are functionally equivalent, at least within specified bounds. On the other hand, to implement this approach requires sophisticated theorem proving technology and a symbolic interpreter for all program constructs and library functions; for this reason, presently TASS supports only C and a subset of MPI. Moreover, it generally cannot scale beyond a relatively small number of processes, though as we will show, defects which usually appear only in large configurations can often be detected in much smaller configurations using symbolic execution.

Static Analysis: Compilers use static analyses to verify a variety of simple safety properties of sequential programs. These work on a formal structure that abstractly represents some aspect of the program, such as a control flow graph (CFG). Extending these techniques to verify concurrency properties of MPI programs, such as deadlock-freedom, will require new abstractions and techniques. In §5, we outline a new analysis framework targeting this problem that introduces the notion of a *parallel CFG*. This approach has the advantage that the pCFG is *independent of the number of processes*, which makes it essentially infinitely scalable. However, these analyses are difficult to automate, so they may require user-provided program annotations to guide the analysis.

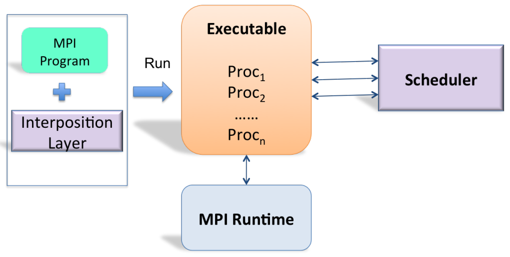

Figure 2. Overview of ISP

2 Dynamic Verification of MPI

We present two dynamic analysis methods for MPI programs. The first approach, implemented by the tool ISP ([25, 26], Figure 2) delivers a formal coverage guarantee with respect to deadlocks and local safety assertions [14]. ISP has been demonstrated on MPI applications of up to 15,000 lines of code. Running on modern laptop computers, ISP can verify such applications for up to 32 MPI processes on mostly deterministic MPI programs. Several tutorials on ISP are being given, including at major HPC venues [27, 28].

ISP's scheduler (Figure 2) exerts centralized control over every MPI action. This approach limits ISP scalability to at most a few dozen MPI processes, which does not help programmers who encounter difficulties at higher ends of the scale, at which user applications and library codes often use different algorithms. Therefore, what if a designer has optimized his/her HPC computation to work efficiently on 1,000 processors and suddenly finds an inexplicable bug? Traditional HPC debugging support is severely lacking in terms of ensuring coverage goals. To address this difficulty, we have built a tool called DAMPI (distributed analyzer of MPI, [29]) that can deterministically replay schedules and ensure non-determinism coverage. DAMPI verifies MPI programs by running them on supercomputers. It scales far more than ISP.

Dynamic Verification using ISP: For programs with non-deterministic MPI calls, simply modulating the absolute times at which MPI calls are issued (e.g., by inserting non-deterministic sleep durations, as done by stress-testing tools) is ineffective [30] because most often this does not alter the way in which racing MPI sends match with MPI non-deterministic receives deep inside the MPI runtime. Also, such delays unnecessarily slow down the entire testing.

The example of Figure 3 helps describe ISP's *active testing* approach in detail. In this example, if P_2's *Isend* can match P_1's *Irecv*, we will encounter a bug. The question is: Can this match occur? The answer is yes: first, let P_0 issue its non-blocking *Isend* call and P_1 its non-blocking *Irecv* call; then allow the execution to cross the *Barrier* calls; after that, P_2 can

P_0	P_1	P_2
$Isend(to:1,22);$	$Irecv(from:*,x)$	$Barrier;$
$Barrier;$	$Barrier;$	$Isend(to:1,33);$
	$if(x==33)bug;$	

Figure 3. Bug manifests on some runtimes

issue its *Isend*. At this point, the MPI runtime faces a non-deterministic choice of matching either *Isend*. We can obtain this particular execution sequence only if the *Barrier* calls are allowed to match *before* the *Irecv* matches. Existing MPI testing tools cannot exert such fine control over MPI executions. By interposing a scheduler (Figure 2), ISP can safely reorder, at runtime, MPI calls that the program issues. In our present example, ISP's scheduler (i) *intercepts* all MPI calls coming to it in program order, (ii) dynamically reorders the calls going into the MPI runtime (ISP's scheduler sends *Barriers* first; this is correct according to the MPI semantics), and (iii) at that point discovers the non-determinism.

Once ISP determines that two matches could occur, it re-executes (replays from the beginning) the program in Figure 3 twice: once with the *Isend* from P_0 matching the receive, and the second time with that from P_2 matching it. To ensure these matches occur, ISP dynamically rewrites $Irecv(from:*)$ into $Irecv(from:0)$ and $Irecv(from:2)$ in these replays. If we did not so determinize the *Irecv*s, but instead issued $Irecv(from:*)$ into the MPI runtime, we might match an *Isend* from another process. ISP discovers the maximal extent of non-determinism through dynamic MPI call reordering and achieves scheduling control of relevant interleavings by dynamic API call rewriting. While pursuing relevant interleavings, ISP additionally detects the following error conditions: (i) deadlocks, (ii) resource leaks (e.g., MPI object leaks), and (iii) violations of C assertions placed in the code.

It is important to keep in mind that MPI programmers often use non-blocking MPI calls to enhance computation/communication overlap. They use non-deterministic MPI calls in master/worker patterns to detect which MPI process finishes first, so that more work can be assigned to it. When these operations, together with "collective" operations such as *Barrier*s are all employed in one example, one can very well obtain situations as shown in Figure 3. The safety-net provided by tools such as ISP is therefore essential to support efficiency oriented MPI programming.

The use of a tool such as ISP gives all the benefits of testing (ability to run the final code of interest) while de-biasing from the behavior of specific MPI libraries. Obviously, ISP has no capability to detect infinite loops (an undecidable problem); it assumes that all iterative loops terminate. It guarantees MPI communication non-determinism coverage under the given test harness. It helps avoid exponential interleaving explosion through two means. First, it avoids redundantly examining equivalent behaviors (for example, it avoids examining the $n!$ different orders in which an MPI barrier call might be invoked; a testing tool might explore this space needlessly). it

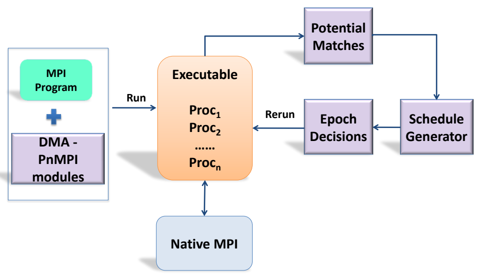

Figure 4. Distributed MPI Analyzer

also comes with execution-space sampling options.

ISP has examined many large MPI programs of thousands of lines of C code that make millions of MPI calls. It has found many subtle bugs on a variety of applications [30]. We have also built the Graphical Explorer of Message passing (GEM) tool [31] that hosts the ISP verification engine. GEM is an official component of the Parallel Tools Platform (PTP, [32]) end-user runtime (PTP version 4.0 onwards), and as such makes dynamic formal verification of MPI become available in a seamlessly integrated manner within a popular development environment.

Dynamic Verification using DAMPI: A widely used complexity reduction approach is to debug a given program after suitably downscaling it. One practical difficulty in carrying out this approach is that many programs are poorly parameterized. For such programs, if a problem parameter is reduced, it is often unclear whether another parameter should be reduced proportionally, logarithmically, or through some other relationship. A more serious difficulty is that some bugs are only manifest when a problem is run at scale. The algorithms employed by applications and/or the MPI library itself can change depending on problem scale. Also resource bugs (e.g., buffer overflows) often show up only at scale.

While user-level dynamic verification supported by ISP resolves significant nondeterminism, testing at larger scales requires a decentralized approach where the supercomputing power aids verification. We have implemented this idea in our tool framework DAMPI [29].

The key insight that allows us to design the decentralized scheduling algorithm of DAMPI is that a nondeterministic (ND) operation, such as `MPI_Irecv(MPI_ANY_SOURCE)` or `MPI_Iprobe(MPI_ANY_SOURCE)`, represents a point on the timeline of the issuing process when it *commits* to a match decision. It is natural to think of each such event as starting an *epoch*—an interval stretching from the current ND event up to (but not including) the next ND event. All deterministic receives can be assigned the same epoch as the one in which they occur. Even though the epoch is defined by one ND receive matching another process's send, how can we determine all *other* sends that can match it? The solution is to pick all those sends that are *not causally after* the ND receive (and subject to MPI's non-overtaking rules). We determine these sends using an MPI-specific version of Lamport clocks [33], which strikes a good compromise between scalability and omissions.

Experimental results show that DAMPI can effectively test realistic problems that run on more than a thousand CPUs, by exploiting the parallelism and the memory capacity offered by clusters. It has successfully examined all benchmarks from the Fortran NAS Parallel Benchmark suite [34], with an instrumentation overhead of less than 10% compared to ordinary testing (but able to provide non-determinism coverage, which ordinary testing does not).

Our experiments also revealed one surprising fact: none of the MPI programs in [34] that employ `MPI_Irecv(MPI_ANY_SOURCE)` calls actually exhibited non-determinism under DAMPI. This means that the programmer had somehow determinized the program (perhaps through additional MPI call arguments). We believe that alternatives to dynamic analysis (*e.g.*, static analysis or code inspection) are, as yet, incapable of yielding this insight in practice.

3 Full-Scale Debugging

The approach described in this section targets the large-scale systems that will emerge over the next several years. Current estimates anticipate half a billion to four billion threads in exascale systems. With these levels of concurrency, we must target debugging techniques that can handle these scales, as experience shows that bugs often are not manifest until a program is run at the largest scale. The bugs often depend on the input, which can be significantly different for full-scale runs. Further, certain types of errors, such as integer overflows, often depend directly on the number of processors.

However, most debugging techniques do not translate well to full scale runs. The traditional paradigm of stepping through code not only has significant performance issue with large processor counts but is impractical with thousands of processes or threads, let alone billions. Dynamic verification techniques offer paradigmatic scaling but have even more performance difficulties, particularly when the number of interleavings depends on the process count.

Faced with the growing scaling requirements, we require new techniques to limit the focus of our debugging efforts. We have recently developed mechanisms to identify *behavioral equivalence classes*. Specifically, our observation is that when errors occur in large-scale programs, they do not exhibit thousands or millions of different behaviors. Instead, they exhibit a limited set of behaviors in which all processes follow the same erroneous path (one behavior) or one or a few processes follow an erroneous path, which then may lead to changes in the behavior of a few related processes (two or three behaviors). While the effect may trickle further out, we rarely observe more than a half dozen behaviors, regardless of the total number of processes used in an MPI program.

Given the limited behaviors being exhibited, we can then focus on only debugging representative processes from each

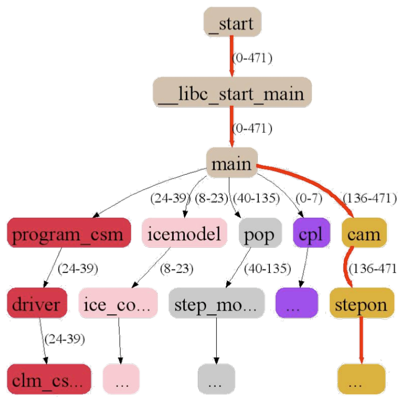

Figure 5. STAT Process Equivalence Classes

behavioral class, rather than having to debug all processes at once, thereby enabling the debugging of problems that were previously not debuggable.

The Stack Trace Analysis Tool (STAT) [35] achieves this goal by attaching to all processes in a large-scale job and gathering stack traces sampled over time in a low overhead and distributed manner. It then merges these stack traces to identify which processes are executing similar code. We consider a variety of equivalence relations, which we often use hierarchically. For example, for any $n \geq 1$, we consider two processes as equivalent if they agree on the first n function calls issued. Increasing n refines this equivalence relation, giving the user control of the precision-accuracy tradeoff.

The resulting tree easily identifies different execution behaviors. Figure 5 shows the top levels of the tree obtained from a run of the Community Climate System Model (CCSM). This application uses five separate modules to model land (CSM), ice, ocean (POP), and atmosphere (CAM) and to couple the four models. In the figure, we can quickly identify that MPI processes 24-39 are executing the land model, 8-23 the ice model, 40-135 the ocean model, and 136-471 the atmospheric model, while 0-7 are executing the coupler. If a problem should be observed in one of the models, we can use this information to concentrate on this subset of tasks; in the case of a more broad error we can pick representatives from these five classes and thereby reduce the initial debugging problem to five processes. We have used the latter to successfully debug several codes with significantly shortened turnaround time, including an Algebraic Multigrid (AMG) package, which is fundamental for many of our codes.

Additionally, tools like STAT have the ability to detect outliers, which can directly point to erroneous behavior without the need for further debugging. For example, we used STAT on the CCSM code introduced above when it hung on over 4096 processes. The stack trace tree showed one task executing in an abnormally deep stack and on closer examination of the stack trace it not only showed that a mutex lock operation within the MPI implementation was called multiple times, which created the deadlock, but also exactly where in the code

```
for (i=0; i<n; i++) a[i] = read element i;
sum = 0.0;
for (i=0; i<n; i++)
  if (a[i]>0.0) sum += a[i];
output sum;
```

(a) `adder_seq`: sequential version

```
int first = n*rank/nprocs;
int count = n*(rank+1)/nprocs - first;
for (i=0; i<count; i++) a[i]=read element first+i;
sum = 0.0;
for (i=0; i<count; i++)
  if (a[i]>0.0) sum += a[i];
if (rank == 0) {
  for (j=1; j<nprocs; j++) {
    recv into buffer from rank j;
    sum += buffer;
  }
  output sum;
} else { send sum to rank 0; }
```

(b) `adder_par`: parallel version

Figure 6. Programs that read an array from a file, sum positive elements, and output result.

the respective erroneous mutex lock call occurred. This lead to a quick fix of the MPI implementation.

Our current efforts include extensions that can provide better identification of the behavior equivalence classes as well as techniques to discern relationships between the classes [36]. Additional directions include using the classes to guide dynamic verification techniques.

4 Symbolic Analysis of MPI

The basic idea of symbolic execution is to execute the program using symbolic expressions in place of the usual (concrete) values held by program variables [23]. The inputs and initial values of the program are *symbolic constants* $X_0, X_1, \ldots$, so-called because they represent values that do not change during execution. Numerical operations are replaced by operations on symbolic expressions. For example, if program variables `u` and `v` hold values X_0 and X_1, respectively, then `u+v` will evaluate to the symbolic expression $X_0 + X_1$.

The situation becomes more complicated at a branch point. Suppose the branch condition is the expression `u+v>0`. Since the values are symbolic, it is not necessarily possible to say whether the condition evaluates to *true* or *false*. Instead, both possibilities must be explored. Symbolic execution handles this by introducing a hidden boolean-valued symbolic variable, the *path condition* `pc`, which is used to record the choices made at branch points. This variable is initialized to *true*. At a branch, a *nondeterministic* choice is made between the two branches, and `pc` is updated accordingly. In our example, `pc` would be assigned the symbolic value of $\texttt{pc} \wedge u+v > 0$ if the *true* branch is selected; if this is the first branch encountered, this means `pc` will now hold the symbolic expression $X_0 + X_1 > 0$. If instead the *false* branch is chosen, `pc` will hold $X_0 + X_1 \leq 0$. Hence the path condition records the condition the inputs must satisfy in order for a particular path to be followed. Model-checking techniques can then be used to explore all nondeterministic choices and verify a property holds on all executions [37], or generate a test set. An automated theorem prover, such as CVC3 [38], can be used to determine if `pc` becomes unsatisfiable, in which case the current path has become infeasible and can be pruned from the search.

One of the advantages of symbolic techniques is that they map naturally to message-passing based parallel programs. Our *Toolkit for Accurate Scientific Software* (TASS) [24], based on CVC3, uses symbolic execution and state exploration techniques to verify properties of such programs. The TASS verifier takes as input the MPI/C source program and a specified number of processes, and instantiates a symbolic model of the program with that process count. TASS maintains a model of the state of the MPI implementation, which includes the state of the message buffers. Like all other program variables, the buffered message data is represented as symbolic expressions. The user may also specify bounds on input variables in order to make the model finite or sufficiently small. An MPI-specific partial order reduction scheme restricts the set of states explored, while still guaranteeing that if a violation to one of the properties exists (within the specified bounds), it will be found. A number of examples are included in the TASS distribution, including cases where TASS reveals defects in the MPI code, such as a diffusion simulation code from the FEVS verification suite [39].

TASS can verify the standard safety properties outlined above, but its most important feature is its ability to verify that two programs are *functionally equivalent*, i.e., if given the same input, they will always return the same output. This is especially useful in scientific computing, where developers often begin with a simple sequential version of an algorithm and then gradually add optimizations and parallelism. The production code is typically much more complex but is intended to be functionally equivalent to the original. The symbolic technique used to compare two programs for functional equivalence is known as *comparative symbolic execution* [40].

To illustrate the approach, consider the example of Figure 6. The sequential program reads n floating-point numbers from a file, sums the positive elements, and returns the result. A parallel version divides the file into approximately equal-sized blocks. Each process reads one of these blocks into a local array and sums the positive elements in its block. On all processes other than process 0, this partial sum is sent to process 0. Process 0 receives these and adds them to its partial sum, and then outputs the final result.

Ignoring round-off error, the two programs are functionally equivalent: given the same file, they will output the same result. To see how this can be established in a simple case, consider the case $n = \texttt{nprocs} = 2$. Call the elements of the file X_0 and X_1. There are then four paths through the sequential program, due to the two binary branches `if a[i]>0.0`. One of these four paths, which arises when both elements are positive, yields the path condition $X_0 > 0 \wedge X_1 > 0$ and output $X_0 + X_1$. We can now explore all possible executions of `adder_par` in which the initial path condition is $X_0 > 0 \wedge X_1 > 0$. (There are many such executions, due to the various ways in which the statements from the two processes can be interleaved.) In each of these executions, the output will be $X_0 + X_1$. A similar fact can be established for the other three paths through the sequential program. Taken together, these facts imply the programs will produce the same result on any input (for n = `nprocs` = 2).

The ability to uncover defects at small scales is a primary advantage of symbolic approaches. Isolating and repairing a defect that only manifests itself in tests with thousands of processes and huge inputs is difficult. Several research projects have focused on making traditional debuggers scale to thousands of processes for just this reason. However, it would be more practical to force the same defect to manifest itself at smaller scales and then to isolate the defect at those scales.

A real-life example illustrates this point. In 2008, a failure was reported in the MPICH2 MPI implementation. The failure occurred when calling the broadcast function `MPI_Bcast` using 256 processes and a message of just over `count = 3200` integers. Investigation eventually revealed that the defect was in a function used to implement broadcasts in specific situations (Figure 7(a)). For certain inputs, the "size" argument (`nbytes-recv_offset`) to an MPI point-to-point operation—an argument which should always be nonnegative—could in fact be negative. For 256 processes and integer data (`type_size = 4`), this fault occurs if and only if $3201 \leq$ `count` ≤ 3251.

The problematic function is guarded by the code shown in Figure 7(b). This refers to three compile-time constants, `MPIR_BCAST_SHORT_MSG`, `MPIR_BCAST_LONG_MSG`, and `MPIR_BCAST_MIN_PROCS`, which are defined elsewhere as 12288, 524288, and 8, respectively. Essentially, the function is only called for "medium-sized" messages, when the number of processes is a power of 2 and above a certain threshold. With these settings, the smallest configuration revealing the defect is 128 processes, with `count = 3073`.

A symbolic execution technique that checks that the "size" arguments to MPI functions are always non-negative easily detects the defect. If we also treat the three compile-time constants as symbolic constants, the defect can be manifest at the much smaller configuration of 8 processes and `count = 1` (in which case `nbytes-recv_offset` $= -1$). Such an approach likely would have detected this defect earlier and with much less effort.

Arithmetic. In the adder example, we pretended that the values manipulated by the program were the mathematical real numbers, and that the numerical operations were the (infinite precision) real operations. If instead we treat the values and the operations as finite-precision floating-point values, the

```
relative_rank = (rank >= root ?
   rank - root : rank - root + comm_size);
nbytes = type_size * count;
scatter_size =
   (nbytes + comm_size - 1)/comm_size;
mask = 0x1; i = 0;
while (mask < comm_size) {
  relative_dst = relative_rank ^ mask;
  dst_tree_root = relative_dst >> i;
  dst_tree_root <<= i;
  recv_offset = dst_tree_root * scatter_size;
  if (relative_dst < comm_size)
  { ... MPIC_Sendrecv(...,
          nbytes-recv_offset, ...); ...  }
  mask <<= 1; i++;
}
```

(a) `MPIR_Bcast_scatter_doubling_allgather`

```
else { /* (nbytes >= MPIR_BCAST_SHORT_MSG)
 && (comm_size >= MPIR_BCAST_MIN_PROCS) */
  if ((nbytes < MPIR_BCAST_LONG_MSG) &&
  (MPIU_is_pof2(comm_size, NULL))) {
     MPIR_Bcast_scatter_doubling_allgather
```

(b) invocation context

Figure 7. Excerpts from MPICH2 broadcast code. The fault occurs when the highlighted expression becomes negative.

two programs are not functionally equivalent, since floating-point addition is not associative [16]. Which is right? The answer is: it depends on what you are trying to verify. For functional equivalence, it is rare that the specification and implementation are expected to be "bit-level" equivalent (witness the adder example), so real-equivalence is probably more useful for this task. TASS uses a number of techniques specialized for real arithmetic; e.g., all real-valued expressions are put into a canonical form which is the quotient of two polynomials, to facilitate matching of expressions. For other tasks, such as detecting the defect in Figure 7, bit-level reasoning is more appropriate. Klee [41] is another symbolic execution tool for (sequential) C programs that uses bit-precise reasoning; there is no reason why these techniques could not be extended to parallel MPI-based programs.

5 Static Analysis of MPI

In the sequential arena, compiler techniques have been very successful at analyzing programs and transforming them to improve performance. However, MPI applications are difficult to analyze because: (i) the number of MPI processes is unknown at compile time and is unbounded; (ii) since MPI processes are identified by numeric ranks, applications use complex arithmetic expressions to define the processes involved in communications; (iii) the meaning of ranks depends closely on the MPI communicators that the MPI calls use; and (iv) MPI provides several nondeterministic primitives such as `MPI_ANY_SOURCE` and `MPI_Waitsome`. While some work has explored analysis of MPI applications, none has successfully addressed these challenges.

Some approaches treat MPI applications as sequential codes, which makes it possible to determine simple application behaviors, such as the relationship between writing to a buffer and sending it. However, they cannot represent or analyze the application's communication topology. Other approaches require knowledge of the number of processes to be used at runtime and analyze one copy of the application for each process. While this can capture the application's full parallel structure, it is inflexible and non-scalable. In [42], static analysis was applied to MPI activity analysis and performance improvement.

We have developed a novel compiler-analysis framework that extends traditional dataflow analyses to MPI applications, extracting the application's communication topology and matching the send and receive operations that may communicate at runtime [43]. The framework requires no runtime bound on the number of processes and is formulated as a dataflow analysis over the Cartesian product of control flow graphs (CFGs) from all processes, which we refer to as a parallel CFG (pCFG). During its execution, the analysis symbolically represents the execution of multiple sets of processes, keeping track of any send and receive operations. Process sets are represented using abstractions such as lower and upper bounds on process ranks or predicates such as "ranks divisible by 4". Sends and receives are periodically matched to each other by proving that the composition of their communication partner rank expressions produces the identity function, which establishes the application's communication topology. We can instantiate the analysis framework with a variety of "client analyses" that leverage the communication structure information derived by the framework to propagate their dataflow information as they do with a sequential application. These analyses and transformations include optimizations, error detection and verification, and information flow detection.

Finally, since topological information is key to a variety of compiler transformations and optimizations, our ongoing work is focusing on source code annotations that the developers can use to describe the application's communication topology and other properties. We will then leverage our analysis' abstract representation of this information to implement novel scalable analyses and transformations that can enable valuable optimizations in complex applications.

6 Concluding Remarks

The main objective of this article is to highlight the fact that formal and semi-formal methods are crucial for ensuring the reliability of message-passing programs across a vast scale of application sizes. Unfortunately, there has rarely been a discussion of these methods in the literature. To address this lacuna, we presented here the perspectives of academic researchers as well as HPC researchers working in national laboratories who are engaged in cutting edge HPC deployment. We propose a continuum of tools based on static analysis, dynamic analysis, symbolic analysis, and full-scale debugging, also complemented by more traditional error-checking tools. Our collaborations have resulted in promising tools that have helped specialize formal methods in ways that best address the needs of verifying real-world MPI applications.

Unfortunately, we alone can only barely scratch the surface of a vast problem area. There is a severe disconnect between "traditional computer scientists" and HPC researchers, which perhaps explains the serious shortage of formal methods researchers interested in HPC problems. This is especially unfortunate considering the many disruptive technologies that are on the horizon, including the usage of hybrid concurrency models along with MPI to program shared memory manycore systems. There are also emerging message passing based standards for embedded multicores (*e.g.*, MCAPI [44]) whose design and tool support can benefit from lessons learnt in the realm of MPI.

We propose two approaches to accelerate the use of formal methods in HPC. First, we must encourage researchers to develop verification methods specifically for *today's* APIs (MPI, and perhaps later CUDA and OpenCL). While verifying newer APIs might be easier, building formal support for widely used APIs can help sway today's HPC practitioners into becoming formal methods believers, and eventually becoming promoters of formal methods. We also believe that funding agencies must begin tempering the hoopla around performance goals (*e.g.*, "ExaFLOPs in this decade") by also setting formal correctness goals that lend the essential credence.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009. `http://www.mpi-forum.org/docs/`, 2009.

[2] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999. ISBN 0-262-57133-1.

[3] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996. ISBN 1-55860-339-5.

[4] MPICH2: High performance and widely portable MPI. `http://www.mcs.anl.gov/mpi/mpich`.

[5] Open MPI: Open source high performance MPI. `http://www.open-mpi.org/`.

[6] International exascale software project. `http://www.exascale.org/iesp/Main_Page`.

[7] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 2006.

[8] OpenMP Software. `http://www.openmp.org`.

[9] Compute Unified Device Architecture (CUDA). `http://www.nvidia.com/object/cuda_get.html`.

[10] OpenCL: Open Computing Language. http://www.khronos.org/opencl.

[11] ParMETIS - Parallel graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[12] L. S. Blackford et al. Scalapack user's guide. SIAM, 1997. ISBN:0-89871-397-8.

[13] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[14] Sarvani Vakkalanka. *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD Dissertation, 2010. http://www.cs.utah.edu/fv.

[15] Anh Vo. *Scalable Formal Dynamic Verification of MPI Programs Through Distributed Causality Tracking*. PhD Dissertation, 2011. http://www.cs.utah.edu/fv.

[16] David Goldberg. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[17] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000. Article 51.

[18] Bettina Krammer, Katrin Bidmon, Matthias S. Mller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing 2003*, September 2003.

[19] Tobias Hilbrich, Martin Schulz, Bronis de Supinski, and Matthias S. Müller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High Performance Computing*. Springer, 2009. doi:10.1007/978-3-642-11261-4_5.

[20] Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[21] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[22] CHESS: Find and reproduce Heisenbugs in concurrent programs. http://research.microsoft.com/en-us/projects/chess Accessed 12/8/09.

[23] James C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[24] S. F. Siegel et al. The Toolkit for Accurate Scientific Software web page. http://vsl.cis.udel.edu/tass, 2010.

[25] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *Computer Aided Verification (CAV 2008)*, pages 66–79, 2008.

[26] 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.

[27] Ganesh Gopalakrishnan and Robert M. Kirby. Dynamic verification of message passing and threading, January 2010. Half-day tutorial, 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2010,.

[28] Ganesh Gopalakrishnan, Bronis R. de Supinski, and Matthias Müller. Scalable dynamic formal verification and correctness checking of MPI applications, 2010. Half-day Tutorial at SC10.

[29] 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. In *Supercomputing*. IEEE Computer Society, 2010.

[30] Test results comparing ISP, Marmot,and mpirun. http://www.cs.utah.edu/fv/ISP_Tests.

[31] GEM - ISP eclipse plugin. http://www.cs.utah.edu/formal_verification/ISP-Eclipse.

[32] The Eclipse Parallel Tools Platform. http://www.eclipse.org/ptp.

[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[34] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[35] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *IPDPS*, 2007.

[36] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *SC09*, Portland, OR, 2009.

[37] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *TACAS 2003*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.

[38] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[39] Stephen F. Siegel and Timothy K. Zirkel. A Functional Equivalence Verification Suite. http://vsl.cis.udel.edu/fevs.

[40] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):Article 10, 1–34, 2008.

[41] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[42] Michelle M. Strout, Barbara Kreaseck, and Paul D. Hovland. Data-flow analysis for mpi programs. In *ICPP*, pages 175–184, 2006.

[43] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, pages 1–12, 2009.

[44] http://www.multicore-association.org.