2018 IEEE International Parallel and Distributed Processing Symposium

COMPI: Concolic Testing for MPI Applications

Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, Rajiv Gupta
Department of Computer Science and Engineering
University of California, Riverside
Riverside, USA
{*hli035, sli049, benavidz, chen, gupta*}*@cs.ucr.edu*

***Abstract*—MPI is widely used as the bedrock of HPC applications, but there are no effective systematic software testing techniques for MPI programs. In this paper we develop COMPI, the first practical concolic testing tool for MPI applications. COMPI tackles two major challenges. First, it provides an automated testing tool for MPI programs — it performs concolic execution on a single process and records branch coverage across all. Infusing MPI semantics such as MPI rank and MPI_COMM_WORLD into COMPI enables it to automatically direct testing with various processes' executions as well as automatically determine the total number of processes used in the testing. Second, COMPI employs three techniques to effectively control the cost of testing as too high a cost may prevent its adoption. By capping input values, COMPI is made practical as too large an input can make the testing extremely slow and sometimes even fail as memory needed could exceed the computing platform's memory limit. With two-way instrumentation, we reduce the unnecessary memory and I/O overhead of COMPI and the target program. With constraint set reduction, COMPI keeps significantly fewer constraints by removing redundant ones in the presence of loops so as to avoid redundant tests against these branches. Our evaluation of COMPI uncovered four new bugs in a complex application and achieved 69-86% branch coverage which far exceeds the 1.8-38% coverage achieved via random testing.**

I. Introduction

MPI is the de-facto standard for message-passing. It is used widely in the field of high performance computing (HPC). To improve programmers' productivity, significant progress has been made towards assisting in the debugging of errors in MPI applications. The research works on debugging can generally be classified into three categories: detecting MPI-semantics related bugs [9], [10], [11], [12], [13], [14]; detecting or diagnosing complex bugs at large scale [15], [16], [17], [18], [19], [20]; and finally reproducing and manifesting non-deterministic bugs [21], [22], [23].

In contrast to the above advances in debugging, little effort has been spent on developing systematic *software testing* techniques for MPI programs, even though testing is the predominant technique in industry to ensure software quality. The lack of testing techniques for MPI applications is likely the result of inadequate interaction between scientists, who play a leading role in HPC application development, and industrial software engineers [28]. In this less-studied area existing works include, message perturbation [24] to manifest non-deterministic bugs, FortranTestGenerator [26] to generate unit tests for subroutines of Fortran applications based on capture-and-replay approach, and MPISE [27] to mainly detect non-deterministic bugs based on symbolic execution [25].

We believe that there is an urgent need to explore effective systematic testing techniques in the field of HPC. As manually generating test inputs is very expensive, error-prone and non-exhaustive, *random testing* [29], [30], [31], [32] is commonly employed for automated test generation. But it is impossible to test all interesting behaviors of a program. *Symbolic techniques* [33], [34] overcome the limitation by generating inputs to force the execution of various paths. However, they do not scale to large programs because (1) large programs result in too complex constraints that are hard to be solved and (2) large programs lead to path explosion and thus exploring all paths is impractical.

A. Concolic Testing

Concolic testing [35], [36] has been proposed as a solution to the problem of solving complex constraints — it uses concrete values to simplify intractable constraints. To *alleviate* the path explosion problem, Burnim and Sen [37] propose a trade-off between the capability and practicality: they focus on branch coverage (the percentage of branches being executed at least once during testing) instead of path coverage, where the former is a more practical metric to evaluate code than the latter as the former is bounded by the total number of branches that is significantly smaller than the total number of paths.

Concolic testing automates the iterative testing of a program by automatically generating inputs with the goal of achieving a high branch coverage. It works as follows. Given a program, execution-path dominant variables reading inputs (from either a file or a command line) need to be marked by developers as *symbolic*, and then the program is instrumented such that the *symbolic execution code* is inserted into the given program. Testing involves iterative execution of this instrumented program. In each concrete execution, all operations of the marked variables are captured by the symbolic execution component. After each execution, symbolic execution history like encountered branches and *symbolic constraint set* satisfying the branches are logged in a file. In the next execution, the symbolic execution

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00096

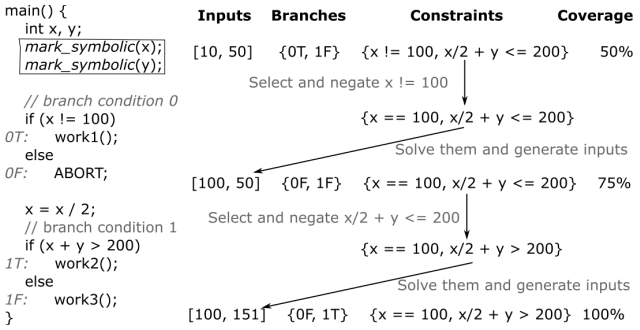

Figure 1. Concolic testing for a sequential C program.

component reads the log and generates new inputs for marked variables to potentially force a different execution path as follows: the constraint set is updatd by negating a selected constraint; and the updated constraint set is solved with the results yielding the new inputs.

Figure 1 shows how concolic testing applies to a sequential program. We denote a branch as $[condition\_id][T/F]$, where $condition\_id$ is the branch condition's unique ID and T/F represents `True` or `False` evaluation of the condition. On the left is a sequential program consisting of a pair of *branches*: ($0T$, $0F$) and ($1T$, $1F$) with a bug hidden at branch $0F$. Variables x and y are marked as symbolic. On the right is the process of concolic testing. At the start, the program is run with random inputs $\{x \leftarrow 10,\ y \leftarrow 50\}$, which uncovers branches $0T$ and $1F$ satisfying constraints $x \neq 100$ and $x/2 + y \leq 200$. To uncover a new branch, the testing tool negates $x \neq 100$ so that the updated constraint set is $\{x = 100,\ x/2 + y \leq 200\}$. It then generates the next inputs $\{x \leftarrow 100,\ y \leftarrow 50\}$ by solving the updated constraints. The inputs force the execution of $0F$ and thus trigger a bug. The testing logs such error-inducing inputs for further bug analysis. As the testing continues, it can derive new inputs and force the execution of $1T$. Finally, 100% branch coverage is achieved.

B. Challenges for MPI Application

Typical SPMD (single program, multiple data) MPI programs usually consist of the following steps: read inputs, check the validity of inputs — known as *sanity check*, distribute workloads across processes, and finally solve the problem based on a loop-based solver. Figure 2 shows the code skeleton of a such program where the inputs x and y from the user are first read (the reads are omitted for brevity), a sanity check is performed on x and y as well as their combination $x * y$, the work is shared and finally the *while* loop solves the problem. When applying concolic testing to such programs, we encounter two challenges described next.

First, standard concolic testing that only tests one process is not sufficient for MPI applications that run with multiple processes. It cannot deal with MPI semantics including *MPI rank* (a process' unique ID) and the number of processes. Hence, it fails to uncover branches related to such MPI semantics. Suppose concolic testing is only performed on

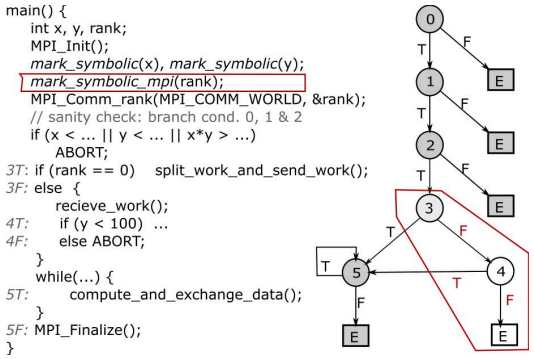

Figure 2. An MPI program's code skeleton and its execution tree.

process 0 for the program in Figure 2. During execution, branches $3F$ and $4T$ are encountered only by processes different from process 0, $4F$ is not encountered, and the remaining are uncovered by process 0. The testing fails to uncover $3F$ and $4T$ as it does not record branches uncovered by processes other than process 0; it does not uncover $4F$ as it does not test processes other than process 0 to satisfy both $rank \neq 0$ and $y \geq 100$. Besides the above missed branches, it should be noted that the testing can not uncover branches that can only be executed once a certain number of processes are used as its ignorance of MPI semantics makes it unable to vary the number of processes.

In addition, concolic testing could be impractical for MPI applications without carefully controlling the testing cost. This could results from three potential sources. First, too large an input can make the testing extremely slow and sometimes even fail as the memory needed could exceed the computing platform's memory limit. Second, running all processes using the same heavy-weight instrumentation incurs unnecessarily high overhead as not all processes need to perform symbolic execution. Third, too much effort is wasted in the presence of loops that characterize MPI applications as loops lead to too many redundant constraints being generated and solving as well as testing with them does not help to boost branch coverage.

C. Our Solution: COMPI

To address the above issues, this paper presents COMPI — a practical concolic testing tool to automate the testing of MPI applications. It is implemented on top of *CREST* [37], a scalable open-source concolic testing tool for C programs that replaces CUTE (one of the first implementations of concolic testing) [35]. COMPI supports testing of SPMD MPI programs written in C. It exposes bugs that result in assertion violation, segmentation fault, or infinite loops. It is able to tackle MPI semantics, uncovering branches that cannot be uncovered by standard concolic testing, by employing the following strategies: (1) it records branch coverages across all processes instead of just the one used to generate inputs; (2) it automatically determines the number of processes used in the testing as well as which process' execution should be used to generate the inputs to guide iterative testing. For the

program in Figure 2, strategy (1) helps uncover $3F$ and $4T$, and strategy (2) helps uncover $4F$. It curtails testing costs via three simple yet effective techniques: (1) input capping — allowing developers to cap the values of marked variables so as to limit the problem size and control the testing time cost; (2) two-way instrumentation — generating two versions of the target program with one being heavily-instrumented to be used by one single process and the other being lightly-instrumented to be used by the other processes; and (3) constraint set reduction — reducing the constraint sets by removing redundant constraints resulting in the presence of loops. COMPI makes the following key contributions.

- COMPI is the first practical automated testing tool for *complex* MPI applications — it tackles basic MPI semantics and effectively controls the testing cost.
- COMPI uncovered four new bugs in one physics simulation program that were confirmed by the developers.
- In our experiments COMPI achieved 69-86% branch coverage which far exceeds the 1.8-38% coverage achieved by random testing.
- COMPI exploits MPI semantics causing it to achieve 4.8-81% higher coverage than standard concolic testing.
- COMPI achieves high branch coverages quicker with input capping delivering practical testing; it reduces testing time by up to 66% via two-way instrumentation; it achieves 4.7-10.6% more coverage for two programs and achieves the best coverage much faster for another with constraint set reduction than without it.

II. Overview of COMPI

A. Work Flow of COMPI

The work flow of COMPI consists of two phases: (1) in the instrumentation phase, COMPI inserts symbolic execution code into the source code; and (2) during the testing phase, COMPI iteratively tests the program to potentially cover new branches via automatic input generation.

Instrumentation. Given a program, developers need to mark the execution-path dominant input-taking variables. Then COMPI instruments the program so as to insert symbolic execution code. In the instrumentation, COMPI marks MPI-semantics variables that represent MPI rank or the size of `MPI_COMM_WORLD` (the number of processes) so that these variables' values for the next test could be derived like other variables' input values. Figure 2 illustrates the marking of one MPI program — $rank$ is marked by COMPI and variable x and y are marked manually by developers.

Testing. COMPI performs an *iterative* testing procedure until a user-specified budget of iterations (executions of the program under test) is exhausted. In each iteration, it first determines the number of processes, as well as which process should be used to perform concolic testing so as to generate inputs to drive the next test — we call this process **focus** and the remaining processes as **non-focus**. In the first iteration, the number of processes and the focus process can be set by the developer, and all other symbolic variables are assigned random values; in future iterations, all the values are generated based on previous iteration. In each iteration, the instrumentation code generates branch coverage information and a set of constraints via executing the program. COMPI updates the coverage information. It updates the constraint set by selecting and negating one of the constraints, and then generates new inputs by solving the updated constraint set. With the new inputs, it drives the testing in the next iteration.

Highlights of COMPI. In summary, COMPI extends CREST with the following two critical features:

- It provides an automated testing framework specifically for MPI programs — it performs symbolic execution on a single focus process and records branch coverage across all processes. Due to its knowledge of MPI semantics, it automatically drives the testing by varying the number of processes as well as the focus process. Recording coverage across all processes makes sure the overall coverage is recorded accurately.
- It enables practical testing via effectively controlling the testing cost based on three techniques: input capping, two-way instrumentation, and constraint set reduction.

Figure 3 illustrates the iterative testing of COMPI from the i-th test to the $(i+1)$-th test on the program given in Figure 2. Suppose after Step 2 of the i-th test, only branch $4F$ is left, and in Step 3 the constraint $rank = 0$ is negated. Supposedly $rank \leftarrow 1$ is obtained from the constraint solver. Hence, in the $(i+1)$-th test COMPI shifts its focus from rank 0 to rank 1. With this focus change, COMPI can uncover the branch $4F$ in a future test.

B. Search Strategy Selection

The decision on which constraint to negate (and thus which path to explore next) is made according to the *search strategy*. There are four strategies available in CREST: **BoundedDFS**, random branch search, uniform random search, and control flow graph (CFG) search. BoundedDFS allows users to specify a *depth bound* and thus can skip branches deeper than the bound, which is better than DFS as it avoids exploring infinitely deep execution tree. Random branch search and uniform random search randomly select a branch to negate, and CFG search selects the branch based on a scoring system that checks the distance between the covered branches and uncovered branches.

BoundedDFS is a classical search strategy that is slow yet *steady* [35] and it matches the need of MPI programs much better than the others because of the major difference between MPI applications and regular ones: MPI programs usually read many inputs and thus need to perform a **sanity check** before entering the *solver* to ensure the validity of inputs (see Figure 2). The sanity check can consist of many conditional statements, and only by passing all the checks

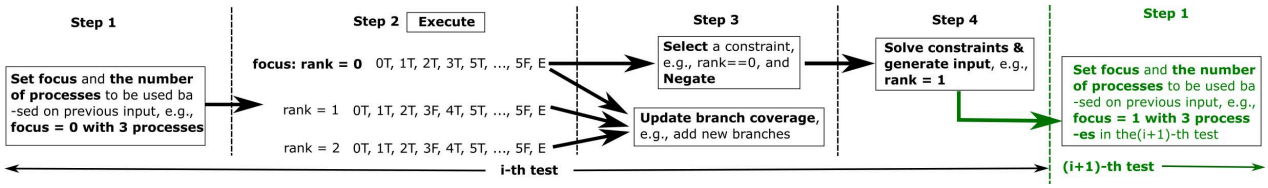

Figure 3. The iterative testing of COMPI: i-th test to $(i+1)$-th test (marked in green).

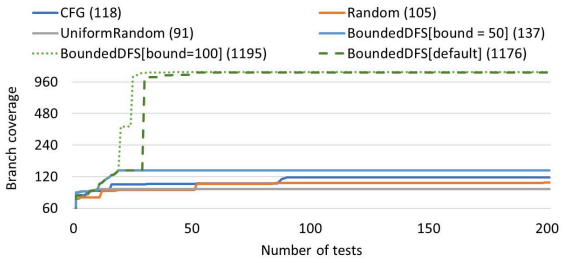

Figure 4. Branch coverage of HPL using four search strategies.

can the program enter the solving phase. BoundedDFS is very effective in passing the sanity check as it systematically traverses the *execution tree* and aims to uncover all possible branches. The remaining strategies are ineffective as they do not search branches in the order by which they are ordered in an execution path. Consider an example based on the execution tree of Figure 2. Suppose the current execution path is $0T \rightarrow 1T \rightarrow 2F$ with all the branches above $2T$ being covered already. These strategies may not take the required step (take $2T$ by negating $2F$) and rather take $0F$ by negating $0T$, and thus they fail to pass the check. This is very common, especially for a complex sanity check. Even if they pass the check, they can deteriorate to the limited path in sanity check due to the same reason.

Let's consider High-Performance Linpack Benchmark (HPL) [3] is one of the most widely used HPC benchmarks. It performs highly optimized LU factorization and has 28 input parameters that include variables and arrays — we treat each array as one regular variable. In its sanity check, each parameter as well as the combinations of parameters are checked. Figure 4 shows its branch coverage comparison for four strategies using COMPI. BoundedDFS with default depth of 1,000,000 and BoundedDFS with bound equal to 100 perform the best with a coverage of over 1100 branches while the others cover at most 137 branches as they fail to pass the sanity check. This shows that a bad bound selection results in poor branch coverage and non-systematic strategies are unable to pass the sanity check.

BoundedDFS for COMPI. To ensure a good choice of the bound for BoundedDFS, COMPI's testing consists of two phases: (1) it uses DFS first so that the maximal size of the constraint set (the longest execution path) can be observed; and (2) it uses BoundedDFS in the remaining iterations with the bound being slightly bigger than the observed considering longer execution path might be observed later. In this way, COMPI has one full execution tree in its sight.

Table I
MPI SEMANTICS RELATED VARIABLES.

Symbol	Meaning
r_w	Variables denoting global rank in `MPI_COMM_WORLD`
r_c	Variables denoting local rank in other communicators
s_w	Variables denoting the size of `MPI_COMM_WORLD`

III. FRAMEWORK ADAPTATION

The framework of COMPI can be summarized as *one focus* and *all recorders*, i.e., it drives the testing with one focus process and accurately tracks the branch coverage across all. One focus is the basic requirement for a concolic testing tool, and all recorders are needed specifically for MPI programs considering that otherwise only recording the coverage of the focus process is not accurate as it misses the branches already being uncovered by non-focus processes. To enable automated testing for MPI programs, we automate the selection of the focus as well as the determination of the number of processes to be used using concolic execution. The framework consists of 4 major aspects: (1) automatic marking, (2) MPI-semantics constraints insertion, (3) conflicts resolving, and (4) test setup and program launching.

A. Automatic Marking

To make the symbolic execution logic recognize important MPI semantics, COMPI automatically marks r_w, r_c and s_w shown in Table I as symbolic. Application developers mark regular input variables manually with trivial effort as these usually cluster together and read inputs at the beginning of the program from either a user-specified file or a command line. Variables including r_w, r_c and s_w do not have to cluster together considering they obtain their values anytime from MPI environment. Since manually marking them is laborious, COMPI automatically marks them in the instrumentation phase. At each invocation of

`MPI_Comm_rank(`*comm*`,` *rank*`)`,

COMPI marks *rank* as a r_w if *comm* is checked to be a constant as `MPI_COMM_WORLD` is a constant in MPI semantics; otherwise, *rank* is marked as a r_c. At each invocation of

`MPI_Comm_size(`*comm*`,` *size*`)`,

COMPI marks *size* as a s_w if *comm* is found to be a constant. So far COMPI does not mark variables representing the size of communicators other than the default.

B. Constraints Insertion

The inherent relations among r_w, r_c and s_w should be obeyed by the constraint solver, e.g., one global rank must be smaller than the size of `MPI_COMM_WORLD` ($r_w < s_w$).

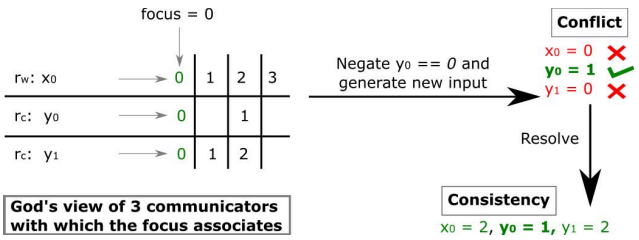

Figure 5. Resolving the conflicts among r_w and r_c variables by using the most up-to-date values. Each row in the left table maps to one communicator, and each column maps to one process.

Without knowing these, the solver can generate invalid inputs, e.g., $r_w \geq s_w$. It is thus necessary to inform the solver these inherent relations, i.e., add the *inherent MPI-semantics related constraints* to the constraint set to be solved. Suppose there are m variables of type r_w — each is represented symbolically as x_i with $0 \leq i < m$, n variables of type r_c — each is represented as y_i with $0 \leq i < n$, and k variables of type s_w — each is represented as z_i with $0 \leq i < k$. As the focus process drives the testing, we need to generate these MPI inherent constraints from the perspective of the focus considering it may only associate with some of the non-default communicators. We summarize these inherent constraints as the union of the following:

$$\begin{cases} \bigcup_{i=1}^{m} \{x_0 - x_i = 0\} \\ \bigcup_{i=1}^{k} \{z_0 - z_i = 0\} \\ \quad\quad \{x_0 - z_0 < 0\} \\ \bigcup_{i=0}^{n} \{y_i - s_i < 0 \mid 0 < i < n\} \\ \bigcup_{i=0}^{n} \{y_i \geq 0\} \bigcup \{x_0 \geq 0\} \bigcup \{z_0 > 0\} \end{cases}$$

where the first specifies the equivalence of all r_w variables representing the focus's global rank, the second specifies the equivalence of all s_w variables representing the default communicator's size, the third specifies the relation between the global rank and the default communicator's size, the fourth specifies the relation between the local rank and non-default communicators' size s_i $(0 < i < n)$, where s_i is a *concrete value* obtained by the instrumentation code at runtime, and the last specifies that the size of the default communicator should be no less than 1 and any of the others should be no less than 0.

C. Conflicts Resolving

The above constraints are not complete as the relation between local ranks and global ranks is not included. The solver thus could generate conflicting constraints — the generated input values for various variables denoting MPI ranks don't map to the same process. Figure 5 shows an example. Suppose there are 3 processes in total with the focus being process 0 (global rank). The focus process resides in `MPI_COMM_WORLD` as well as two local communicators, and x_0, y_0 and y_1 respectively record the rank of the focus in each communicator. Starting with an input (0, 0, 0) for (x_0, y_0, y_1), COMPI supposedly negates $y_0 = 0$ and generates input values in conflict as (0, 1, 0) — $x_0 = 0$ and $y_1 = 0$ map to global rank 0 but $y_0 = 1$ maps to global rank 2. We resolve the conflicts based on the following important property of the underlying constraint solver.

Incremental solving property. Solving the *whole* constraint set every time is time consuming. *Incremental solving* is thus proposed an efficient strategy based on the iterative tests' property — two constraint sets being solved consecutively usually share many common constraints. It works in following way: (1) it only solves *incremental constraints* — the *negated* constraint as well as the constraints dependent upon it, and (2) it assigns old values from the previous inputs to variables not being solved. We find an useful **property**: if the value of one variable read from the solver is different from its previous reading, its value is more *up-to-date* compared with those whose values stay the same.

Conflict resolving. Because of the presented property, we resolve the potential conflicts by using the *most up-to-date* values among r_w and r_c since they satisfy the negated constraints while stale values don't. As shown in Figure 5, only y_0 is updated and thus is the most up-to-date value. The conflicting values are corrected using y_0, so they map to the same process, i.e., global rank 2. Note this resolving method assumes that the r_c and r_w variables are not dependent, which does make sense as one constraint involving both (MPI ranks) doesn't map to a realistic meaning.

D. Test Setup and Program Launching

In the iterative testing, we launch the current test by feeding the inputs generated from the previous test. However, the value passing phase of r_w, r_c and s_w differs from that of regular input variables: the former has to take place in the test setup phase to guide the program launching while the latter occurs at runtime, which is due to the fact that the values of r_w, r_c and s_w are fixed when the program is launched, e.g., global rank can't be changed at runtime.

Test setup consists of two parts: determine the number of processes used to launch the program, and select focus process. The number of processes is set as the derived value for s_w. To set the focus, we need to find the global rank of the focus as it is the key to launch the program. Based on the presented property, the focus stays unchanged if there is not any value change among r_w and r_c; otherwise, the focus's global rank should be derived based on the value change. *When r_w changes, its new value is the focus' new global rank; otherwise (r_c changes), the case is trickier as the new value of r_c doesn't directly translate to a global rank.* To solve this problem, COMPI builds a *mapping data structure* between local ranks and global ranks at runtime — a two dimensional array with each row storing all the global ranks belonging to one local communicator by the increasing order of local MPI ranks. Given a local rank with its communicator's index known, its mapped global rank can be easily retrieved. Table II illustrates the mapping

Table II
THE MAPPING BETWEEN LOCAL RANKS AND GLOBAL RANKS.

Sorted local ranks →		0	1	2	3	4	5
Global ranks →	Local Comm. 0	0	4	2	-	-	-
	Local Comm. 1	0	3	-	-	-	-
	...	...	...	...	...	...	...

array from the perspective of the focus (global rank 0) given five processes in `MPI_COMM_WORLD`. There are three global ranks (0, 4, and 2) in local communicator 0 and two global ranks (0 and 3) in local communicator 1. Suppose we hope to access the global rank of *local rank 1* in *local communicator 0*. The global rank can be obtained as $mapping[0][1] = 4$.

Program launching. The instrumentation generates two copies of programs: *ex1* and *ex2*, where the former is used to launch the focus process and the latter is used to launch the remaining. COMPI runs the given SPMD program in a MPMD (multiple program, multiple data) style. Suppose the focus' global rank is i and the total number of processes to run the program is s. We launch the program with

`mpiexec -n 1 ./ex1 : -n` $s-1$ `./ex2`

if $i = 0$; otherwise, we launch it with

`mpiexec -n` $i-1$ `./ex2 : -n 1 ./ex1 : -n` $s-i$ `./ex2`

By default, global ranks are assigned by the order in launching processes. We hence can shift the focus by varying i, and vary the number of processes by varying s.

IV. PRACTICAL TESTING

The tool would not be practical to be adopted without seeking every means to reduce its testing cost. Below we detail three major techniques to reduce the test cost.

A. Input Capping

Usually MPI programs are designed to be capable of solving various problems sizes. Given a fixed number of parallel processes, the larger the problem size is the more time-consuming the testing is though very often varying problem sizes lead to very similar coverages. Take HPL for example. We respectively run it at various *matrix size* (the width of a square matrix) 100, 200, ..., 1000 while maintaining all other inputs as default (see Figure 6). Except for the small coverage increase from matrix width 100 to 200 the coverage almost stays the same from 200 to 1000. However, the execution time cost at matrix width 1000 is 27.2 times the cost at 200. Most importantly, too large an input value can make the testing fail. This manifests in two ways: (1) too large a problem size might exceed the testing platform's memory limit; and (2) way too many processes can crash the platform, e.g., once our rudimentary COMPI made the computer freeze when it demanded hundreds of thousands of processes to run the program.

To avoid unnecessary time-consuming tests, COMPI provides additional marking interfaces to allow developers to specify a cap for the input variable that plays a pivotal role

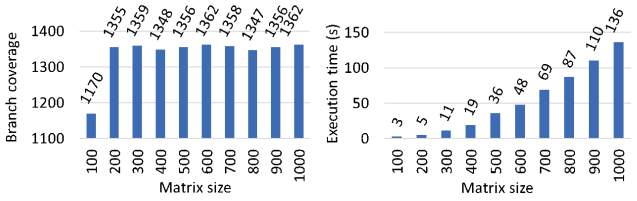

Figure 6. The achieved branch coverage as well as the time cost at various matrix sizes for HPL.

on determining the execution time cost. Take the marking of an *int* variable for example. It can be marked as

`COMPI_int_with_limit(int` x`, int` *cap*`)`,

where the *cap* is the upper bound for variable x. COMPI would generate the symbolic constraint $x \leq cap$ and feed it to the solver as shown in Section III-B.

B. Two-way Instrumentation

The instrumentation code performs symbolic execution at runtime. After executing the program, each process outputs collected symbolic execution information (symbolic constraints, branch coverage, inputs, etc.) to a file, which will be read by COMPI to drive the next test. With very little effort, we can enable concolic testing for MPI programs based on *one-way instrumentation* — all processes run with the same instrumented program. However, this effort-saving way is not efficient due to two reasons: (1) it brings about unnecessary *memory overhead at runtime* for non-focus processes since these perform unnecessary symbolic execution though they only care about recording branch coverage; and (2) it brings about much unnecessary I/O overhead for non-focus processes considering I/O on data unrelated to coverage is not useful for the testing framework.

Hence we propose two-way instrumentation: (1) the program (*ex1*) used to launch the focus process is instrumented heavily — each expression is instrumented — to enable full symbolic execution in each concrete run; and (2) the program (*ex2*) used to launch the non-focus processes is instrumented lightly — only branches are instrumented — to only record the branch IDs being uncovered. This differentiating style minimizes the workload for non-focus processes and makes testing efficient.

C. Constraint Set Reduction

Loops characterize MPI programs and cause hundreds and thousands of reducible constraints generated from the same branch. They thus cause a significant waste of testing efforts on the repetitive branches. For example, as shown in Figure 7 at least 101 constraints can be generated from one loop's execution — the constraint set size could be far greater considering function `do_A()` could also contain branches. Repetitive tests over `if(`$x < 100$`)` simply waste time as the first constraint $x < 100$ subsumes the remaining but the last one, i.e.,

$$\{x \mid x + i < 100 \text{ and } 0 < i < 100\} \subset \{x \mid x < 100\}.$$

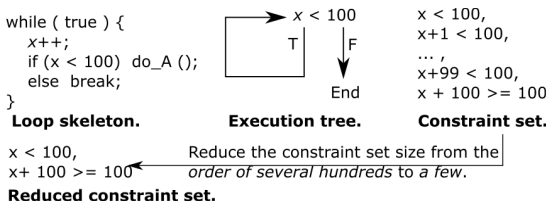

Figure 7. Constraint set reduction given x is marked as symbolic and $x = 0$ before entering the loop.

We avoid such unnecessary tests via a heuristic based on the property of reducible constraints as shown following.

Property of reducible constraints. Given a time-ordered sequence of constraints generated by one single conditional statement in one non-nested loop at runtime. All constraints except the last one evaluate to `True` (or `False`), and the last constraint evaluates to `False` (or `True`).

Constraint set reduction. Based on the property, we reduce the number of constraints generated by each conditional statement using following heuristics. At runtime, a constraint is recorded only if (1) this conditional statement is encountered for the first time or (2) its evaluated *boolean* value is the opposite of the previous observed value.

V. Implementation

COMPI is implemented on top of CREST that consists of four main parts: an instrumentation module, an execution library, and a search strategy framework, and a constraint solver based on Yices SMT (satisfiability modulo theories) solver [5]. COMPI's work spreads across all four. COMPI's implementation is based on over 3500 lines of C++/Ocaml code changes – 1436 lines of CREST were modified and 2151 new lines of code were added. COMPI is publicly available at https://github.com/westwind2013/compi.

Instrumentation is performed using CIL (C Intermediate Language) [38] under the guidance of an instrumentation module written in *OCaml* [6]. COMPI provides two separate OCaml instrumentation modules to achieve two-way instrumentation. Both modules instrument MPI_Init(), MPI_Comm_rank() and MPI_Comm_size() so as to equip COMPI with basic MPI knowledge. Only one module instruments programs heavily by inserting the symbolic execution code, while the other instruments only programs' branches to help non-focus processes record coverage.

Concolic execution library defines all instrumentation functions. The major new features of COMPI include following: (1) it provides separate instrumentation functions for the program used by non-focus process; (2) it defines additional marking functions to achieve input capping; and (3) it implements the constraint set reduction technique.

Search strategy framework is the brain of COMPI as it directs the testing. Particularly, COMPI selects the focus as well as sets the number of processes based on derived input values before the program launching. Additionally, COMPI allows developers to specify a timeout for a test. It logs the derived error-inducing input for further analysis if either the program returns a non-zero value or fails to complete within the specified timeout.

Table III
Complexity of Target Programs.

Program ↓	SLOC ↓	The number of branches	
		Total	Reachable
SUSY-HMC	19,201	2,870	2030
HPL	15,699	3,754	3,468
IMB-MPI1	7,092	1,290	1,114

Constraint solver solves the constraint sets. COMPI creates additional constraints based on MPI semantics and input capping and insert them to the set before solving.

VI. Evaluation

We detail the newly uncovered bugs first and then evaluate four major features of COMPI: input capping, two-way instrumentation, constraint set reduction, and framework. Each feature is evaluated by comparing the *default* COMPI with its *variation* that either modifies or disables the feature of interest while incorporating all the other features.

Target programs: Table III shows the three target programs we use to evaluate COMPI: (1) *SUSY-HMC*, a major component in SUSY LATTICE — a physics simulation program performing Rational Hybrid Monte Carlo simulations of extended-supersymmetric Yang–Mills theories in four dimensions [39]; (2) *HPL* (High-Performance Linpack Benchmark) used for solving a dense linear system via LU factorization; (3) *IMB-MPI1*, which is one major component of IMB (Intel MPI benchmarks) and can benchmark MPI-1 functions' performance. Table III also shows the code complexity in different metrics: the source lines of code (SLOC) measured by SLOCCount [7]; the total number of branches obtained in the instrumentation phase via static analysis; and the estimated number of *reachable* branches obtained via summing up all the branches of all the encountered functions in testing [8]. We use the reachable branches to evaluate our coverage as some of branches found by static analysis are not reachable due to build configurations [8].

Marking input variables: The users of COMPI must mark a subset of input variables – these are non-floating point inputs as COMPI does not handle floating-point variables. The effort required is minimal. Respectively, we marked 13 variables in SUSY-HMC, 24 variables in HPL, and 15 variables in IMB-MPI1. For illustration we describe one relevant input for each program: (1) the *lattice size* of each of the four dimensions in SUSY-HMC — we change the four as well as set input caps for them with the same value; (2) the *width* of the square matrix in HPL; and (3) the *number of iterations* required to benchmark one function's performance in IMB-MPI1. We denote these as N.

Experiment setup: We perform experiments on a platform that is equipped with two Intel E5607 CPUs (totaling 8 cores) and 32 GB memory. Initially, 8 processes are used to launch the program with the focus being process 0. The number of processes is restricted to no bigger than 16 via input capping. Suppose each *test* consists of n iterations. To

be consistent as directed in Section II-B, in each test we use pure DFS in the first x iterations — $x = 50$ for SUSY-HMC, $x = 1000$ for HPL and IMB-MPI1; we use BoundedDFS afterwards for the remaining $n - x$ ($n > x$) iterations — the depth limits are 500 for SUSY-HMC, 600 for HPL, and 300 for IMB-MPI1 (estimated based on the constraint set sizes in the first phase). Unless otherwise specified, the *default caps* of the introduced input variable N are: (1) $N_C = 5$ for SUSY-HMC, (2) $N_C = 300$ for HPL and (3) $N_C = 100$ for IMB-MPI1. Sometimes the testing can be constrained to a very short shallow path in the execution tree due to an error that is lacking a constraint for tackling it. Once this error is encountered, like bugs in SUSY-HMC, concolic testing can not step out of this error as its constraint-based derivation is broken. Using tens of tests that only costs a few seconds this can be found easily if the constraint set size is too small. We just redo the testing to avoid it. In practice, developers should fix such known bugs and then continue testing for uncovering additional bugs.

A. Uncovered Bugs

The use of COMPI on the programs detected *four bugs* in SUSY-HMC, where three cause segmentation faults [1] and one causes a floating point exception [2].

The segmentation fault occurs due to wrong use of `malloc()`. Take one bug for example. The program declares a double pointer `src` and allocates space for it:

`Twist_Fermion` $**src = malloc(Nroot * sizeof(**src))$;

where `Twist_Fermion` is a struct and $Nroot$ is an integer denoting the number of elements the allocated space would hold. Variable src expects the space allocation to store $Nroot$ `Twist_Fermion`$*$ elements, but the above allocates space to store $Nroot$ `Twist_Fermion` elements. This causes a program crash due to a segmentation fault. This can be easily fixed by changing $sizeof(**src)$ to $sizeof($`Twist_Fermion`$*)$. COMPI detects three bugs due to this error. We reported these bugs and the fix to the developer, who confirmed them and adopted our fix.

The floating point exception bug is a more serious one. It leads to a division-by-zero error whose triggering requires not only specific input values but also a specific number of processes in the run — it manifests with 2 or 4 processes but it does not occur with 1 or 3 processes. We provided the triggering condition generated by COMPI to the developer and he was easily able to reproduce the bug and then fix it.

B. Input Capping

We compare the testing cost using various input caps. Each cap is evaluated using 10 times of testing with each containing 50 iterations for SUSY-HMC and 500 iterations for both HPL and IMB-MPI1, which are enough to show the time cost variance on the basis of a decent coverage is achieved, i.e., the testing passes the programs' sanity check. Figure 8 shows the testing time and the coverage comparison using different caps. For SUSY-HMC, the average time

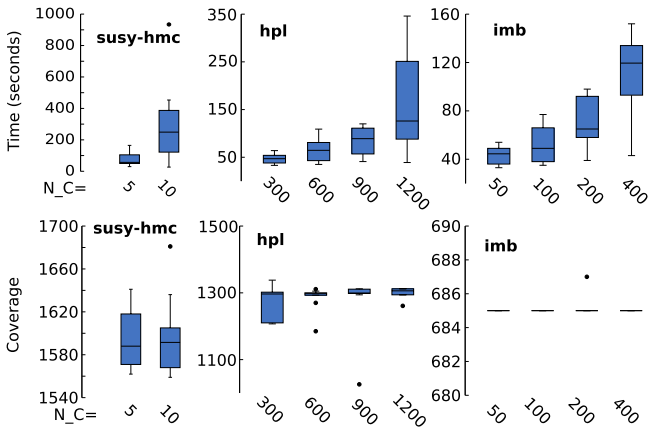

Figure 8. Evaluation of input capping.

increases by four times as N_C increases from 5 to 10 while the coverages using two caps are comparable. For HPL, the coverage ranges from about 1100 to 1300 (such variance can occur even for the same cap size), and when $N_C = 1200$ the testing time cost in the worst case is about seven times of the cost when $N_C = 300$. For IMB-MPI1, the average cost increases by four times as N_C increases from 50 to 400 while always about 685 branches are discovered. Obviously, bigger caps lead to more expensive testing cost on the basis of providing comparable coverages. Without it the concolic testing is never possible.

C. Two-way Instrumentation

COMPI using two-way instrumentation is compared with its variation that uses one-way instrumentation based on simulated testing that fixes the inputs to defaults for each program (the dynamic derivation of input values is disabled). The time cost is fixed and thus the comparison reflects only the difference between instrumentations. Each configuration is evaluated using one 10-iteration test. Table IV shows the testing cost comparison of two instrumentation methods given different input values. Two-way instrumentation saves over 47% testing time for SUSY-HMC, over 62% for HPL, and 0-12.5% for IMB-MPI1. Also Table IV shows the average size of non-focus processes' log files — the I/O between the target program and COMPI. Using two-way instrumentation non-focus processes only output a few kilobytes while using one-way instrumentation the log size could be as high as a few hundred megabytes. Moreover, the trivial log file size indicates that non-focus processes don't eat too much memory at runtime as they do not

Table IV
ONE-WAY VS. TWO-WAY

Program	N	Time cost (seconds)			Avg. log size (B)	
		1-way	2-way	Saving	1-way	2-way
SUSY-HMC	2	163	86	47.0%	104M	6.4K
	4	479	226	52.8%	337M	6.4K
HPL	300	92	35	62.0%	71.1M	4.5K
	600	382	127	66.8%	261.8M	4.5K
IMB-MPI1	100	7	7	0.0%	562.0K	1.9K
	400	16	14	12.5%	1.8M	1.9K
	1600	43	38	11.6%	5.5M	1.9K

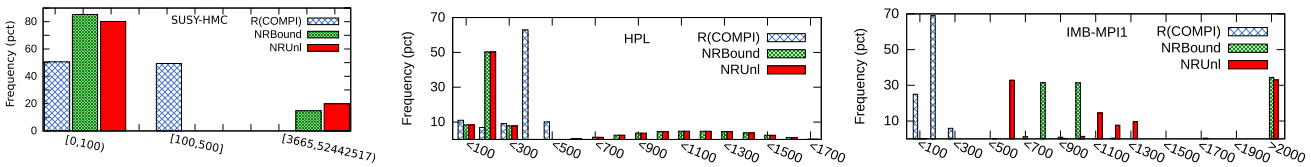

Figure 9. Constraint set size distribution for SUSY-HMC, HPL and IMB-MPI1.

Table V
EVALUATION OF CONSTRAINT SET REDUCTION.

Program ↓	COMPI (R)		NRBound		NRUnl	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
SUSY-HMC	84.7%	86.1%	80.0%	82.0%	80.1%	80.2%
HPL	69.6%	71.9%	59.0%	59.6%	59.4%	60.4%
IMB-MPI1	69.0%	69.1%	69.0%	69.1%	69.0%	69.0%

Table VI
COMPI'S COVERAGE RATE.

Program ↓	COMPI (Fwk)		No_Fwk		Random	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
SUSY-HMC	84.7%	86.1%	3.4%	3.5%	38.3%	38.3%
HPL	69.4%	71.6%	58.9%	59.1%	2.2%	2.2%
IMB-MPI1	69.0%	69.1%	64.2%	64.3%	1.8%	1.8%

need to perform tasks other than executing the program and recording the branch coverage information.

D. Constraint Set Reduction

We evaluate constraint set reduction by comparing COMPI with reduction (R) with its two variations: non-reduction with a depth limit (NRBound) (the same to COMPI's default depth limit for each program) and non-reduction with unlimited depth (NRUnl). To perform a fair comparison, we apply COMPI (R), NRBound and NRUnl to each program based on a fixed time budget. The time budget of each test experiment is set to match the time taken by COMPI (R) to achieve the maximum attainable coverage. The durations are 1.5 hours for SUSY-HMC, 3.5 hours for HPL, and 34 minutes for IMB-MPI1. The reported results are based upon three repetitions of each experiment.

SUSY-HMC: As shown in Table V, R in average achieves about 4.6% more coverage than NRBound and NRUnl. Also we notice that sometimes both NRBound and NRUnl need to spend tens of minutes to derive a set of inputs. This occurs due to two reasons: too many redundant constraints are generated and negating these makes the constraint set insolvable. Figure 9 shows that our reduction technique generates constraint sets whose size are always smaller than 500, but without using it the constraint set could be as large as a few thousands to tens of millions.

HPL: Based on the average coverage, we observe following: (1) R achieves respectively 10.6% and 10.2% more coverage than NRBound and NRUnl; (2) all three achieve about 59% coverage (the maximum of NRUnl) in three minutes; (3) In the remaining time of over three hours, NRBound's and NRUnl's coverages stay the same as the coverage in the first three minutes, let alone get any closer to R's coverage. This results from the fact that the non-reduction methods spend a significant portion of time traversing redundant branches. Figure 9 shows that our reduction technique significantly reduces the constraint set size — R's maximal size is about 500 but the size for other two can be over 1600.

IMB-MPI1: All of them achieve equivalent coverages with a difference of only 1 or 2 branches — the average coverage rate is 69.0%. The required time to achieve the minimum of all methods' maximal coverages, i.e., 767 branches, are respectively: (1) 116s, 64s and 386s for R; (2) 257s, 279s and 966s for NRBound; and (3) 226s, 286s and 4433s for NRUnl. By excluding the outliers 966s and 4433s — their occurrences are related to the randomness feature of COMPI, the average time costs to uncover 767 branches are respectively 189s, 268s and 256s. Most importantly, Figure 9 shows that R generates less than 300 constraint in testing while the other two generate more than 2,000 constraints in over 30% testing iterations.

E. COMPI Framework and Random Testing

We evaluate the effectiveness of COMPI's framework by comparing COMPI with the framework enabled (Fwk, COMPI itself) with its variation with the framework disabled (No_Fwk) — No_Fwk drives the testing using only one fixed focus process, records the coverage of this process only, and always uses 8 processes (the initial setting of COMPI). we apply COMPI (Fwk) and No_Fwk to each program based on a fixed time budget as used in Section VI-D. The reported results are based on three repetitions of each experiment. As No_Fwk doesn't vary the focus process, the above evaluation is performed on each process and the obtained branch coverage using each process are combined to form No_Fwk's final coverage. As shown in Table VI, for SUSY-HMC Fwk achieves an average coverage of 84.7% which is about 25 times the coverage of No_Fwk; for HPL Fwk achieves an average coverage of over 69% that is about 10% higher than No_Fwk; for IMB-MPI1, Fwk achieves 69% coverage that is about 5% higher than No_Fwk. We observe that No_Fwk performs far worse than Fwk only for SUSY-HMC because under the condition of using 8 processes persistently No_Fwk fails to generate sound inputs that exercise the full program. The effectiveness of our framework is hence obvious — it gives COMPI the freedom to vary not only the focus process but also the number of processes and this freedom helps COMPI achieve higher coverages.

We also compared the default COMPI with purely random testing (Random). Random testing generates random values for marked variables and randomly sets the number of processes used as well as the focus process. For a fair

comparison, all the random values are generated under the limits set by the input capping. We apply COMPI and `Random` to each application using the fixed time budgets as used in Section VI-D. The reported results are based on three repetitions of each experiment. As shown in Table VI, COMPI's coverage is over 2 times that of `Random`'s for SUSY-HMC, and it is over 30 times the coverage of `Random` for HPL and IMB-MPI1.

VII. Conclusion

We presented COMPI that automates the testing of MPI programs. In COMPI, MPI semantics guide testing using different processes and dynamically varying the number of processes used in testing. Its practicality is achieved by effectively controlling its testing cost. COMPI was evaluated using widely used complex MPI programs. It uncovered new bugs and achieved very high branch coverages.

VIII. Acknowledgement

This research is supported by the NSF grants CNS-1617424 and CCF-1524852. We thank Dr. David Schaich for reproducing and confirming the bugs uncovered by COMPI.

References

[1] Three segmentation fault bugs in SUSY-HMC. https://github.com/daschaich/susy/issues/15.

[2] Floating point exception in SUSY-HMC. https://github.com/daschaich/susy/issues/16.

[3] HPL: a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/.

[4] How do I run an MPMD MPI job? https://www.open-mpi.org/faq/?category=running#mpmd-run.

[5] The Yices SMT Solver. http://yices.csl.sri.com/.

[6] OCaml. https://ocaml.org/

[7] SlOCCount. https://www.dwheeler.com/sloccount/.

[8] https://github.com/jburnim/crest/wiki/CREST-Frequently-Asked-Questions.

[9] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *SC*, Article No. 51, 2000.

[10] J. Coyle, J. Hoekstra, G. R. Luecke, Y. Zou, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[11] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. Marmot: An mpi analysis and checking tool. In *PARCO*, pp. 493–500, 2003.

[12] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *ACM/IEEE SC*, Article No. 15, 2007.

[13] Z. Chen, Q. Gao, W. Zhang, and F. Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *ACM/IEEE SC*, 2010.

[14] Z. Chen, X. Li, J-Y. Chen, H. Zhong, and F. Qin. Syncchecker: Detecting synchronization errors between mpi applications and libraries. In *IEEE IPDPS*, pp. 342–353, May 2012.

[15] 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 *IEEE IPDPS*, pp. 1–10, March 2007.

[16] 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 *ACM/IEEE SC*, 2009.

[17] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. Ahn, and M. Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *IEEE/IFIP DSN*, 2010.

[18] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, B. Rountree. Large scale debugging of parallel tasks with AutomaDeD. In *SC*, 2011.

[19] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Diagnosis of performance faults in large scale mpi applications via probabilistic progress-dependence inference. In *IEEE TPDS*, 26(5):1280–1289, 2015.

[20] H. Li, Z. Chen, and R. Gupta. Parastack: Efficient Hang Detection for MPI Programs at Large Scale. In *ACM/IEEE SC*, Article No. 63, 2017.

[21] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz: Subgroup reproducible replay of mpi applications. In *ACM PPOPP*, 2009.

[22] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *ACM/IEEE SC*, 2015.

[23] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau. Noise injection techniques to expose subtle and unintended message races. In *PPOPP*, pp. 89–101, 2017.

[24] R. Vuduc, M. Schulz, D. Quinlan, B. De Supinski, and A. Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *Workshop on Parallel and Distributed Systems: Testing and Debugging*, pp. 27–36, 2006.

[25] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Conf. on Computer Systems*, pp. 183–198, 2011.

[26] C. Hovy and J. Kunkel. Towards automatic and flexible unit test generation for legacy hpc code. In *International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 1–8, Nov 2016.

[27] X. Fu, Z. Chen, H. Yu, C. Huang, W. Dong, and J. Wang. Symbolic execution of mpi programs. In *IEEE ICSE*, 2015.

[28] U. Kanewala and J. M. Bieman. Testing scientific software: A systematic literature review. In *Information and Software Technology*, vol. 56, no. 10, pp. 1219 – 1232, 2014.

[29] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems J.*, 22(3):229–245, 1983.

[30] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. In *SP&E*, 34(11):1025-1050, 2004.

[31] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *USENIX Windows System Symposium*, pp. 59–68, 2000.

[32] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, 2000.

[33] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *IEEE ICSE*, pp. 326–335, 2004.

[34] J. C. King. Symbolic execution and program testing. In *Comm. of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[35] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *ESEC/FSE-13*, pp. 263–272, 2005.

[36] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, pp. 213–223, 2005.

[37] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *IEEE/ACM ASE*, pp. 443–446, 2008.

[38] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pp. 209–265, 2002.

[39] D. Schaich and T. DeGrand. Parallel software for lattice N= 4 supersymmetric yang–mills theory. In *Computer Physics Communications*, vol. 190, pp. 200–212, 2015.