UNIVERSITY OF CALIFORNIA
RIVERSIDE

Pre- and Post-Deployment Dynamic Bug Detection Techniques for MPI Programs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hongbo Li

September 2018

Dissertation Committee:

Dr. Rajiv Gupta, Co-Chairperson
Dr. Zizhong Chen, Co-Chairperson
Dr. Philip Brisk
Dr. Zhijia Zhao

The Dissertation of Hongbo Li is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

This dissertation would not have been possible without the generous help of various kinds I received from my advisors, professors, lab-mates and friends, and my family.

I would like to express my sincere gratitude to Prof. Rajiv Gupta for leading me to this milestone. His critical suggestions bettered my research work. His tireless revising of my papers improved more than my writing and presentation skills. His extensive knowledge ensured this dissertation stay on the right track. His insightful vision helped me find simplicity out of clutter. His perpetual enthusiasm in research and hard-working spirit kept me motivated. Most importantly, his great understanding, patience, and tolerance helped me survive my lowest point. Thank you, Prof. Gupta!

I would like to express my sincere gratitude to Prof. Zizhong Chen for the generous support of the last five years. It is conference attending opportunities he gave that broadened my horizon. It is the freedom he provided that allowed me to find the most exciting project that transfers Software Engineering knowledge to High Performance Computing. It is his encouragement, perseverance, and valuable suggestions that turned naive ideas into decent research with years of honing. Also I appreciate his great understanding, patience, and tolerance for all these years. Thank you, Prof. Chen!

I would like to thank my committee members Prof. Philip Brisk and Prof. Zhijia Zhao for their valuable feedback and support. I would like to thank Ms. Kelly Downey for having me work as her teaching assistant for one full year.

I would like to like to acknowledge the support of National Science Foundations via grants CCF-1318103, CCF-1513201, CCF-1524852, CNS-1617424, and OAC-1305624. I

also would like to acknowledge the support of the MOST key project 2017YFB0202100 and the SZSTI basic research program JCYJ20150630114942313.

I would like to thank my lab-mates and friends for helping me in various ways: Zachary Benavides, Jieyang Chen, Longxiang Chen, Sihuan Li, Yuanlai Liu, Xin Liang, Kaiming Ouyang, Li Tan, Dingwen Tao, Keval Vora, Panruo Wu, Chengshuo Xu, and Keli Zhang. I am grateful for the joyful chats we had, encouraging words you gave, the basketball games we played, the delicious food we had, and the time we fought hard together as a team for paper submissions.

Words cannot express my gratitude to my parents and my sister. Without their unconditional love and unflagging support, I could not have gone this far. I am also indebted and grateful to my parent-in-laws for treating me like their own son. Last but not least, I would like to thank my better half, Shangjie, for her encouragement, for her understanding, for her great temper, for her standing by me all the time, and for her making me realize the most precious thing in life. It is your company and endless love that make all this happen.

To my wife and parents.

ABSTRACT OF THE DISSERTATION

Pre- and Post-Deployment Dynamic Bug Detection Techniques for MPI Programs

by

Hongbo Li

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Rajiv Gupta, Co-Chairperson
Dr. Zizhong Chen, Co-Chairperson

MPI is the de-facto standard message-passing based parallel programming model. However, the bug detection support for MPI applications is lacking. This thesis seeks to address the challenges of bug detection techniques for MPI applications. Specifically, it tackles two kinds of bugs: (1) general software bugs (e.g., segmentation faults, assertion violations, and infinite loops) that lead to abnormal execution termination or program hangs at *small scale*, i.e., when a program is executed with only a few processes and a small problem; and (2) scaling problems that manifest only at *large scale*, i.e., when a program is executed with a large number of processes or a large-sized problem.

To aid in the detection of general bugs, we developed COMPI as an automated bug detection tool. COMPI tackles two major challenges. First, it provides an automated testing framework for MPI programs — it performs concolic execution on a single process and records branch coverage across all. Second, COMPI effectively controls the cost of testing as too high a cost may prevent its adoption or even make the testing infeasible.

Furthermore, we enhanced the usability of COMPI via addressing two issues: input

values generated by COMPI do not deliver cost-effective testing, and COMPI does not support floating-point arithmetic and thus much code cannot be explored. We address the first issue via proposing a novel input tuning technique without requiring the intervention of users. We enable handling of floating point data types and operations and demonstrate that the efficiency of constraint solving can be improved if we rely on the use of reals instead of floating point values.

To tackle scaling problems, we provided a testing suite and designed an avoidance framework for scaling problems associated with the use of MPI collectives. To improve users' productivity, we establish the necessity of user side testing and provide a protection layer to avoid scaling problems non-intrusively, i.e., without requiring any changes to the MPI library or user programs. This provides an immediate remedy when an official fix is not readily available.

Finally, we built a hang detection tool that saves computing resources in the presence of program hangs at large scale. ParaStack is an extremely lightweight tool to detect hangs in a timely manner with high accuracy, in a scalable manner with negligible overhead, and without requiring the user to select a timeout value. For a detected hang, it tells users whether the hang is the result of an error in the computation phase or the communication phase. For a computation-error induced hang, our tool pinpoints the faulty process by excluding hundreds and thousands of other processes.

Contents

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**
1.1 Dissertation Overview 4
1.1.1 Concolic Testing for MPI Applications 5
1.1.2 Efficient Concolic Testing of MPI Applications 6
1.1.3 Tackling Scaling Problems with the Use of MPI Collectives 7
1.1.4 Hang Detection at Large Scale 8
1.2 Dissertation Organization 9

2 Concolic Testing for MPI Applications **10**
2.1 Concolic Testing 11
2.1.1 Challenges for MPI Application 13
2.1.2 Our Solution: COMPI 15
2.2 Overview of COMPI 16
2.2.1 Work Flow of COMPI 16
2.2.2 Search Strategy Selection 18
2.3 Framework Adaptation 21
2.3.1 Automatic Marking 21
2.3.2 Constraints Insertion 22
2.3.3 Conflicts Resolving 23
2.3.4 Test Setup and Program Launching 25
2.4 Practical Testing 27
2.4.1 Input Capping 27
2.4.2 Two-way Instrumentation 28
2.4.3 Constraint Set Reduction 29
2.5 Implementation 31
2.6 Evaluation 32
2.6.1 Uncovered Bugs 34
2.6.2 Input Capping 35

2.6.3 Two-way Instrumentation . . . 36
2.6.4 Constraint Set Reduction . . . 37
2.6.5 COMPI Framework and Random Testing . . . 40
2.7 Summary . . . 41

3 Efficient Concolic Testing of MPI Applications 42
3.1 Issues of COMPI and Our Solution . . . 43
3.1.1 Issues of COMPI . . . 43
3.1.2 Our Solution: Input Tuning and Floating-Point Support . . . 44
3.2 Background and Overview of Solutions . . . 46
3.2.1 Concolic Testing of MPI Programs . . . 46
3.2.2 Overview of Our Solutions . . . 48
3.3 Input Tuning . . . 51
3.3.1 Design of Our Approach . . . 51
3.3.2 Applicability . . . 56
3.4 Floating Point Support . . . 57
3.5 Evaluation . . . 60
3.5.1 HPL . . . 62
3.5.2 IMB-MPI1 . . . 65
3.5.3 SUSY-HMC . . . 67
3.6 Summary . . . 70

4 Tackling Scaling Problems In and Out of MPI Collectives 71
4.1 Scaling Problems . . . 73
4.1.1 Challenges . . . 75
4.1.2 Our Approach . . . 76
4.1.3 Overview . . . 78
4.2 Manifesting Scaling Problems . . . 79
4.2.1 Basics of MPI Collectives . . . 79
4.2.2 Testing . . . 80
4.2.3 Scaling Problems Uncovered . . . 82
4.3 Online Problem Detectors . . . 84
4.3.1 Class D: Displacement Array Corruption . . . 85
4.3.2 Class G: Global Data Buffer Too Large . . . 88
4.3.3 Class X: Trigger Form Not General . . . 90
4.3.4 Case Studies: Class G and X . . . 91
4.4 Non-intrusive Avoidance . . . 92
4.4.1 Workaround 1: Communication Partitioning . . . 93
4.4.2 Workaround 2: Big Data Type . . . 95
4.4.3 Applicability and Limitation . . . 96
4.4.4 Evaluation . . . 97
4.5 Summary . . . 103

5 Hang Detection at Large Scale **104**
5.1 Program Hang at Large Scale . . . 105
5.1.1 Challenge . . . 105
5.1.2 Our Solution: ParaStack . . . 106
5.1.3 The Case for ParaStack . . . 108
5.2 Lightweight Hang Detection . . . 110
5.2.1 Model Based Hang Detection Scheme . . . 116
5.2.2 Robust Model with a Limited Sample Size . . . 120
5.2.3 Lightweight Design Details . . . 123
5.3 Identifying Faulty Process . . . 126
5.4 Implementation . . . 128
5.5 Discussion . . . 129
5.6 Experimental Evaluation . . . 131
5.6.1 Hang Detection Evaluation . . . 132
5.6.2 Faulty Process Identification . . . 140
5.7 Summary . . . 142

6 Related Work **143**
6.1 General Bug Detection . . . 143
6.2 Tackling Scaling Bugs . . . 146
6.3 Techniques for Handling a Program Hang . . . 147

7 Conclusions and Future Work **149**
7.1 Contributions . . . 149
7.2 Future Work . . . 151

Bibliography **154**

List of Figures

1.1 Dissertation overview. 4

2.1 Concolic testing for a sequential C program. 12
2.2 An MPI program's code skeleton and its execution tree. 13
2.3 The iterative testing of COMPI: i-th test to $(i+1)$-th test (marked in green). 18
2.4 Branch coverage of HPL using four search strategies. 19
2.5 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. 24
2.6 The achieved branch coverage as well as the time cost at various matrix sizes for HPL. 27
2.7 Instrumentation comparison: one-way v.s. two-way. 28
2.8 Constraint set reduction given x is marked as symbolic and $x = 0$ before entering the loop. 30
2.9 Evaluation of input capping. 35
2.10 Constraint set size distribution for SUSY-HMC. 38
2.11 Constraint set size distribution for HPL: 38
2.12 Constraint set size distribution for IMB-MPI1. 39

3.1 Concolic testing of MPI programs: (1) on the left is a segment of one instrumented program with the code lines in bold being the original code, mark_symbolic() being inserted by developers, and the remaining being the symbolic execution code inserted automatically; and (2) on the right shows how the test engine tests the instrumented program. 47
3.2 Input tuning achieves cost-effective testing: (1) on the left is an MPI program performing square matrix multiplication with n denoting the matrix width; (2) on the right input tuning helps avoiding expensive execution via replacing 1234567 with 101. 49
3.3 Concolic testing of a program without support for floating-point data types and operations. 50

3.4 Two-stage tuning is applied on the solution generated by the solver — the solution contains the values generated for variables $x1$, $x2$, and $x3$, which are respectively $C1$, $C2$, $C3$. After Stage I tuning, the smallest upper bound, $B1$, is found for all involved variables (i.e., $x \leq B1$ for $\forall x \in \{x1, x2, x3, ...\}$ with $B1 \leq max\{C1, C2, C3, ...\}$). After Stage II, the smallest bound is found for variable $x1$ if $x1$ is the single variable in the target constraint (i.e., $x1 \leq B2$ with $B2 \leq B1$). Within the limits of these bounds, the constraints are solved to get the optimized solution. 51
3.5 Branch coverage progress over one-hour of testing of HPL using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y. 63
3.6 Branch coverage progress over one-hour of testing of IMB-MPI1 using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y. 66
3.7 Branch coverage progress over one-hour testing of SUSY-HMC using *input tuning* and *input capping*. 68
3.8 Branch coverage progress of testing of SUSY-HMC based on 3 versions of COMPI: (Int) only integers; (Real) with floating point extension using reals; and (Float) with floating point extension directly using floating point numbers. 69

4.1 Avoiding scaling problems via interception. 78
4.2 Safe bounds of G problems (Prob. 10, 11 and 13). 91
4.3 Safe bounds of an X problem (Prob. 12). 92
4.4 Illustration of the partitioning strategies for MPI_Gatherv ($P = 4$ and $n = 2$) by breaking down the filling process of the global data buffer. Process 0 is the root and the bug would be triggered when $nP > 4$. 93
4.5 Workaround 1-A for MPI_Gatherv. 94
4.6 Performance comparison between W1-A and the default MPI_Gatherv (MPICH) before the scaling problem's occurrence. 100
4.7 Performance comparison among the three workarounds for MPI_Gatherv (MPICH) whose scaling problem (Class D) is triggered once $sn > 2.625MB$ when $P = 768$. 100
4.8 Performance comparison among the three workarounds for MPI_Igather (Open-MPI) whose scaling problem (Class G) is triggered once $sn > 2.625$MB when $P = 768$. 100
4.9 Performance trend of W1-B for MPI_Gather (MPICH) whose scaling problem (Class X) is triggered once $sn > 7.75$MB when $P = 768$. 101
4.10 Performance comparison based on MPI_Gather (MPICH) supposing a Class G problem is triggered when $n > 128$K at $P = 768$. 101
4.11 Performance comparison based on MPI_Allgatherv (MPICH) supposing a Class G problem is triggered when $n > 128$K at $P = 768$. 102

5.1 ParaStack workflow – steps with solid border are performed by ParaStack and those shown with dashed border require a complimentary tool. 109

5.2 Dynamic variation of S_{out} observed from 3 benchmarks: LU, SP, FT from NPB suite. All are executed with 256 processes at problem size *D*. 111
5.3 The S_{out} variation of a faulty run of LU, where a fault is injected on the left border of the red region. 112
5.4 Hang detection. Three panels show the empirical distribution of randomly sampled S_{out} of LU, where the red region shows the suspicion region, the blue curve shows the probability density function $P(S_{out})$, and the dashed black curve shows the cumulative distribution function $F_n(S_{out})$. The red arrow crosses the suspicion region 3 times meaning 3 consecutive observations of suspicion. 118
5.5 Relation among sample size, suspicion probability and tolerance error, where $\hat{n}(\hat{p}) = \frac{3.8416}{e^2}\hat{p}(1-\hat{p})$. 121
5.6 Faulty process identification for *computation-error induced hangs*. On the left is an MPI program skeleton, which hangs due to a computation error in process 100. Traditionally, the faulty process can be detected based on the progress dependency graph as shown in the middle. Our technique greatly simplifies the idea by just checking runtime states as shown on the right. . 126
5.7 Performance comparison of running applications with ParaStack ($I = 100$ ms), with ParaStack ($I = 400$ ms) and without ParaStack (clean) on Stampede at scale 1024 based on 5 runs in each setting. The performance is evaluated as *GFLOPS* for HPCG and as *time cost in seconds* for all the others, and the the 5 runs are ordered by performance. 133
5.8 Performance comparison of running applications with ParaStack ($I = 100$ ms), with ParaStack ($I = 400$ ms) and without ParaStack (clean) on Tianhe-2 at scale 1024. 135
5.9 The response delay of hang detection based upon 100 erroneous runs for each application at scale of 256 on Tardis. *The horizontal axis represents response delay in seconds and the vertical axis represents the number of times ParaStack identifies a hang with the corresponding delay.* 138
5.10 The percentage of time savings ParaStack brings to application users in batch mode based on 10 erroneous runs of HPL with the average percentage equal to 35.5%. 139

List of Tables

2.1 MPI semantics related variables. 21
2.2 The mapping between local ranks and global ranks. 26
2.3 Complexity of Target Programs. 32
2.4 One-way vs. Two-way 36
2.5 Evaluation of constraint set reduction based on branch coverage. 37
2.6 Evaluation of COMPI's framework based on branch coverage. 40

3.1 Time cost (unit: seconds) of floating-point constraint solving using reals and floating-point values based 100 iterative tests of a simple synthetic program. 60
3.2 Comparison among **T**uning, **C**apping (C2, C4, C8, and C8 using **N**o **T**imeout), and **N**one based on HPL with two metrics: the time costs of covering 1860 branches and the number of tests completed in one hour. 64
3.3 Comparison among **T**uning, **C**apping (C2, C4, C8, and C8 using **N**o **T**imeout), and **N**one based on IMB-MPI1 with two metrics: the time costs of covering 730 branches and the number of tests completed in one hour. 67

4.1 Well-documented *scaling problems* reported online [20, 21, 133, 13, 6, 15]. Notes: (1) Effect - *H*ang, *C*rash and performance *D*egradation; (2) Failing scale (P, M) - the *P*arallelism scale and *M*essage size that trigger the problem. 72
4.2 Newly uncovered scaling problems. 73
4.3 Who can fix the scaling problems? 76
4.4 Notations. 79
4.5 MPI collectives and their global data buffer size. If (I) follows a collective, the collective has a non-blocking variation; if v follows a collective, the collective has an irregular variation. 80
4.6 Experiment Setup. 81
4.7 Safe bounds. 83
4.8 Scaling problem detectors. 84
4.9 Detector G's lookup table. 90
4.10 Workarounds applicability: "✓" - apply; "✗" - does not apply; "✘" - apply with restrictions. 96

4.11 Workarounds' effectiveness for MPI_Gatherv (D). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB. 97
4.12 Workarounds' Effectiveness for MPI_Igather (G). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB. 98
4.13 Effectiveness of Workaround 1-B for MPI_Gather (X). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB. 98

5.1 Adjusting the timeout method to various benchmarks, platforms and input sizes at scale 256 based on 10 erroneous runs per configuration. *Metrics*: AC – accuracy; FP – false positive rate; D – average response delay in seconds, i.e. the elapsed time from when the fault is injected to when a hang is detected. 114
5.2 Default input sizes used by each application at various running scales. Inputs D and E are the two largest inputs that come with the benchmarks. The input size for HPL specifies the width of a square matrix and the input size for HPCG specifies the local domain dimension. 130
5.3 For an execution of HPL on a 15000*15000 matrix, the clean run on average takes 185.05 seconds. O_t is the total stack trace overhead due to n stack trace operations. 132
5.4 Performance comparison of running applications with ParaStack ($I = 100ms$), with ParaStack ($I = 400ms$) and without ParaStack (clean) on Tardis at scale 256. *P*erformance is measured by the delivered *GFLOPS* for HPCG and by the *time cost in seconds* for the others, and *S*tandard deviation of the performance is shown. 134
5.5 ParaStack's Overhead on Tianhe-2 at scale 1024 based on the average of 5 runs. 135
5.6 Accuracy of hang detection. The rough time cost of a correct run is shown. 136
5.7 Response delay on Tianhe-2: D is the average response delay in seconds; S is the standard deviation. 137
5.8 Response delay on Stampede: D is the average response delay in seconds and S is the standard deviation. 137
5.9 ParaStack's generality for variation of platforms, benchmarks and input sizes at scale 256 based on 10 erroneous runs per configuration. Notes: (1) P stands for the default ParaStack with I being initialized as 400ms; $P*$ stands for ParaStack with I being initialized as 10ms. (2) AC, accuracy; FP, false positive rate; D, average response delay. 140
5.10 Evaluation of faulty process identification. 141

Chapter 1

Introduction

High-performance computing has been profoundly impacting our world. It provides vital support to various scientific discoveries and technological innovations such as physics simulation, weather forecasting, climate research, and oil and gas exploration. To meet this critical demand, ever-increasingly powerful supercomputers have kept being created – in 1993 the fastest supercomputer only attained 59.7 gigaflops (5.97×10^{10} FLOPS), now the No. 1 supercomputer, Sunway TaihuLight, reaches 93 petaflops (9.3×10^{16} FLOPS), and exascale computing (10^{18} FLOPS) is just around the corner [31]. Over the past two decades, distributed cluster system has evolved from none to the predominant architecture in the current HPC world — it accounts for over 85% of the current top500 supercomputers [31]. Along with the rise of cluster, MPI has evolved into the de facto standard for HPC applications on distributed clusters due to its great portability and performance [52]. Hence, enormous amount of MPI applications have been developed to serve various scientific discoveries and technological innovations.

It is known that software bugs undermine the correctness of applications and thus impair the efficient use of high performance applications. In the pre-deployment phase, i.e., software development phase, general software bugs such as segmentation fault, assertion violation, and infinite loop, can lead to abnormal execution termination or program hang at *small scale* (i.e., when a program is executed only with a few processes and a small problem). In the post-deployment phase, i.e., after software release, challenging scaling problems (bugs) — a class of bug that manifests only at *large scale* (i.e., when a program is executed with too many processes or a too large problem) — can escape the testing and linger inside software, and harm application users' use experience. However, the bug detection support for MPI applications lags far behind the ever-increasingly sophisticated hardware. The lack of support manifests in following respects.

- **Scarce systematic testing techniques and tools.** Though testing is the predominant technique in industry to manifest bugs prior to software release, there is little effort spent on developing systematic *software testing* techniques for HPC applications [70, 106], let alone MPI programs. The lack of testing techniques is likely the result of inadequate interaction between scientists, who play a leading role in HPC application development, and industrial software engineers [77, 70, 106, 76]. Without effective testing techniques, general software bugs can easily escape developers' sight and linger in released MPI applications.

- **No immediate remedies for scaling problems.** As the complexity of MPI collectives is directly impacted by both parallelism scale and problem size, their use often triggers scaling problems. Scaling problems arising from MPI collectives can be

very challenging to deal with due to the aggregated complexity of a large number of processes, a big input, the user code space, the MPI library, the environment setting, and even platform [134, 92, 13, 14, 6, 15, 16, 20, 22]. It thus is very common that application users are limited to inefficient small scale runs prior to an official patch release which sometimes is even not available as developers cannot reproduce the reported bug [15].

- **Deficient hang detection at large scale.** On supercomputers, users execute programs in batch mode and each job execution occupies the requested computing resources till its completion. Errors causing a program hang can arise in either the computation phase, e.g., a thread-level deadlock within a process, an unexpected infinite loop, and a soft error in one single process, or the MPI communication phase, e.g., a communication deadlock. Program hangs, once occurring, stall the program execution and thus waste all the requested resources before the allocated time expires. A suitable solution to reduce wastage is thus detecting hangs at runtime and terminating the job once a hang is detected. Ad hoc *timeout* mechanism [11, 85, 84, 98] is a traditional hang detection method; however, it is difficult to set an appropriate threshold even for users who have good knowledge of an application considering the threshold can vary across computing platforms, input sizes, and applications [90].

This thesis addresses all of the above challenges of bug detection for MPI applications. Our novel bug detection techniques facilitate the experience of software development for developers as well as the software use experience for software users. Prior to deployment, our tool automates the testing of MPI applications and provides bug reports using which

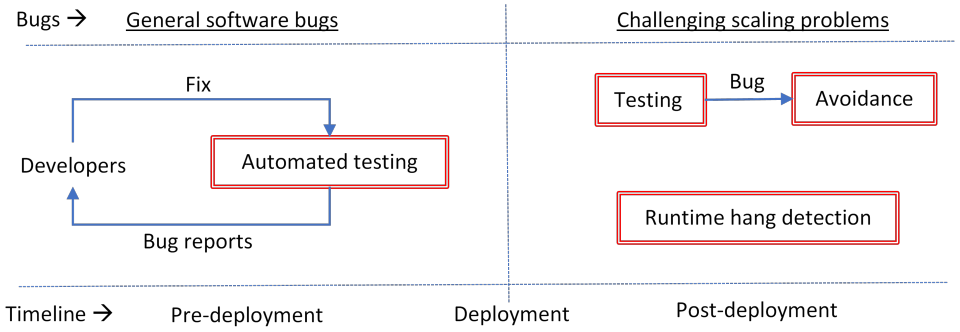

Figure 1.1: Dissertation overview.

developers can easily reproduce the bug and then fix it. After deployment, our testing suite can help users test suspicious MPI collectives and our avoidance framework helps avoid detected bugs, if any, without requiring any changes to MPI library and applications; also our hang detection tool helps save a great amount of computing resources in presence of a hang at large scale production runs.

1.1 Dissertation Overview

In this dissertation, we use dynamic techniques to aid the bug detection for both application developers and users. Figure 1.1 depicts the overview of this dissertation. Prior to software deployment, we provide an automated testing tool for developers to aid the detection of general software bugs. With the generated bug reports, developers can easily reproduce the bug and then fix it. After the deployment, we facilitate the use of applications in presence of scaling problems in two aspects. First, for users we provide a testing suite to test suspicious MPI collectives and an easy-to-use avoidance framework as an immediate remedy for a scaling problem. Also, for users we built a hang detection tool that saves computing resources in presence of program hangs at large scale production runs.

1.1.1 Concolic Testing for MPI Applications

We develop COMPI, the first concolic testing framework for MPI applications. COMPI tackles two major challenges. First, it provides an automated testing framework 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.

COMPI's framework make it achieve 4.8-81% more coverage than regular concolic testing. It uncovered four new bugs in one physics simulation program. It achieved 69-86% branch coverage which far exceeds the 1.8-38% coverage achieved via random testing. Its testing cost controlling techniques' effectiveness for practical testing is justified: (1) input capping lays the foundation of applying COMPI to practical MPI applications, without which the testing cost could be unreasonably high; (2) two-way instrumentation enables up to 66% testing time saving; and (3) constraint set reduction enables 4.7-10.6% more branch coverage.

1.1.2 Efficient Concolic Testing of MPI Applications

COMPI has extended concolic testing to MPI programs. However, two issues hinder its usability. First, it requires the user to specify an upper limit on input size – if the chosen limit is too big, considerable time is wasted and if the chosen limit is too small, the branch coverage achieved is limited. Second, COMPI does not support floating point arithmetic that is common in HPC applications.

To address the above issues, we propose input tuning and support floating-point data types and operations. We propose *input tuning* that eliminates the need for users to set hard limits and generates inputs such that the testing achieves high coverage while avoiding waste of testing time by selecting suitable input sizes. Moreover, we enable handling of *floating point* data types and operations and demonstrate that the efficiency of constraint solving can be improved if we rely on the use of reals instead of floating point values. Our evaluation demonstrates that with *input tuning* the coverage we achieve in 10 minutes is typically higher than the coverage achieved in 1 hour when input tuning is not used. Without input tuning, 9.6-57.1% loss in coverage occurs for a real-world physics simulation program. For the physics simulation program, using our *floating-point extension that uses reals* covers 46 more branches than without using the extension. Also, we cover 122 more branches when solving floating-point constraints using *reals* rather than directly using floating-point numbers.

1.1.3 Tackling Scaling Problems with the Use of MPI Collectives

As the complexity of MPI collectives is directly impacted by both parallelism scale and problem size, their use often triggers scaling problems. Scaling problems' root cause can be outside of MPI libraries and these can be easily exposed via dynamic interaction between user code and MPI library as the scale goes up. Specifically, irregular collectives suffer the most as the *C int* displacement array can easily be corrupted with integer overflow. Scaling problems can also result from a bug inside the released MPI libraries due to the lack of a systematic testing of MPI libraries as well as the platform or environment dependency of some scaling problems. Hence it is important for library users to perform testing on their platform to expose potential scaling problems. Fixing a scaling problem is challenging, and thus it usually takes much time for users to wait for an official fix, which sometimes is even not possible due to the difficulty of bug reproduction, root-cause identification, and fix development. To improve users' productivity, we establish the necessity of user side testing and provide a protection layer to avoid scaling problems non-intrusively — once the protection layer detects a condition that triggers a scaling problem it avoids the problem by either (1) chopping the communication into smaller ones or (2) building big data types. Our work hence provides an immediate remedy when an official fix is not readily available.

We uncover two kinds of Type-3 scaling problems: (1) an inherent defect in MPI standard on irregular collectives that impacts 8 MPI routines; and (2) 4 hidden scaling problems inside the released MPI libraries including OpenMPI and MPICH. Our protection layer consisting of three potential avoidance strategies is validated to be effective to bypass the scaling problems.

1.1.4 Hang Detection at Large Scale

While program hangs on large parallel systems can be detected via the widely used timeout mechanism, it is difficult for the users to set the timeout — too small a timeout leads to high false alarm rates and too large a timeout wastes a vast amount of valuable computing resources. To address the above problems with *hang detection*, this thesis presents *ParaStack*, an extremely lightweight tool to detect hangs based on runtime history in a timely manner with high accuracy, negligible overhead with great scalability, and without requiring the user to select a timeout value. It detect hangs by detecting dynamic manifestation of following pattern of behavior — persistent existence of very few processes outside of MPI calls. This simple, yet novel, approach is based upon the following observation. Since processes iterate between computation and communication phases, a persistent dynamic variation of the *count* of processes outside of MPI calls indicates a healthy running state while a continuous small count of processes outside MPI calls strongly indicates the onset of a hang. Based on execution history, *ParaStack* builds a runtime model of *count* that is robust even with limited history information and uses it to evaluate the likelihood of continuously observing a small count. A hang is verified if the likelihood of persistent small count is significantly high. Upon detecting a hang, *ParaStack* checks if there is any process in computation phase. If there is at least one process, it claims the hang is incurred by a computation error and reports such processes as faulty processes that contain the root-cause of this hang; otherwise, it claims the hang is incurred by a communication error.

We have adapted *ParaStack* to work with the *Torque* and *Slurm* batch schedulers

and validated its functionality and performance on *Tianhe-2* and *Stampede* that are respectively the world's current 2^{nd} and 12^{th} fastest supercomputers. Experimental results demonstrate that *ParaStack* detects hangs in a timely manner at negligible overhead with over 99% accuracy. No false alarm is observed in correct runs taking 66 hours at scale of 256 processes and 39.7 hours at scale of 1024 processes. *ParaStack* accurately reports the faulty process for computation-error induced hangs.

1.2 Dissertation Organization

The rest of thesis is organized as follows. Chapter 2 presents COMPI, an concolic testing tool for MPI programs in search of general software bugs. Chapter 3 details we improve COMPI with our proposed methods: input tuning and floating-point extension. Chapter 4 details the testing techniques to uncover scaling problems with the use of MPI collectives as well as our avoidance framework. Chapter 5 introduces our hang detection technique at large scale considering the detection efficiency and accuracy. Chapter 6 surveys existing literature in related areas and Chapter 7 summarizes our work and presents future outlook.

Chapter 2

Concolic Testing for MPI Applications

In industry, software testing is the predominant technique to ensure software quality, which is commonly known as an effective technique to uncover software bugs. However, little effort has been spent on developing systematic *software testing* techniques for HPC applications [70, 106], let alone MPI programs. It is thus not unexpected that the quality of HPC code is often lacking [78]. 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 [77, 70, 106, 76].

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* [38, 49, 57, 47] is commonly employed for automated test generation. But it is impossible to test all interesting behaviors of a program. *Symbolic*

techniques [37, 82] 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.

2.1 Concolic Testing

Concolic testing [116, 65] 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 [42] 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 sym-

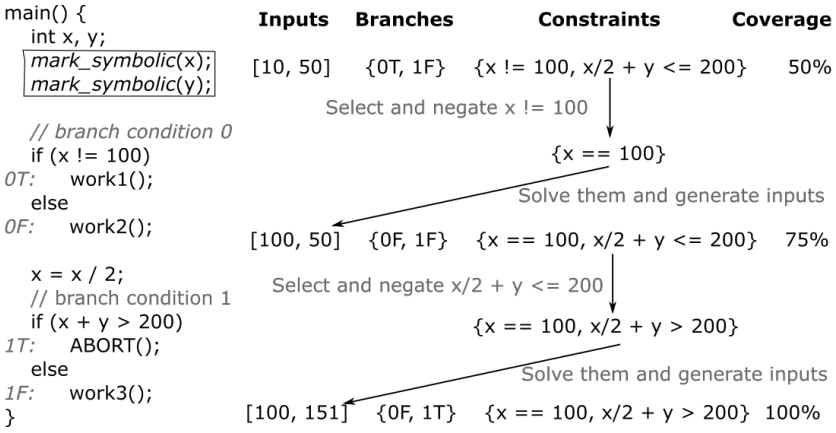

Figure 2.1: Concolic testing for a sequential C program.

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

Figure 2.1 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 four *branches*: $0T$, $0F$, $1T$ and $1F$ 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 covers branches $0T$ and $1F$ satisfying constraints $x \neq 100$ and $x/2 + y \leq 200$. To cover a new branch, the testing tool negates $x \neq 100$ and thus gets $\{x = 100\}$. It then generates the next inputs $\{x \leftarrow 100,\ y \leftarrow 50\}$ by solving the updated constraints. The inputs force the execution of $0F$. As the testing continues, it can derive new inputs and force the execution

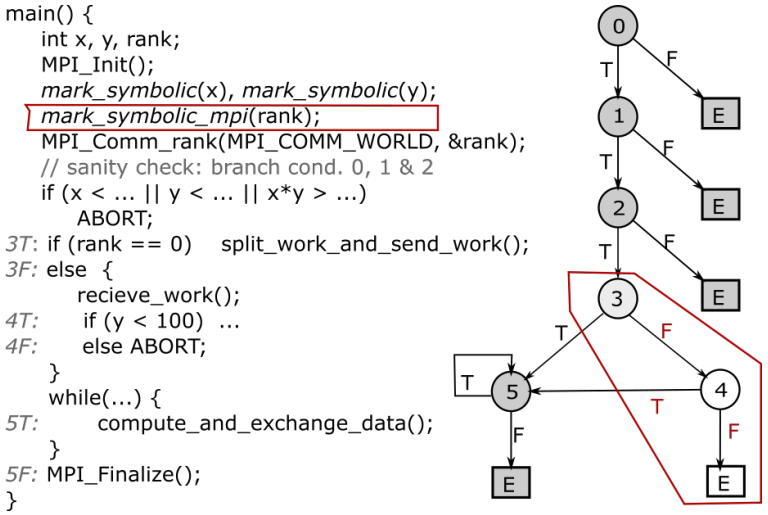

Figure 2.2: An MPI program's code skeleton and its execution tree.

of $1T$. Finally, 100% branch coverage is achieved. It should be noted during testing a bug is triggered when $1T$ is executed. The testing logs the error-inducing inputs for developers to perform further bug analysis.

2.1.1 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 2.2 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 cover branches related to such MPI semantics. Suppose concolic testing is only performed on process 0 for the program in Figure 2.2. During execution, branches $3F$ and $4T$ are encountered only by processes different from process 0, $4F$ is not encountered, and the remaining are covered by process 0. The testing fails to cover $3F$ and $4T$ as it does not record branches covered by processes other than process 0; it does not cover $4F$ 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 cover 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.

2.1.2 Our Solution: COMPI

To address the above issues, this chapter presents COMPI — a practical concolic testing tool to automate the testing of MPI applications. It is implemented on top of *CREST* [42], a scalable open-source concolic testing tool for C programs that replaces CUTE (one of the first implementations of concolic testing) [116]. 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, covering branches that cannot be covered 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 2.2, strategy (1) helps cover $3F$ and $4T$, and strategy (2) helps cover $4F$. 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 con-

firmed 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.

2.2 Overview of COMPI

2.2.1 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 2.2 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.

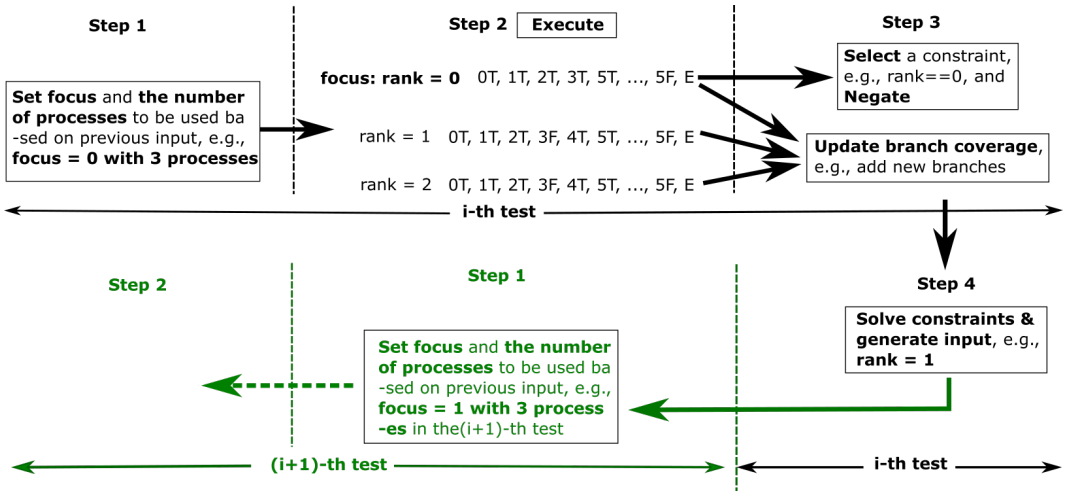

Figure 2.3: The iterative testing of COMPI: i-th test to $(i+1)$-th test (marked in green).

- It enables practical testing via effectively controlling the testing cost based on three techniques: input capping, two-way instrumentation, and constraint set reduction.

Figure 2.3 illustrates the iterative testing of COMPI from the i-th test to the $(i+1)$-th test on the program given in Figure 2.2. 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 cover the branch $4F$ in a future test.

2.2.2 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

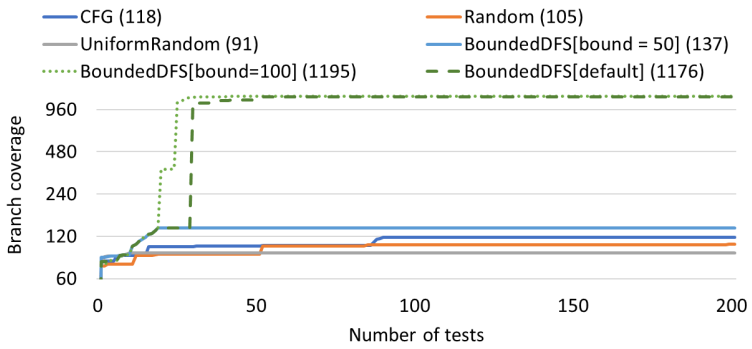

Figure 2.4: Branch coverage of HPL using four search strategies.

(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* [116] 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 2.2). The sanity check can consist of many conditional statements, and only by passing all the checks 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 cover 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 2.2. 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) [9] 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 2.4 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.

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

Table 2.1: MPI semantics related variables.

2.3 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 covered 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.

2.3.1 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 2.1 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.

2.3.2 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 ($r_w < s_w$). 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:

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

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.

2.3.3 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 2.5 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

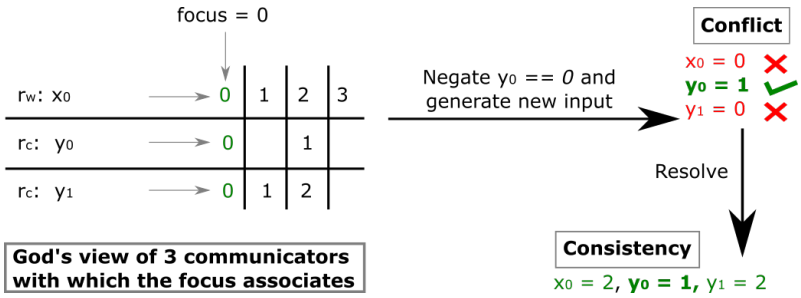

Figure 2.5: Resolving the conflicts among r_w and r_c variables by using the most up-to-date values. Each row in the left table maps to one communicator, and each column maps to one process.

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 2.5, 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.

2.3.4 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 2.2 illustrates the mapping array from the perspective of the focus

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

Table 2.2: The mapping between local ranks and global ranks.

(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

$$\text{mpiexec} \quad \text{-n} \ \ 1 \ \ ./\text{ex1} \ : \text{-n} \ \ s{-}1 \ \ ./\text{ex2}$$

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

$$\text{mpiexec} \quad \text{-n} \ \ i{-}1 \ \ ./\text{ex2} \ : \text{-n} \ \ 1 \ \ ./\text{ex1} \ :$$
$$\text{-n} \ \ s{-}i \ \ ./\text{ex2}$$

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.

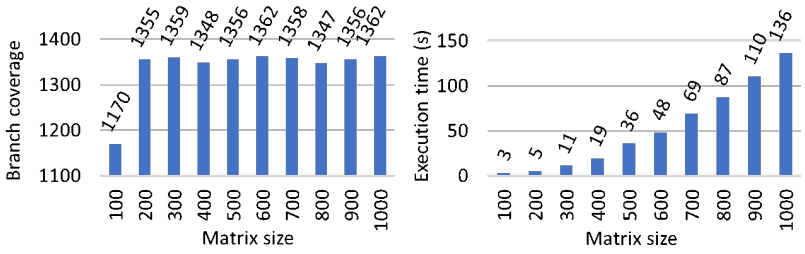

Figure 2.6: The achieved branch coverage as well as the time cost at various matrix sizes for HPL.

2.4 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.

2.4.1 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 2.6). 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

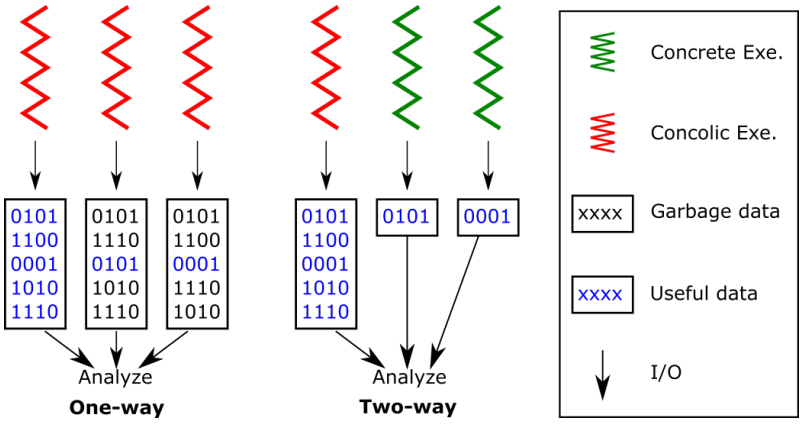

Figure 2.7: Instrumentation comparison: one-way v.s. two-way.

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

$$\text{COMPI\_int\_with\_limit(int } x\text{, int } cap\text{)},$$

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 2.3.2.

2.4.2 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 covered. This differentiating style minimizes the workload for non-focus processes and makes testing efficient. Figure 2.7 illustrates how two-way instrumentation saves redundant I/O for non-focus processes compared with one-way instrumentation.

2.4.3 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 2.8 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

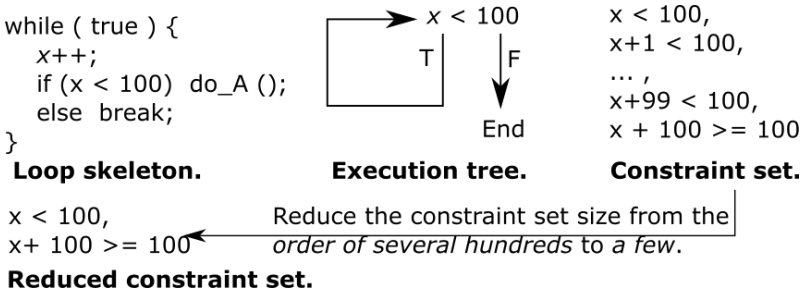

Figure 2.8: Constraint set reduction given x is marked as symbolic and $x = 0$ before entering the loop.

the last one, i.e.,

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

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.

2.5 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 [33]. 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) [99] under the guidance of an instrumentation module written in *OCaml* [18]. 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

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

Table 2.3: Complexity of Target Programs.

either the program returns a non-zero value or fails to complete within the specified timeout.

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.

2.6 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 2.3 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 [110]; (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 2.3 also shows the code complexity in different metrics: the source lines

of code (SLOC) measured by SLOCCount [27]; 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 [42]. We use the reachable branches to evaluate our coverage as some of branches found by static analysis are not reachable due to build configurations [42].

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 2.2.2, 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 covering additional bugs.

2.6.1 Uncovered Bugs

The use of COMPI on the programs detected *four bugs* in SUSY-HMC, where three cause segmentation faults [30] and one causes a floating point exception [7].

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:

$$\text{Twist\_Fermion } **src = \text{malloc}(Nroot * \text{sizeof}(** src));$$

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

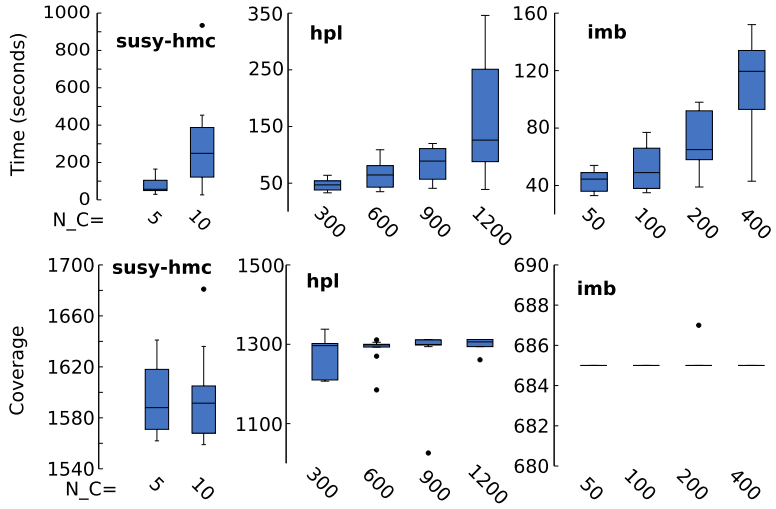

Figure 2.9: Evaluation of input capping.

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.

2.6.2 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 2.9 shows the testing time and the coverage comparison using different caps. For SUSY-HMC,

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

Table 2.4: One-way vs. Two-way

the average time 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.

2.6.3 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 2.4 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.

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 2.5: Evaluation of constraint set reduction based on branch coverage.

Also Table 2.4 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 need to perform tasks other than executing the program and recording the branch coverage information.

2.6.4 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.

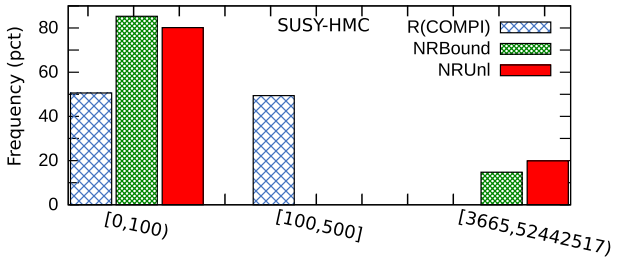

Figure 2.10: Constraint set size distribution for SUSY-HMC.

SUSY-HMC As shown in Table 2.5, 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 2.10 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.

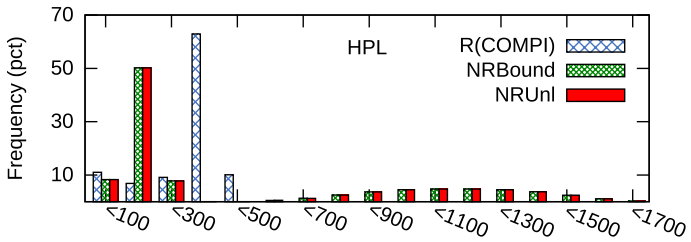

Figure 2.11: Constraint set size distribution for HPL:

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

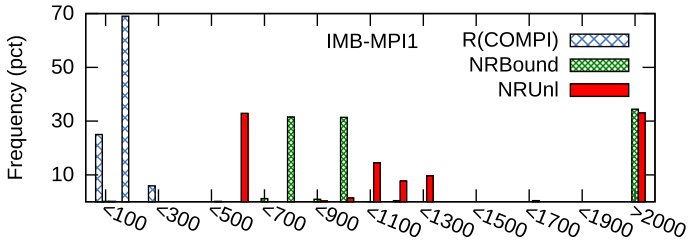

Figure 2.12: Constraint set size distribution for IMB-MPI1.

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 2.11 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 cover 767 branches are respectively 189s, 268s and 256s. Most importantly, Figure 2.12 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.

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%

Table 2.6: Evaluation of COMPI's framework based on branch coverage.

2.6.5 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 2.6.4. 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 2.6, 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 2.6.4. The reported results are based on three repetitions of each experiment. As shown in Table 2.6, 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.

2.7 Summary

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.

Chapter 3

Efficient Concolic Testing of MPI Applications

COMPI [91] applied *concolic testing* [116, 65] to boost the *branch coverage* of MPI applications. It proposed a concolic testing framework for MPI applications with adaptations enabling practical testing via controlling the cost of testing MPI programs. It performs symbolic execution only on one *focus* process in each execution and records branch coverage across all processes. Based on the same input, it can dynamically vary the *number of processes* (i.e., the size of MPI_COMM_WORLD), as well as the focus such that it can cover branches whose conditional statement depends on the size of MPI_COMM_WORLD or MPI rank such as the statement if ($rank == 0$).

3.1 Issues of COMPI and Our Solution

3.1.1 Issues of COMPI

First, the input values generated by COMPI do not guarantee cost-effective testing. It is known that the larger the problem size presented to an MPI program, the more time-consuming is the execution. If an excessively large value is generated for a variable that is closely related to the size of the problem, the testing cost can be exorbitant. To address this problem, COMPI proposes a technique, known as *Input Capping*, allowing developers to set an upper limit, referred to as the *cap*, for the input generation of each variable. Its underlying idea is that with a *well-selected* smaller cap values, inputs generated achieve branch coverages that are comparable to larger cap values at a far less testing cost. However, selecting such good caps is challenging. Excessively large caps ensure good coverage but incur exorbitantly high testing cost. Conversely, too small caps ensure the overhead per program execution is low but this comes at the cost of lower coverage because some constraints may have no solution under the *cap* limits and thus some branches cannot be explored. For simple programs manual inspection of the constraints of all branches can help developers find caps such that the caps do not prevent the constraints from being solved. *However, manual inspection is infeasible for complex or large programs and thus an automated approach is essential.*

Second, COMPI does not support floating-point types and operations that are commonly used in HPC applications. Using COMPI to test an MPI program that reads many floating-point values requires developers to manually fix the floating-point variables to selected values. But fixing the variables to certain values prevents testing from cover-

ing branches depending on these variables (e.g., the *true* side of conditional statement if ($x < 1$) cannot be exercised if we fix x to 2.0). Furthermore, floating-point operations are either ignored or recorded imprecisely (e.g., assignment statement $x = y + 1.5$ is ignored as expression $y + 1.5$ is a floating point operations). *The lack of floating-point support can cause some constraints not to be recorded or solved, and branches related to the use of floating-point types and operations may never be covered during testing.*

3.1.2 Our Solution: Input Tuning and Floating-Point Support

We propose *input tunning* to make testing cost-effective while avoiding the need for user to manually set hard *cap* limits. Its overall idea is as follows. COMPI generates new input values via solving a subset of dependent constraints (details in Section 3.2.1) — the new values are consumed by the variables appearing in these constraints in the next test run. Input tuning aims to make these values as small as possible as follows. It identifies the largest value L in the generated values and then, via binary search over the range $(0, L]$, it finds the smallest values for the involved variables such that the constraints can still be satisfied and thus uses them to drive the next test run. That is to say, we can achieve cost-effective testing via searching for small values to drive the testing as (1) the search does not disrupt the constraint solving unlike hard *cap* limits, and (2) they are small enough to ensure the least-expensive execution during testing.

We also extend COMPI to support *floating-point data types and operations* and show that the efficiency of constraint solving can be greatly improved if we rely on the use of reals instead of floating point values. Satisfiability modulo theories (SMT) solvers like Z3 [50] have begun to support floating point reasoning due to the recent advances of the

solver technology. This leads to the incorporation of floating-point reasoning into concolic testing [95]. However, solving constraints over floating-point numbers is far slower than over reals. Though approximating floating-point arithmetic using real arithmetic sacrifices precision, we show that the high efficiency of the approximation outweighs the imprecision in terms of achieving higher testing coverage in practice.

Our main contributions include:

- We present *input tuning* to achieve the most cost-effective testing via automatically searching for the smallest values that satisfy the collected constraints and thus eliminate the need for manually setting hard *cap* limits.

- We support *floating-point* data types and operations and demonstrate significant improvement in constraint solving and testing efficiency by approximating floating-point arithmetic using real arithmetic.

- We evaluate input tuning for HPL, IMB-MPI1, and SUSY-HMC based on one-hour of testing. For HPL, with input tuning we cover 1865 branches in less than 10 minutes which is 6× faster than the time it takes to achieve the same coverage without using input tuning. For IMB-MPI1, with input tuning we achieve coverage of 766 branches in less than 8 minutes while without it only 735 branches are covered in one hour. For SUSY-HMC, with input tuning we achieve the highest coverage, while with input capping 9.6-57.1% coverage loss occurs in other settings.

- We evaluate our floating-point extension using SUSY-HMC physics simulation program with one-hour of testing. With our floating point extension using reals we cover

46 more branches than without it. Also we cover 122 more branches when solving floating point constraints using reals rather than directly using floating point numbers during solving.

3.2 Background and Overview of Solutions

Here we briely describe the concolic testing process for MPI programs and the incremental constraint solving approach used for testing. We also illustrate with examples the existing issues of the current concolic testing tool for MPI programs as well as overview our proposed solutions.

3.2.1 Concolic Testing of MPI Programs

Testing process. The concolic testing of a given MPI program consist of two major steps: *instrumentation* and *iterative testing*.

In the instrumentation step, developers manually mark variables that read input values and dominate the program execution, then the marked program is transformed into a simplified program in C Intermediate Language (CIL) [99], and finally the simplified program is instrumented with symbolic execution code as shown in Figure 3.1. With the simplification, branch statements like loops and *switch* are all translated into *goto* and *if* statements. Each *if* statement only contains a simple condition, e.g., $\{x > 0\}$ instead of $\{x > 0 \text{ and } x < 10\}$, and is always accompanied with an *else* statement. The true/false *branch* outcome causes the execution of the *if*-side/*else*-side of the conditional statement. The *branch coverage* metric represents the number of branch outcomes covered during

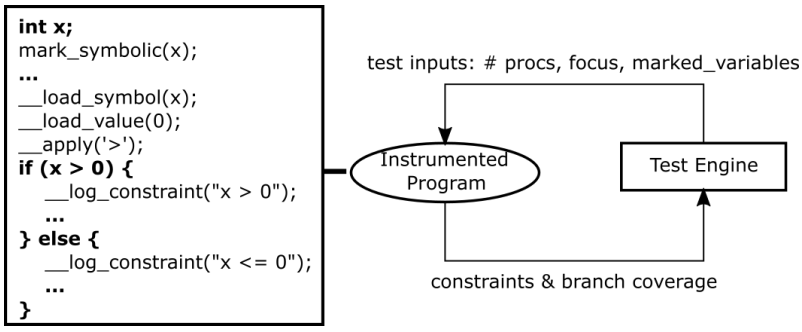

Figure 3.1: Concolic testing of MPI programs: (1) on the left is a segment of one instrumented program with the code lines in bold being the original code, mark_symbolic() being inserted by developers, and the remaining being the symbolic execution code inserted automatically; and (2) on the right shows how the test engine tests the instrumented program.

testing. Note the term *branch coverage* used in this chapter refers to the branch coverage of the simplified CIL program.

Next, iterative testing (i.e., iteratively executing the target program with generated inputs) is performed so as to increase the branch coverage and potentially uncover software bugs. At the end of each execution, a series of symbolic constraints mapped to the branches along the program execution path are recorded. The testing tool can generate a set of input values via solving constraints in a prefix of the execution path followed by a *negation* of the next constraint in the prefix. Due to the negation, the new inputs can potentially cover a new branch outcome. Among these inputs, some are used to determine the number of processes to be used as well as which process should be the *focus* process — the focus is the only process on which symbolic execution is performed while all the other processes only perform concrete execution (e.g., the code lines in bold in Figure 3.1). Based on these, the test engine can configure the right number of processes and the focus when launching the program. The remaining input values are passed to the marked variables at runtime, e.g., variable x takes one value via mark_symbolic() in Figure 3.1.

Incremental constraint solving is a widely used approach in many concolic testing tools due to its efficiency when solving similar constraints repeatedly. CREST [42], on which COMPI is built, benefits from it as well. Its basic idea is to exploit the similarity between two constraint sets being solving consecutively to speedup the solving process. It works as follows: (1) it only solves subset of constraints — the *target negated constraint* as well as *constraints depending on it* [1] — such that new values are generated for variables appearing in these constraints; and (2) it assign old values from previous input to all the other variables that do not appear in the constraints. Since each time only a subset of, instead of all, the constraints are solved, this technique greatly speedups the constraint solving.

We observe that a property inherent to this technique is: *An input value generated for a variable remains unchanged as long as the variable does not appear in the incrementally solved constraints.*

3.2.2 Overview of Our Solutions

Input tuning. Though COMPI's input capping relieves the issue to a certain degree, it is very challenging to select a good set of *cap* limits. Consider the MPI program performing square matrix multiplication shown in Figure 3.2 where variable n representing the matrix width determines the execution time. The program is designed to use different strategies for different range of matrix widths to optimize performance — small_matrix_multi() is invoked when $n < 100$ and large_matrix_multi() is invoked otherwise. If the upper *cap* limit is set to 50 (i.e., $n \leq 50$), large_matrix_multi() will not be explored during testing. On the other

[1] Two symbolic constraints are claimed to be dependent if only they share the same variables.

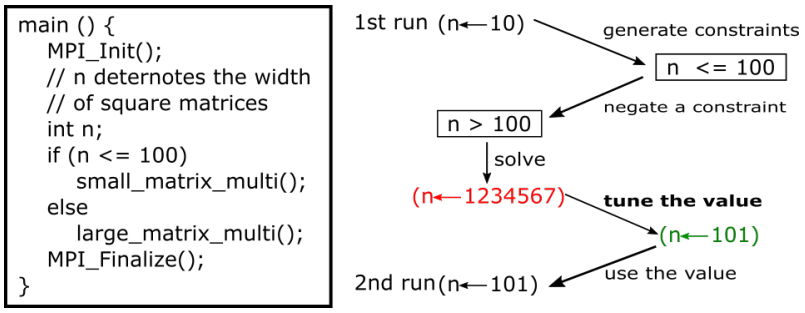

Figure 3.2: Input tuning achieves cost-effective testing: (1) on the left is an MPI program performing square matrix multiplication with n denoting the matrix width; (2) on the right input tuning helps avoiding expensive execution via replacing 1234567 with 101.

hand, if the upper limit is set to 500, the testing could be very expensive as the matrix width could be as high as 500. The property of incremental solving as discussed earlier in Section 3.2.1 exacerbates the high cost problem — once a large width value is generated it could stay unchanged for a long time and thus repeated time-consuming executions will be performed.

With our input tuning technique we can achieve the best cost-effective testing without the need for finding the best upper limits as input tuning always finds the smallest value to satisfy a given constraint. In Figure 3.2, the input tunning technique is illustrated. Suppose in the first run a random value is generated $n \leftarrow 10$. After execution, constraint $n \leq 100$ is obtained. Via negating it ($n > 100$) the testing aims to cover the branch outcome that invokes large_matrix_multi(). The solver can generate any value like $n \leftarrow 1234567$ to satisfy $n > 100$. This obviously is the worst scene the testing needs to avoid. With input tuning, we can find that $n \leftarrow 101$ also satisfy $n > 100$. This small value ensures large_matrix_multi() is invoked with the minimum possible execution time.

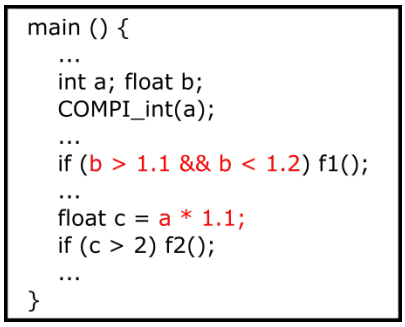

Figure 3.3: Concolic testing of a program without support for floating-point data types and operations.

Floating-point support. We exemplify the consequence of missing floating point support with the example shown in Figure 3.3. In this program, variable a and b read inputs from users. We mark a as symbolic. As there is no marking interface to support marking of b, a *float* variable, as symbolic in COMPI, we can only fix it to a selected value (e.g., 1.1). Variable c is also a *float* and its value is derived from a. Suppose a is initialized to 1 in the first test. However, function f1() cannot be explored as $b = 1.1$ does not satisfy $b > 1.1$ && $b < 1.2$, and f2() cannot be explored as the symbolic constraint $a * 1.1 \leq 2$ is not recorded, which is ultimately due to the fact that floating-point multiplication like $a * 1.1$ is ignored by COMPI's symbolic execution component.

To address this issue, we provide an interface to developers for marking floating-point variables and allow floating-point arithmetic in the symbolic execution component. Our extension helps cover branches related to the use of floating-point calculations.

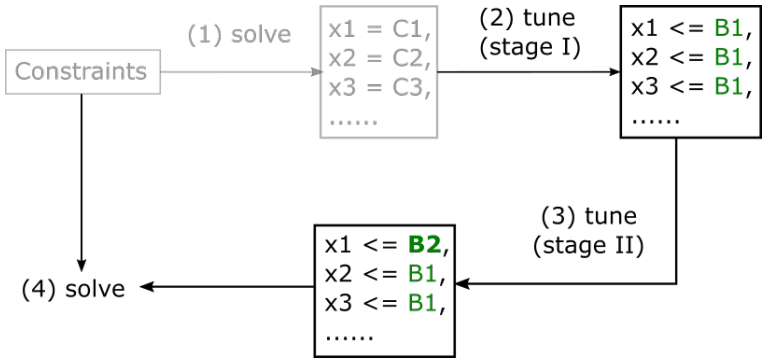

Figure 3.4: Two-stage tuning is applied on the solution generated by the solver — the solution contains the values generated for variables $x1$, $x2$, and $x3$, which are respectively $C1$, $C2$, $C3$. After Stage I tuning, the smallest upper bound, $B1$, is found for all involved variables (i.e., $x \leq B1$ for $\forall x \in \{x1, x2, x3, ...\}$ with $B1 \leq max\{C1, C2, C3, ...\}$). After Stage II, the smallest bound is found for variable $x1$ if $x1$ is the single variable in the target constraint (i.e., $x1 \leq B2$ with $B2 \leq B1$). Within the limits of these bounds, the constraints are solved to get the optimized solution.

3.3 Input Tuning

Directly applying the values generated by the constraint solver often incurs high testing cost that is not necessary. Though setting upper limits relieves this problem to a certain degree, it is challenging to manually find the best limits with which the testing achieves a high coverage yet incurs the least time cost. We thus propose *input tuning* as a solution to achieve effective testing that eliminates the challenge of setting hard *cap* limits.

3.3.1 Design of Our Approach

The tuning process consists of two stages: (Stage I) *Multi-variable tuning* that optimizes all variables appearing in the *dependent constraints* such that their values are no bigger than the detected smallest upper bound; and (Stage II) *Single-variable tuning* that optimizes the single variable in the target negated constraint, i.e., the target negated constraint only contains one variable, under the limit of the detected bound. Figure 3.4

illustrates how the two-stage tuning optimizes the solution, i.e., input values. These two stages are complimentary. Stage I ensures a dependent variable, like $x2$ and $x3$ in Figure 3.4, is not significantly increased when tuning the single variable in the target constraint, like $x1$. Stage II ensures the single variable gets the smallest value under the upper bound detected in Stage I. Below we details the two-stage tuning process shown as Algorithm 1, Algorithm 2, and Algorithm 3.

Stage I. Suppose *target* is the negated constraint, *cstrs* stands for the constraint set including *target* as well as the constraints depending on *target* (see Section 3.2.1), *excls* is a set of *symbolic symbols* [2] that do not need tuning, and *soln* stores the generated values for *symbols* appearing in *cstrs* as *key-value pairs* with *key* being a symbolic symbol and *value* being the generated value for symbol *key*. The goal of Stage I tuning is to find the lowest upper bound for all symbols not appearing in *excls*, i.e., we exclude symbols/variables that do not need to be tuned. This process is composed of the following steps:

- **Decide to tune or not** (Algorithm 1 and Algorithm 2). At first, we find the largest value, denoted as *bound*, among the generated values, stored in *soln*, for symbols not appearing in *excls* (lines 3 in Algorithm 1, i.e., Algorithm 2). If the largest value is too small (i.e., $bound < 2$), we directly return *soln* as the input values are already small enough and there is no need to tune them further (lines 4-5 in Algorithm 1).

- **Fix variables requiring no tuning** (Algorithm 1). We fix the variables that do not

[2]Each *symbolic symbol* represents a *variable* marked in the tested program. In the chapter, we use the term symbol and variable interchangeably

Algorithm 1 Two-stage Input Tuning

```
1: function TUNE(target, cstrs, excls, soln)
2:     /*** STEP 1: optimize a group of variables ***/
3:     bound ← get_largest(soln, excls)
4:     if bound ¡ 2 then return soln                                  ▷ 1.1
5:     end if
6:     _cstrs ← cstrs                                                 ▷ 1.2
7:     // fix the values of symbols in excls
8:     for all s.key ∈ excls do
9:         // construct new constraint: s.key = s.value
10:        c ← new_cstr(" = ", s.key, s.value)
11:        _cstrs ← _cstrs ∪ {c}
12:    end for
13:    opt_bound ← optimize_multi(_cstrs, excls,                      ▷ 1.3
14:          soln, bound);
15:    // set upper bounds                                            ▷ 1.4
16:    for all s ∈ soln AND s.key ∉ excls do
17:        c ← new_cstr(" <= ", s.key, opt_bound)
18:        _cstrs ← _cstrs ∪ {c}
19:    end for
20:    /*** STEP 2: optimize a single variable ***/
21:    if target contains more than one variable then                 ▷ 2.1
22:        return solve(_cstrs)
23:    end if
24:    if opt_bound < 2 then
25:        return solve(_cstrs)
26:    end if
27:    symb ← single symbolic symbol in target
28:    opt_bound2 ← optimize_single(_cstrs, excls,                    ▷ 2.2
29:          opt_soln, opt_bound, symb)
30:    if opt_bound2 < opt_bound then                                 ▷ 2.3
31:        c ← new_cstr(" <= ", symb, opt_bound2)
32:        _cstrs ← _cstrs ∪ {c}
33:    end if
34:    return solve(_cstrs)                                           ▷ 2.4
35: end function
```

Algorithm 2 Retrieve the largest value in a solution

```
1: function GET_LARGEST(soln, exls)
2:     max ← −1
3:     for all s ∈ soln do
4:         // largest not in excls
5:         if s.key ∉ excls and s.value > max then
6:             max ← s.value
7:         end if
8:     end for
9:     return max
10: end function
```

Algorithm 3 Search for the lowest upper bound

```
1: function OPTIMIZE_MULTI(cstrs, excls, soln, bound)
2:     /* * * optimize variables * * */
3:     lower ← 0, upper ← bound
4:     prev_upper ← upper
5:     while lower + 1 < upper do
6:         mid ← lower + (upper − lower)/2
7:         cstrs_ ← ∅
8:         for all s ∈ soln do
9:             if s.key ∉ excls then
10:                // construct constraint: s.key ≤ mid
11:                c ← new_cstr(" <= ", s.key, mid)
12:                cstrs_ ← cstrs_ ∪ {c}
13:            end if
14:        end for
15:        if cstrs_ is consistent with cstrs then
16:            upper ← mid
17:        else
18:            lower ← mid
19:        end if
20:    end while
21:    return upper
22: end function
```

need tuning, i.e., those appearing in $excls$, to the generated values in $soln$ to avoid any value changes caused by the tuning (lines 6-12).

- **Search the lowest upper bound** (Algorithm3). We search for the smallest upper bound for symbols/variables to be tuned using binary search in the range of $(0, bound]$ (lines 2-21). In the search, we construct new constraints via new_cstr() that specifies tuned variables are no greater than mid, where mid is the average of the lower and upper bound (lines 6-14). Then we check if the new constraints $cstrs\_$ are consistent with old ones $cstrs$ (line 15), and set the upper bound as mid if they are consistent (line 16) and the lower bound as mid otherwise (line 18). The lowest bound is obtained after the search is complete.

- **Set upper bound** (Algorithm 1). We set an upper bound for tuned variables via constructing new constraints specifying their values must be no larger than the detected bound (lines 15-19).

Stage II. Stage II aims to optimize the value for the single variable within the restriction of the upper bound detected in Stage I only if the variable is the single variable in the target negated constraint. It consists of similar steps.

- **Decide to tune or not** (Algorithm 1). We check if the target negated constraint, namely $target$, only contains single variable (lines 21-23) and if the detected bound is already small enough (lines 24-26). If either is not satisfied, we directly solve and return; otherwise, we proceed to the next step.

- **Search for the lowest upper bound** (Algorithm 1). This is the same to the search

in Stage I except that it optimize only a single variable (lines 27-29).

- **Update upper bound** (Algorithm 1). If the new bound is smaller than the older one, we update the upper bound for the single variable (lines 30-33).

- **Generate optimized values** (Algorithm 1). We solve the updated constraints, i.e., $\_cstrs$, to get the optimized values (line 34).

Additional setup. As no constraints are available prior to the first test, input generation for the first test is not available. We need to assign input values. We make all the initial values as the smallest positive integer for integer varaiables (i.e., 1). This setting makes not only the first test as well as latter tests efficient enough considering the value persistence property of incremental constraint solving.

3.3.2 Applicability

Input tuning is effective for tuning input values for a variable when the following conditions are satisfied: (1) the variable is of integer type like char, int, and long; (2) the larger the value of the variable is the longer the execution takes; and (3) eligible values allowing the program performing its function must be positive. For example, for the square matrix multiplication program the matrix width must be positive so as to perform valid matrix multiplication. Below we detail how we deal with the cases when one of the conditions is not satisfied.

Floating-point variables do not satisfy condition (1). We do not perform input tuning for floating-point variables as usually the values of integer variables, like matrix

width in the matrix multiplication program, determine the problem size. However, this technique can be applied for floating-point variables as well.

There two types of variables that do not satisfy condition (2): Type-1 variables whose values are unrelated to execution time and Type-2 variables for which increase in value leads to shorter execution. We do not differentiate type-1 when applying the tuning as tuning it does not have much side-effect other than the tuning cost. To deal with type-2, we allow developers to mark variables that need to be excluded from the tuning process. A variable representing the number of processes, i.e., the size of MPI_COMM_WORLD, is a good example of Type-2. As aforementioned, the testing also generates input values used for determining the number of processes: for the same workload the execution takes more time when less processes are used to run the program. In this chapter, we only mark variables representing the number of processes in *exclusion*, but application developers can feel free to add more when needed.

We do not tune the variables violating condition (3) as for the majority of, if not all, HPC applications only positive values are meaningful.

3.4 Floating Point Support

Enabling floating-point data types and operations in concolic testing requires adapting three components of COMPI: instrumentation module, symbolic execution library, and the constraints solving component.

Instrumentation module guides the insertion of symbolic execution code into the target program. The instrumentation is performed at instruction level. For example, the instruc-

tion $x = y + z$ needs to be inserted with four code sequences to achieve symbolic execution: two for loading the symbolic expressions of x and y, one for applying the add operation, and one for storing the symbolic expression for $y + z$ into x. The instrumentation module of COMPI only instruments integer variables and operations. We adapted the module such that it also instrument floating-point variables and operations.

Symbolic execution library defines all the instrumentation functions — the instrumented instructions discussed above are function calls to the functions of the library. These functions manipulate symbolic expressions according to the original instructions. In COMPI, all symbolic expressions only represent linear arithmetic operations as

$$E = C + \sum_{i=0}^{i=N-1} C_i * x_i,$$

where E is a symbolic expression, C is a constant, x_i denotes a symbolic symbol representing one input-taking variable, C_i is the coefficient of x_i, and N is the number of symbolic symbols in E. COMPI ensures linear constraints via replacing symbolic expressions with concrete values as needed. For example, consider the multiplication of two symbolic expressions: $x * y$ with both x and y being symbolic expressions. To avoid non-linear operation $x * y$ being recorded, COMPI substitutes the symbolic expression of y with the concrete value of y like 2 such that the result expression is $2x$ which is still linear. As COMPI only targets integers, it records C and C_i using 64-bit integers.

Our floating-point extension requires us to represent integer expressions in the same way, but for a floating-point expression we record C and C_i using double-precision

floating point values. Also, the extension also needs conversion between floating-point and integer expressions. We convert a integer expression into a floating -point expression via converting C and C_i from 64-bit integers to double precision floating point numbers. We convert a floating-point expression into an integer expression via converting the concrete value of the floating-point expression into an 64-bit integer, i.e., after the conversion the integer expression is a concrete value instead of symbolic expression. In addition, we provide the marking functions for developers to mark variables of data type *float* and *double* as symbolic such that these variables can also be involved in the symbolic execution.

Constraints solving component solves constraints to generate new inputs that are used in the next test run and this process is used repeatedly during iterative testing. For incremental solving, this component finds all constraints depending on the target negated constraint, and uses Yices-1.0 [33], an SMT solver, to solve the dependent constraints. In COMPI, the component is only able to solve integer constraints.

As SMT solvers like $Z3$ [50] has begun to support floating-point reasoning, concolic testing is also able to solve constraints with floating-point arithmetic based on the floating-point reasoning of Z3. However, the floating-point reasoning is known for its high cost — the cost of solving floating-point constraints is hundreds of times the cost of solving integer constraints [129]. Therefore, instead we propose simulating floating-point arithmetic using real arithmetic that is far less expensive. To compare the efficiency between solving using reals and using floating-point values, we created two versions of COMPI: one solves constraints using floating-point reasoning of Z3, and the other solves constraints using real arithmetic of Z3. We use the two versions of the tool to test a simple synthetic program

Expression →	x	$x+y$	$x+y+z$
Float →	31.4	75.0	91.2
Real →	8.2	8.1	8.2

Table 3.1: Time cost (unit: seconds) of floating-point constraint solving using reals and floating-point values based 100 iterative tests of a simple synthetic program.

with 3 if statements below:

```
if (expr == 0) ...

if (expr < 0) ...

if (expr <= 0) ...
```

where *expr* stands for an C floating-point expression. In the testing, the program can generates 6 constraints including $expr = 0$, $expr \neq 0$, $expr < 0$, $expr \geq 0$, $expr \leq 0$, and $expr > 0$ such that all the relational operators are covered.

Based on 100 iterative tests, we measured the time cost of constraints solving using reals and using floating-point values based on three expressions: x, $x + y$, $x + y + z$ (the data type of x, y and z are all *float*). Table 3.1 shows that the solving time using floating point values is 3.8× to 11.1× times the solving time using reals. Also the solving time using floating point value grows as the number of variables in the expression grows, while the solving time using reals stays almost the same. Hence, we believe the efficiency of solving floating point constraints using *reals* makes it a better fit for practical testing.

3.5 Evaluation

We evaluate *input tuning* and *floating point* extension of concolic testing based on three non-trivial MPI applications.

Hardware and Tool setup. The evaluation is performed on a computer equipped with two Intel E5607 CPUs with total of 8 cores and 32 GB memory. In the evaluation, COMPI tool uses Z3 instead of Yices-1.0 as its constraint solver due to the floating point extension. By default, the tool runs the target program with 8 processes with the focus being rank 0 in the first test. Additionally, the number of processes is restricted to no more than 16 during dynamic variation as without it the computer can crash when running with too many processes. Our tool sets all input values to 1 for the first test run for both input tuning and input capping techniques for fair comparison. The decision on which constraint to negate is made by the search strategy — COMPI uses is *BoundedDFS*. BoundedDFS explores the execution tree using a variation of depth-first search (DFS) strategy which skips constraints as well as branches that are deeper than a *specified depth bound* in the execution tree. The depth bound is selected to ensure that COMPI has the ability to explore the entire execution tree. The testing process using BoundedDFS (1) applies x tests without setting a bound first so that the maximal number of constraints M can be observed and (2) performs the testing with a selected bound B, which is obtained via rounding up M to the next hundred. In the default setting, we perform 100 tests to detect the bound, i.e., $x = 100$.

Evaluation goals and applications. Our evaluation aims to show that input tuning is more effective than input capping, i.e., it achieves *higher coverage at lower testing cost*. We use HPL [9], IMB-MPI1 [10], and SUSY-HMC [111] to evaluate input tuning as they all have integer inputs. For floating-point support, we aim to show that testing with floating-point extension achieves higher coverage than without it and solving floating-point constraints using real values saves testing time without sacrificing branch coverage. This evaluation

uses only SUSY-HMC as it has multiple floating-point inputs while HPL has only one and IMB-MPI1 has none.

3.5.1 HPL

HPL [9] is a high-performance Linpack benchmark for distributed memory computers. It solves a dense linear system using LU factorization. Many of the algorithm features can be exploited by configuring the abundant parameters it provides. To enable concolic testing, we need to mark variables for which the testing tool is to generate input values. HPL read inputs from a designated file, marking variables requires us to insert the marking lines as well as commenting out the reading from the file. For HPL we mark 23 integer variables (the variable can also be an array) by inserting 23 lines of code as well as commenting out the same amount of lines. The depth bound for *BoundedDFS* is 500 based on the observations in the first 100 tests.

We compare *input tuning* with four *input capping* settings as well as the case where neither input tuning or input capping is used (called *None*). In the input capping evaluations, we set the same cap (or upper bound limit), denoted as c, for all variables, and use three caps: $c = 2$, $c = 4$, and $c = 8$. We also evaluate $c = 8$ without the timeout mechanism — the tool by default uses timeout to identify excessively long executions such as those caused by infinite loops — to avoid the interference from timeout as many large input values cause the execution to timeout when $c = 8$. We allow each of the above configurations to *test for one hour*.

Figure 3.5 shows that using *input tuning*, the testing covers 1865 branches, which

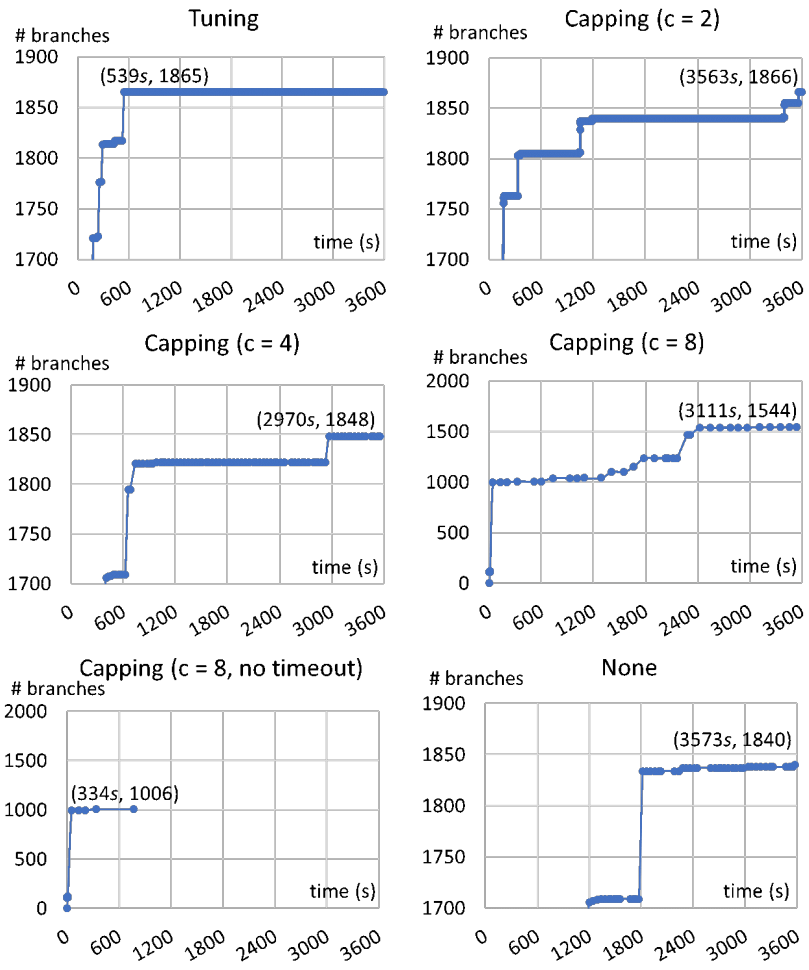

Figure 3.5: Branch coverage progress over one-hour of testing of HPL using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y.

is only 1 less the highest coverage. Using input capping with $c = 2$, the testing achieves the highest coverage but the time cost to achieve such coverage is almost 1 hour while the time cost of covering 1865 branches using input tuning is less than 10 minutes. This is because, for $c = 2$ the values of all variables must be smaller than 2, and thus very often the constraints have no solution. Using capping with $c = 4$, the testing coverage is 17 branches fewer than when using *input tuning*. Using capping with $c = 8$ in the default setting, 321

Metric ↓	**T**	**C**2	**C**4	**C**8	**C**8_**NT**	**N**
Cost (1860)	539	3563	–	–	–	–
# tests	390	1717	231	63	32	215

Table 3.2: Comparison among **T**uning, **C**apping (C2, C4, C8, and C8 using **N**o **T**imeout), and **N**one based on HPL with two metrics: the time costs of covering 1860 branches and the number of tests completed in one hour.

branches fail to be covered as larger upper bound permits larger values and larger values make the execution unnecessarily long such that many executions are killed by the timeout mechanism. Using capping with $c = 8$ without the timeout scheme, the coverage obtained is even less due to the same reason — too large values can cause one program execution to take tens of minutes (the program execution that started after 768 seconds did not finish till finally 1 hour expired). The *None* configuration that directly uses the values generated by the solver (i.e., neither tuning nor capping is used) delivers coverage of 1840 branches after running for over 30 minutes. This is not only worse coverage than *input tuning* but also at a much higher execution time cost (10 minutes vs. 30 minutes).

Table 3.2 demonstrates the high efficiency of testing using input tuning. The time it takes to cover 1860 branches using *input tuning* is 539 seconds which is only 15.1% cost of using capping with $c = 2$. In all other configurations the coverages and time costs are significantly worse. This high efficiency is the result of *input tuning* preferring smaller values and only using larger values when necessary. Thus, input tuning ensures testing makes progress at a good pace. Table 3.2 also shows the efficient testing using *input tuning* executed 390 tests in one hour. All other configurations, except input capping with $c = 2$, perform fewer tests in one hour because unnecessarily long executions are involved. Although input capping with $c = 2$ executes many more test cases with short runs, it still

takes about one hour to deliver nearly the same coverage because frequently constraints have no solution. In other words, *input tuning* choses neither too small nor too large inputs and as a result execution runs are just long enough to keep delivering solvable constraints and thus higher and higher coverage.

3.5.2 IMB-MPI1

IMB-MPI1 [10] is a major component of Intel MPI Benchmarks (IMB) and is used for benchmarking MPI-1 functions. It reads inputs by parsing the command line. We mark 14 integer variables by commenting out the whole code block that parses command line and inserting 30 lines with about half of them being the marking lines and the others being sanity checks on the inputs. The depth bound for *BoundedDFS* is 200 based on the observation in the first 100 tests. We compare input tuning with input capping as well as the case where neither tuning nor capping is used. In the *input capping* evaluation, we set the same cap for all variables, and use three configurations: $c = 2$, $c = 4$, $c = 8$. We also evaluate $c = 8$ without using timeout. Once again we perform *testing for one-hour testing* in each configuration.

Figure 3.6 shows using *input tuning*, we cover the most branches, i.e., 766 branches. Using capping with $c = 2$, we cover about 700 branches as the cap limit is too small. When we use bigger cap limits like $c = 4$ and $c = 8$ in the default setting, the coverage is over 30 branches less than the coverage based on input tuning. Using capping when $c = 8$ without timeout scheme, the coverage does not improve since without timeout expensive tests costing several minutes are used and thus one hour is not enough to explore the

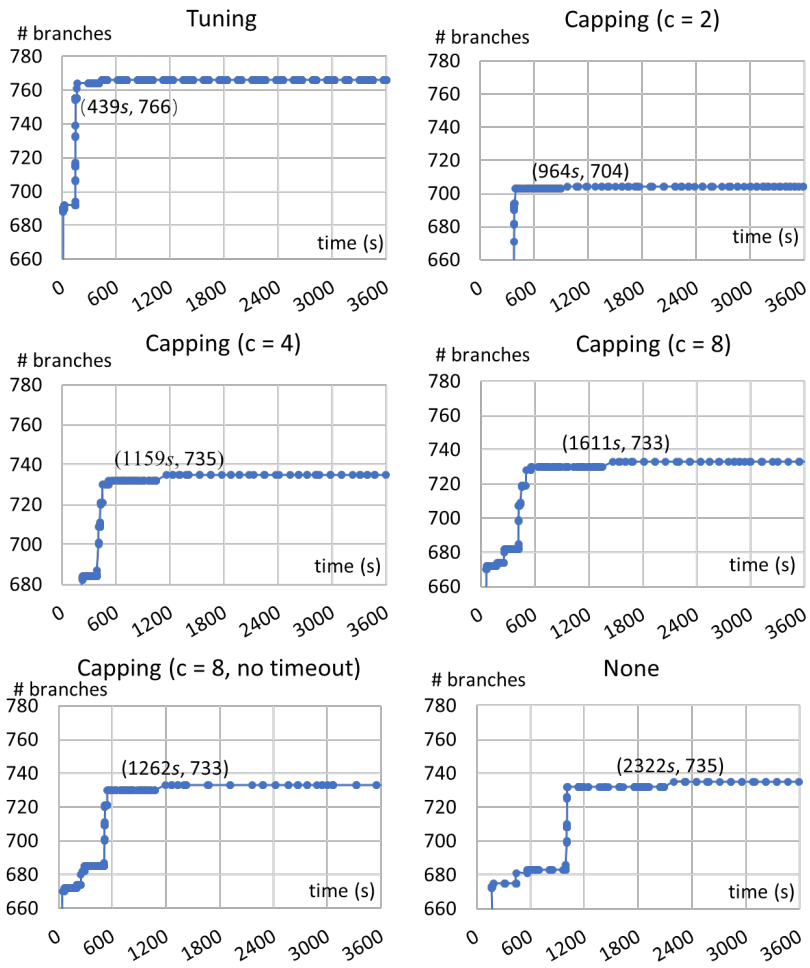

Figure 3.6: Branch coverage progress over one-hour of testing of IMB-MPI1 using *input tuning*, *input capping*, and *None* of them: a point (x, y) in each plot indicates that it takes x seconds to attain the maximum branch coverage of y.

branches. Without using either input tuning or capping technique (*None*), the coverage is less than the coverage using *input tuning* as frequently long executions are killed by the timeout mechanism of COMPI.

Most importantly, the efficiency of *input tuning* is justified — with input tuning we cover 766 branches in only 439 seconds, which cannot be achieved in any other configurations. Further Table 3.3 shows with input tuning we cover 730 branches in 142 seconds,

Metric ↓	**T**	**C**2	**C**4	**C**8	**C**8_**NT**	**N**
Cost (730)	142	–	449	559	545	1011
# tests	2806	280	246	244	240	224

Table 3.3: Comparison among **T**uning, **C**apping (C2, C4, C8, and C8 using **N**o **T**imeout), and **N**one based on IMB-MPI1 with two metrics: the time costs of covering 730 branches and the number of tests completed in one hour.

which is only 14.0-31.6% the time cost of other techniques. Still this is because input tuning permits large values as well as long program executions only when necessary. Table 3.3 shows the number of tests performed by testing using input tuning is far bigger than the number of tests performed by testing in input capping configurations. This result from the fact that if using input capping we need to carefully find the most appropriate cap limit for each variable to achieve cost-effective testing, while the default limits for all variables are set to the same value. It is obviously very challenging to make input capping cost-effective as selecting cap limits is hard. On the other hand, *input tuning* eliminates the need for setting limits.

3.5.3 SUSY-HMC

SUSY-HMC [111] is a major component in SUSY LATTICE — a physics simulation program for Rational Hybrid Monte Carlo simulations of extended-supersymmetric Yang–Mills theories in four dimensions. It reads inputs from standard input stream. We consider 13 integer variables and 7 double-precision floating point variables and mark different numbers of variables depending on the evaluation goals. The marking is achieved by commenting out the code block that reads values from standard input stream and inserting around 23 lines of marking code. The depth bound for *BoundedDFS* is 500 based on the observation in the first 100 tests.

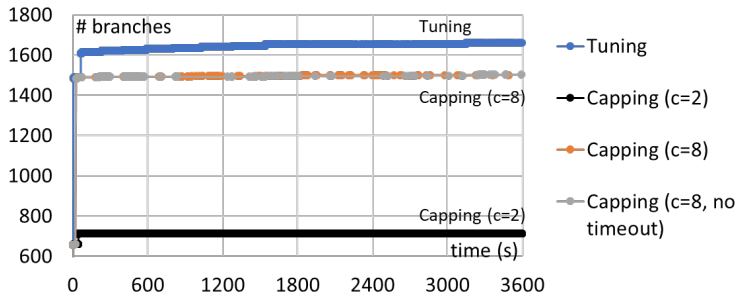

Figure 3.7: Branch coverage progress over one-hour testing of SUSY-HMC using *input tuning* and *input capping*.

Input tuning. We mark 13 integer variables using COMPI that does not have floating-point support. We compare input tuning with three input capping settings ($c = 2$, $c = 8$, and $c = 8$ without the timeout scheme). Our tool aborts in two configurations: capping with $c = 4$ and the case where neither input tuning nor input capping are used. Thus, results in these configurations are not shown. Each configuration is evaluated over one-hour of testing.

Figure 3.7 demonstrates with *input tuning* we obtain the highest coverage, i.e., 1662 branches. Using input capping with $c = 2$, we only cover 713 branches. Using capping with $c = 8$, we covers at most 1503 branches regardless of whether the timeout scheme is used or not. Using input capping, the loss in coverage ranges from 9.6% to 57.1%. Obviously, the serious coverage loss using input capping is caused by a bad *cap* limit. On the other hand, *input tuning* delivers high coverage without requiring users to find a good *cap* limit.

Floating-point support. On the basis of input tuning, we evaluate our floating point extension by comparing three versions of the testing tool: one that only considers integers

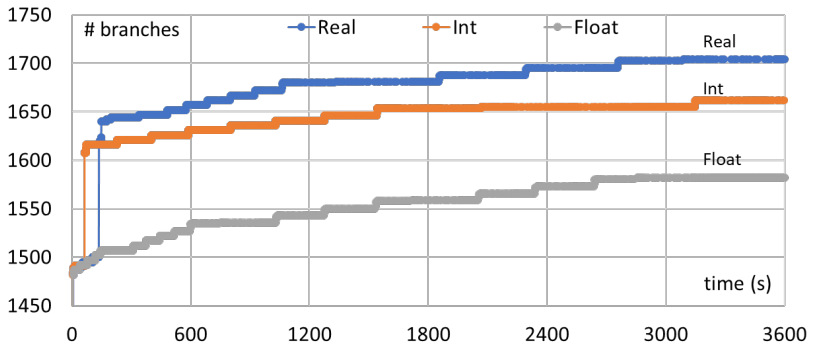

Figure 3.8: Branch coverage progress of testing of SUSY-HMC based on 3 versions of COMPI: (Int) only integers; (Real) with floating point extension using reals; and (Float) with floating point extension directly using floating point numbers.

(Int); one with floating point extension using reals (Real); and one with floating point extension that directly uses floating point numbers (Float). When using Real and Float, we mark all the identified 13 integer variables and 7 floating-point variables. When using Int, we (1) only mark the 13 integer variables as floating-point variables cannot be marked in the Int version, and (2) fix the floating-point variables to 1 for fair comparison, considering all COMPI versions set all input values to 1 in the first test run. We perform one-hour of testing using each version of the tool.

Figure 3.8 shows that Real achieves the best coverage after 200 seconds of testing. Real covers 1704 branches, Int covers 1662 branches, and Float only covers 1582 branches. We find that Real covers 42 more branches than Int. This demonstrates that floating-point extension help testing achieve greater coverage. Also, we find Float achieves the worst coverage though it also support floating point arithmetic. This results from the fact that constraint solving directly using floating-point arithmetic is inefficient — the constraint solving cost of Float accounts for 10.9% in the total testing time while the cost of constraint solving with Real only accounts for 1.7%. Thus, constraint solving using reals is efficient

and thus more practical for testing in comparison to solving constraints by directly using floating-point numbers.

3.6 Summary

Two existing issues hinder the use of COMPI. First, the input values generated by COMPI do not guarantee cost-effective testing. Second, floating-point data types and operations are not supported and thus coverage loss can occur. We propose *input tuning* to achieve cost-effective testing by favoring small input values. Also, we provide *floating-point* support and argue that the efficiency of constraint solving as well as testing could be significantly boosted if solving using reals instead of floating-point numbers. Evaluation results demonstrate that *input tuning* achieves high branch coverages much quicker than when it is not used. We further demonstrate that floating-point extension using reals helps us achieve higher coverage and solving constraints using reals is a better fit for practical testing compared with direct use of floating-point numbers.

Chapter 4

Tackling Scaling Problems In and Out of MPI Collectives

MPI application developers face the challenge of dealing with bugs whose root-cause is often hard to locate because errors in one process can easily propagate to other processes via communication. To make matters worse, some errors, such as integer overflow and resource exhaustion, manifest only at *large scale*. We refer to them as *scaling problems* [133, 134, 121, 86].

It has been recognized that program *scale* has two dimensions: *parallelism scale*, i.e. the number of parallel processes; and *problem size* [134] that impacts the *message size* that must be handled by the MPI library. Thus, scaling problems can be triggered by large values in either one dimension or both leading to the following natural classification: *Type-1* problems are only triggered by a large parallelism scale; *Type-2* problems are only triggered by a large problem size; and *Type-3* problems are triggered by the combination of

Prob.	Collective	MPI library	Type	Effect	Scale (P, M)	Root cause (inside MPI)
1	MPI_Gather	OpenMPI 1.4.3	3	H	(64, 4KB)	Environment setting dependency
2	MPI_Alltoall	OpenMPI 1.4.3	3	H	(44, 4MB)	Environment setting dependency
3	MPI_Allgather	OpenMPI 1.4.3	3	H	(64, 4MB)	—
4	MPI_Alltoallv	OpenMPI 1.7	3	H	(96, 512KB)	Network connection failure
5	MPI_Allgather	MPICH 2	3	D	$P \cdot M >$ INT_MAX	Integer overflow in MPI
6	MPI_Send + Recv	Intel MPI 5.1.2	2	H	(2, 64KB)	OS (ubuntu) dependency
7	MPI_Bcast	Intel MPI 5.1.2	2 or 3	H	(2, 64KB)	Unknown to developers
8	MPI_Bcast	Intel MPI 2017	2 or 3	H	(—, 16KB)	Platform (KNL & BDW) dependency

Table 4.1: Well-documented *scaling problems* reported online [20, 21, 133, 13, 6, 15]. Notes: (1) Effect - *H*ang, *C*rash and performance *D*egradation; (2) Failing scale (P, M) - the *P*arallelism scale and *M*essage size that trigger the problem.

Prob.	Collective	MPI library	Type	Effect	Scale (P, M)	Root cause
9	MPI_Gatherv(I)	MPI Standard	3	C	(48, 44MB)	Outside MPI
	MPI_Scatterv(I)		3	C/H		
	MPI_Allgatherv(I)		3	C		
	MPI_Alltoallv(I)		3	C		
10	MPI_Igather	OpenMPI 1.7 & 1.10	3	C	(48, 44MB)	Inside MPI
11	MPI_Iscatter		3	C/H		
12	MPI_Gather	MPICH 3.1.3	3	C	(48, 128MB)	
13	MPI_Scatter		3	C	(48, 44MB)	

Table 4.2: Newly uncovered scaling problems.

the two. We collected a list of well-documented scaling problems reported online as shown in Table 4.1. Also, we detected new bugs in various MPI versions that are listed in Table 4.2. A scaling problem is classified as Type-3 if the description stresses both parallelism scale and message size, as Type-2 if it is only related to message size, and as unknown (either Type-2 or Type-3) if it depends on message size while its dependence on parallelism scale is unknown. With such classification, ten scaling problems are Type-3 (Prob. 1-5, 9-13), one is Type-2 (Prob. 6), and two are unknown (Prob. 7-8). To our knowledge, Type-3 is the most common type of scaling problem, Type-2 is next, and Type-1 is the least common as in our investigation we are yet to observe a Type-1 problem. *This chapter focuses on Type-2 and Type-3 scaling problems – Type-3 problems are discussed in the context of MPI collectives as collective communication directly depends on* both *the parallelism scale and the problem size.*

4.1 Scaling Problems

Scaling problems can be exposed in the dynamic interaction between user code and MPI library. In the interaction, the target program runs with various number of processes

and demands the passing of messages of differing lengths. In extreme cases, the use of too many processes (too large messages) causes the corruption of MPI routines though it only demands communications of messages of moderate lengths (a moderate number of processes). Among all the MPI routines, *irregular collectives*, that enable processes to transfer varying amounts of data, suffer from this problem the most due to their use of *C int displacement array* that characterizes irregular collectives. Take MPI_Gatherv as an example and suppose P processes are used, the address of the root' buffer for received messages is $recvbuf$, and the displacement array is $displs$. With MPI_Gatherv, one process, known as the root, gathers messages from all P processes and stores them in $recvbuf$ according to $displs$ — the i-th entry of $displs$ specifies the displacement relative to $recvbuf$ at which to place the incoming message from process i ($0 \leq i < P$), i.e., the starting address of the message from process i is

$$recvbuf + displs[i] * s, \tag{4.1}$$

where s denotes the size of the messages' data type. The maximum of a *int* type in C is denoted as INT_MAX. Since $displs[P-1] \leq$ INT_MAX, the number of elements that the root receives from the first $P-1$ processes must be no bigger than INT_MAX -1, which is about $1/(P-1)$ of the number of elements the root receives from the first $P-1$ processes when using MPI_Gather (INT_MAX$*(P-1)$). In addition, C int is represented with 32 bits on most current platforms [68]. When $P = 1024$, each process sending a few megabytes (2^{20} Bytes) can easily corrupt MPI_Gatherv's $displs$ as well as MPI_Gatherv. Hence irregular collectives face an urgent scalability issue that must be dealt with.

Scaling problems can result from a bug *inside* released MPI libraries due to the following two reasons. First, the lack of systematic testing over MPI software stack has caused scaling problems to go undetected – Type-2 problems triggered when operating on large messages have seen little test coverage [68] and the fact that Type-3 scaling problems manifest due to the combined force of parallelism scale and message size has not been adequately appreciated. Second, manifestation of some scaling problems is platform or environment dependent [20, 22, 13, 14, 6, 15] and compleletly removing them is extremely challenging. *Therefore, it is important for the library users, including both MPI application developers and the application users, to perform testing by themselves to detect potential scaling problems of MPI routines of their interest.*

4.1.1 Challenges

A scaling problem caused by breaking the aforementioned limits of irregular collectives can be fixed by MPI application developers via changing the application code so as to avoid corrupting the irregular collectives' displacement array. But application developers might not be willing to fix it when most often the application is used at small scale without breaking the limit. In addition, many — surely not all — application developers argue that MPI standard should replace all the uses of *C int* with *C long long int* to avoid the scaling problem due to integer overflow on MPI routines. However, it has been a struggle for MPI standard to make this replacement. The issue of *C int* has been discussed since at least 2011. However, MPI forum believes that developers can support large count by themselves, like by building big data types, and persists using *C int* till today to provide backward compatibility [4, 12].

Prob.	MPI Developer	App. Developer	App. Users	Our protection
1-4, 6-8	✓			✓
9		✓		✓
5, 10-13	✓			✓

Table 4.3: Who can fix the scaling problems? .

For a scaling problem whose root cause is inside MPI, MPI library developers are responsible for fixing it, but it takes time to release an official fix and sometimes even not possible due to the difficulty of platform or environment dependent bugs' reproduction. Reproducing a scaling problem is challenging since some scaling problems are platform-dependent [13, 14, 6, 15] and some occur due to an incompatible environment setting [20, 22]. Because of these reasons some scaling problems might never be reproduced [6]. After a bug is reproduced, it can still take much time to issue a fix due to the difficulties of root-cause identification and the development of a safe fix [100].

4.1.2 Our Approach

To relieve the tension among *MPI developers*, *application developers* and *application users* as shown in Table 4.3, this chapter proposes *user-side testing* to uncover scaling problems and provides a framework that non-intrusively bypasses the uncovered problems. First, we eliminate the aforementioned limits of irregular collectives: based on interception, we check if the displacement array is corrupted, i.e., if it contains negative values, recover the value if a corruption occurs, and avoid the scaling problem via either (1) chopping the communication into smaller ones or (2) building big data types. Second, we bypass scaling problems inside the MPI collectives based on testing and the same avoidance strategies.

We provide an automated testing tool set for MPI collectives that users can use to test the correctness of MPI routines of interest at large scale either when MPI is installed or when they suspect that some routines trigger scaling problems for applications built on them. If a scaling problem is detected for an MPI routine, the testing procedure reveals the problem trigger point, i.e. the parallelism scale or message size that triggers a scaling problem. When running an application, MPI routines are intercepted to dynamically check if the problem trigger is reached. If this is the case, our avoidance routine as discussed above is invoked to bypass the problem. The key contributions of this chapter are:

- It makes a clear classification of scaling problems, and this classification leads to a useful observation — testing for Type-3 scaling problems does not necessarily require a large scale supercomputer if we exploit the interplay between message size and parallelism scale.

- It establishes the necessity of *user-side testing* to manifest scaling problems inside MPI collectives. We uncover two kinds of Type-3 scaling problems as shown in Table 4.2: (1) an inherent defect in MPI standard on irregular collectives that impacts eight MPI routines; and (2) four hidden scaling problems inside the released MPI libraries including OpenMPI and MPICH.

- It provides a protection layer to avoid scaling problems without requiring any changes to the MPI library or user programs. It is an immediate remedy when an official fix is not readily available.

```
int MPI_Collective (…) {
        if (true == check_problem_trigger)
                MPI_Collective_Fix (…)        // invoke our fix
        else
                PMPI_Collective (…)           // invoke the default
}
```

Figure 4.1: Avoiding scaling problems via interception.

- It evaluates the practicality of our protection layer consisting of three potential avoidance strategies for four representative MPI collectives.

4.1.3 Overview

To affect a non-intrusive fix, we need the following: (1) *problem trigger* which is the scale at which a scaling problem manifests itself; and (2) *an avoidance* that alters the execution to avoid the problem. Once both of them are known we intercept an MPI Collective as shown in Figure 4.1. The interception permits the default collective $PMPI\_Collective()$ only when a scaling problem's trigger is not reached; otherwise, it invokes the avoidance routine $MPI\_Collective\_Fix()$.

The trigger of the scaling problems caused by the corruption of the displacement array of irregular collectives is obvious: it is when at least one element in the array is negative (corrupted). To identify the triggers for other problems, we employ *testing*. As both Type-2 and Type-3 scaling problems relate to the message size, they can be triggered even on a small cluster by testing using large message sizes. Testing not only tells us if a scaling problem exists, it also identifies the scale that triggers the problem.

The avoidance we develop either (1) replaces the default large scale communication

Symbol	Meaning
n	Element count in one message
s	Size of the data type in bytes
P	Total number of processes
G_b	Global data buffer size in bytes
G_e	Global data buffer size in *element count*, $\frac{G_b}{s}$

Table 4.4: Notations.

specified by multiple communications at a smaller scale or (2) exploits the interplay between *element count* and *data type size*, whose product equals the message size, by building a big data type. Without involving uncovered details, we just give an example of one strategy for the above approach (1). As shown in Table 4.1, MPI_Gather (Prob. 1) breaks when the message size is 4KB when running with 64 processes. Suppose users use it at 8 KB message size with 64 processes. By testing we supposedly get the *maximum workable message size* like 3KB. Our avoidance bypasses it by carrying out two rounds of 3KB message transfers and one round of 2KB transfer.

4.2 Manifesting Scaling Problems

4.2.1 Basics of MPI Collectives

Table 4.4 lists the notations we use. With a collective, P processes communicate with each message having n elements whose data type's size is s. MPI collectives can be classified into four types: *All-to-All*, *All-to-One*, *One-to-All*, and *other collectives* that do not fit into any type above [120]. Table 4.5 lists the collectives considered in this chapter, which covers both the blocking/non-blocking regular collectives and blocking/non-blocking irregular collectives. **Root process** is the process holding the final result for All-to-One

Type	Function	G_b
One-to-All	MPI_Bcast(I)	sn
	MPI_Scatter(I, v)	snP
All-to-All	MPI_Allgather(I, v)	snP
	MPI_Allreduce(I)	sn
	MPI_Alltoall(I, v)	snP
	MPI_Reduce_scatter	sn
All-to-One	MPI_Gather(I, v)	snP
	MPI_Reduce(I)	sn

Table 4.5: MPI collectives and their global data buffer size. If (I) follows a collective, the collective has a non-blocking variation; if v follows a collective, the collective has an irregular variation.

collectives and the one holding the data sent out to all processes for One-to-All collectives. No root exists in symmetrical All-to-All collectives.

Global data buffer stands for the data buffer whose contents are either contributed by or distributed to all processes. The global data buffer is the root's receiving buffer for All-to-One and the root's sending buffer for One-to-All. Each data buffer for All-to-All is a global data buffer, but we refer to the largest data buffer when discussing its size. We denote the global data buffer size in *bytes* as G_b and in terms of *element count* as G_e. G_b can be expressed as functions of s, n, and P (see Table 4.5), and $G_e = G_b/s$.

4.2.2 Testing

Experiment Setup. Table 4.6 provides an overview of our experiment setup. The MPI libraries we study include MPICH 3.1.3, OpenMPI 1.7, and OpenMPI 1.10.0. MPICH 3.1.3 runs over of TH-express—a specialized high performance network interconnect of Tianhe-2 [104]. OpenMPI on the other hand is run by using TCP/IP over TH-express, which can be achieved by assigning *btl* framework to *tcp* [62]. We modified collective

Platform	Tianhe-2, each node having 2 E5-2692 processors (24 cores) and 64GB memory
MPI	MPICH 3.1.3 based on InfiniBand OpenMPI 1.7 & 1.10.0 based on TCP/IP
Programs	OMB adapted for automated testing

Table 4.6: Experiment Setup.

benchmark set from the OSU micro-benchmark suite (OMB) [23] so that it enables us to set a time limit for the test at each message size and to vary n and s.

Testing Scheme. We perform testing by scaling both parallelism and message size. To increase parallelism P, we increment the number of nodes while allocating 24 processes per node (1 process per core). To increase message size sn we increase n while fixing s – for MPI_Reduce, MPI_Reduce_scatter, and MPI_Allreduce, $s = 4B$ as data type MPI_FLOAT is used and for the rest $s = 1B$ as MPI_CHAR is used. We perform testing for $P = 48$ and 96. Given P, *the testing is fully automated via a Linux shell script* that submit time-limited tests (jobs) to job scheduler — each test is denoted as $test(n, t)$, where t stands for the time limit requested to run the job. If a test crashes or cannot finish in time t, a failure is reported. Testing steps are:

– *Step 1* iterates until (1) the message size grows to INT_MAX, the maximum allowed by its data type, (2) memory limit is hit [1] or (3) a failure is encountered. This process starts from $n = 1$ with $t = 60$ seconds as it is far more than enough to complete a transfer of 1 or 4 bytes with 60 seconds for any collective in our configuration. If it succeeds, we update t as the real time cost of the current run. We continue tests by increasing message size via $n \leftarrow 2 * n$ as well as sufficiently increasing the time limit via $t = 10 * t$. In this

[1] During execution if a test runs out of memory due to the huge memory footprint, we can identify this error from the error logs showing some processes being killed by the kernel, or more specifically by OOM killer.

step, the testing procedure terminates without finding any scaling problem if condition (1) is met; the testing with the next P configuration starts if condition (2) is met; and the testing proceeds to *Step 2* upon condition (3) with the detected largest n that passes the test, denoted as n'_s.

– *Step 2* refines n'_s found in Step 1 as follows. We know n'_s succeeds and $2n'_s$ fails, so we test at interval $\Delta = n'_s/f$ (we use $f = 16$ in our testing and users can vary f to configure Δ to satisfy their requirement) in the range $[n'_s + \Delta, 2n'_s)$. Finally the largest n that passes the test at interval Δ is found. The **safe bound**, n_s, is the largest n that passes the test under our testing scheme for the given s and P, i.e., the test is able to pass if $n \leq n_s$ but it fails if $n > n_s + \Delta$.

4.2.3 Scaling Problems Uncovered

Using the above testing scheme we uncovered scaling problems shown in Table 4.2. These scaling problems can result from (1) *Displacement array corruption outside MPI library* that impacts all 8 irregular collectives from any MPI library and (2) *A corruption inside MPI library*, which maps to 4 corrupted functions in various released MPI libraries.

Outside MPI (Prob. 9). In the default setting that allocates 24 processes per node, all irregular collectives except MPI_Alltoallv(I) are found to be susceptible to this problem, and MPI_Alltoallv(I) are not as it hits the memory limit first due to its higher memory consumption. The scaling problem occurs with the use of MPI_Alltoallv(I) when we reduce the memory consumption by allocating one process per node. These scaling problems are invariably caused by an integer overflow error when calculating the *C int* displacement array for irregular collectives. On the other hand, even this error does not

Root cause location	Prob.	Safe bound n_s (Δ)	
		$P = 48$	$P = 96$
Outside MPI	9	42M (2M)	21M (1M)
Inside MPI	10-11, 13		
	12	124M (4M)	62M (2M)

Table 4.7: Safe bounds.

occur in user code, i.e., users calculate correctly based on a larger data type like *C long long int*, the scaling problems would still occur due to *truncation error* in the data type conversion.

Inside MPI (Prob. 10, 11, 12, 13). Table 4.2 shows two collectives — each in both OpenMPI 1.7 and 1.10.0 and MPICH 3.1.3 — encounter a scaling problem due to an integer overflow inside the MPI library. Next we illustrate this problem using MPI_Igather from OpenMPI 1.10. In MPI_Igather's underlying function *ompi_coll_libnbc_igather*, the root process needs to calculate the starting address $rbuf$ for storing the message from process i with

$$rbuf = (\text{char } *)recvbuf + i * recvcount * rcvext, \tag{4.2}$$

where $recvbuf$ is the starting address of the root's receiving buffer, $recvcount$ (*C int*) is the number of elements in one message, and $recvext$ is the size of the used data type. Integer overflow occurs when $i * recvcount >$ INT_MAX, which results in a negative integer as well as an invalid address assigned to $rbuf$. Considering there are P processes in total, the problem is triggered once $n(P-1) \geq$ INT_MAX.

Safe bound. For each MPI routine having a scaling problem, we report the safe bound n_s, where test is able to pass if $n \leq n_s$ but it fails if $n > n_s + \Delta$. The safe bound for each scaling problem is reported in Table 4.7.

Class	Detector
D	$displs[i_0] < 0$
G	$G_b > B_h$ or $G_e > B_h$
X	$n > B_h$ given s and P

- $displs$, the displacement array for an irregular collective
- i_0, the index of the first corrupted element
- B_h, a bound restricted by an unknown scaling problems

Table 4.8: Scaling problem detectors.

Useful insights have been gained based on the problems reported online as well as new problems detected by us. First, manifesting a Type-3 scaling problems does not necessarily demand a supercomputer and many scaling problems can be found by interplaying the message size and parallelism scale. Second, the testing coverage of MPI software stack is inadequate as shown by newly uncovered problems and scaling problems resulting from platform and environment dependency are hard be removed. Third, it takes time to obtain an official fix and sometimes the fix is not possible considering the platform-dependent scaling problems that are hard to be reproduced [6] as well as the displacement array corruption for irregular collectives. All these inspires us to propose user-side testing and to provide an easy-to-use protection layer, which acts as an immediate remedy when an official fix is not available.

4.3 Online Problem Detectors

Depending upon the difficulty of *detection*, we classify the problems into 3 classes as shown in Table 4.8: (1) Class D caused by displacement array corruption, (2) Class G triggered when the global data buffer is too big, and (3) Class X whose trigger form is not known.

4.3.1 Class *D*: Displacement Array Corruption

To detect the occurrence of a scaling problem (e.g., Prob. 9) because of displacement array *displs* corruption is very straightforward. One pass over the array is enough. Below we detail the validity of our assumptions and how we detect the corruption as well as how *displs* can be recovered.

Assumptions. We assume (1) uncorrupted values in array *displs* are non-descending; (2) $n \leq$ INT_MAX; and (3) *two's complement* is used to represent integers. Assumption (1), though not specified by MPI standard, is based on a commonly used programming convention of organizing the data in global data buffer by MPI rank. Using this convention makes programming less error-prone. Assumption (2) implies that the number of elements sent by each process is at most INT_MAX, which is specified by the standard. Assumption (3) is true on nearly all modern machines [40].

Detector D. A corruption is detected if $displs[i_0] < 0$, where $0 \leq i_0 \leq P-1$ and the i_o^{th} entry is the first element being corrupted.

Proof. We denote the actual value of $displs[x]$ as a_x and its correct value as c_x ($c_x > 0$), where $0 \leq x \leq P-1$. Let the first corrupted value in *displs* be $displs[i_0]$. As a correct *displs* is non-descending, all elements following the i_0^{th} element must be corrupted. Our goal is to prove $a_{i_0} < 0$ based on the known relations below:

$$\begin{cases} c_{i_0} & = & c_{i_0-1} + n, \\ a_{i_0} & \neq & c_{i_0}, \\ a_{i_0-1} & = & c_{i_0-1}, \end{cases} \tag{4.3}$$

From $a_{i_0-1} = c_{i_0-1}$, we get

$$0 \leq a_{i_0-1} \leq \text{INT\_MAX} \tag{4.4}$$

As $c_{i_0} = c_{i_0-1} + n$ and $n \leq \text{INT\_MAX}$, we get

$$\text{INT\_MAX} < c_{i_0} \leq 2\ \text{INT\_MAX}. \tag{4.5}$$

As mentioned earlier, the corruption of *displs* can result from a *positive overflow* of integer addition in user code as well as *truncation error* during type conversion from *long long int* to *int*. In either case, based on two's complement system, we have:

$$a_{i_0} = c_{i_0}\%(2\ \text{INT\_MAX} + 2) - (2\ \text{INT\_MAX} + 2), \tag{4.6}$$

and hence

$$a_{i_0} = c_{i_0} - (2\ \text{INT\_MAX} + 2). \tag{4.7}$$

Combining Equations 4.5 and 4.7, we get $a_{i_0} \leq -2 < 0$ and thus the validity of Detector I is proved.

Recovery. Suppose the *a*ctual values of array *displs* are a_0, a_1, a_2, ..., a_{P-1} and the supposed *c*orrect values are c_0, c_1, c_2, ..., c_{P-1}. Upon two's complement system, we have:

$$a_i = c_i\%(2\ \text{INT\_MAX} + 2) - (2\ \text{INT\_MAX} + 2). \tag{4.8}$$

This implies that for a corrupted array the actual values will have several *segment*s, where

the actual values are sorted increasingly in the range of $[-\text{INT\_MAX} - 1, \text{INT\_MAX}]$. We can always recover *displs* based on the corrupted values as below: (1) if $i = 0$,

$$c_i = a_0 \ ; \tag{4.9}$$

(2) else if $i > 0$ and $a_i \geq a_{i-1}$

$$c_i = c_{i-1} + a_i - a_{i-1} \ ; \tag{4.10}$$

and (3) else

$$c_i = c_i + a_i + 2\text{INT\_MAX} + 2 - a_{i-1} \ ; \tag{4.11}$$

Proof. Let $[s_0, s_1 - 1]$ be the index range of one *segment* while c_i and a_i denote the correct and actual values respectively of the element at index i in *displs* such that $i \in [s_0, s_1 - 1]$ and $s_1 < P$. First, we guarantee $c_0 = a_0$ as $a_0 = displs[0] > 0$. Secondly, we consider the case of inside the segment, i.e. when $i > 0$, $a_i \geq a_{i-1}$. When corruption doesn't occur, Equation 4.10 holds for sure. Below justifies when corruption occurs. The *correct value* c_i can be expressed as

$$c_i \quad = \quad k_i * (2 \ \text{INT\_MAX} + 2) + C_i, \tag{4.12}$$

where k_i is an integer and $\text{INT\_MAX} + 1 \leq C_i \leq 3 \ \text{INT\_MAX}$. Since in the same segment $c_i \in [-\text{INT\_MAX} - 1, \text{INT\_MAX}]$, we guarantee that $k_{i-1} = k_i$ when $s_0 < i \leq s_1$. Based

on Equation 4.8, we proved

$$a_i - a_{i-1} \;=\; c_i - c_{i-1} \;=\; C_i - C_{i-1}, \tag{4.13}$$

and thus Equation 4.10 is proved. Lastly, let's consider the last case, i.e., the relationship among c_{s_1}, c_{s_1-1}, a_{s_1} and a_{s_1-1}. In this case, we have

$$\begin{aligned} c_{s_1-1} &= k_{s_0} * (2\ \text{INT\_MAX} + 2) + C_{s_1-1}, \\ c_{s_1} &= k_{s_0} * (2\ \text{INT\_MAX} + 2) + C_{s_1} \\ &\quad +2\ \text{INT\_MAX} + 2. \end{aligned} \tag{4.14}$$

Still based on Equation 4.8, we get

$$a_{s_1} - a_{s_1-1} \;=\; C_{s_1} - C_{s_1-1}, \tag{4.15}$$

while

$$c_{s_1} - c_{s_1-1} \;=\; C_{s_1} - C_{s_1-1} + 2\ \text{INT\_MAX} + 2. \tag{4.16}$$

Hence, we obtain Equation 4.11. The proof is complete.

4.3.2 Class *G*: Global Data Buffer Too Large

This class of scaling problem manifests when the global data buffer size exceeds a certain *bound* B_h. For example, Prob. 5, 10, 11, and 13 fall in this class.

Detector G: $G_b > B_h$ or $G_e > B_h$, i.e., the *global data buffer size*, evaluated in either *bytes* or *elements*, exceeds a bound caused by an unknown scaling problem.

This problem trigger is built based upon the analysis of certain scaling problems – Prob. 5, 10 and 11. Prob. 5 is a Type-3 scaling problem that was found in MPI_Allgather in MPICH2 [133]. It was tracked down to an integer overflow that caused a non-optimal communication algorithm to be selected and this leads to serious performance degradation. Its problem trigger relation is

$$snP > \text{INT\_MAX}. \tag{4.17}$$

The triggers of Prob. 10 and 11 can be expressed as

$$n(P-1) \approx nP > \text{INT\_MAX}, \tag{4.18}$$

where $P \gg 1$. *All these problem triggers represent cases where the global data buffer size exceeds a certain bound.*

Based on the global data buffer size, we can classify MPI collectives into two types: (1) collectives with $G_b = snP$ including MPI_Alltoall(I,v), MPI_Allgather(I,v), MPI_Gather(I,v), and MPI_Scatter(I,v), whose trigger (*Type-3*) can be expressed as

$$nP > B_h, \text{or } snP > B_h; \tag{4.19}$$

and collectives with $G_b = sn$ including MPI_Allreduce(I), MPI_Reduce(I), MPI_Reduce_scatter(I) and MPI_Bcast(I), whose triggers (*Type-2*) are

$$n > B_h, \text{or } sn > B_h. \tag{4.20}$$

1	2	3	Detector	Type
(s_s, P_s)	$(s_s, 2P_s)$	$(2s_s, P_s)$		
n_s	$n_s/2$	$n_s/2$	$snP > s_s n_s P_s$	3
	$n_s/2$	n_s	$nP > n_s P_s$	
	n_s	$n_s/2$	$sn > s_s n_s$	2
	n_s	n_s	$n > n_s$	

Table 4.9: Detector G's lookup table.

Identifying class G and its detector. Based on one round of testing, we can get the safe bound n_s given (s_s, P_s, Δ). To tell whether such scaling problem is from Class G, we simply check two additional safe bounds at different (s, P) configurations by varying each parameter – $(2s_s, P_s)$ and $(s_s, 2P_s)$ as given in Table 4.9. To avoid unnecessary brute-force stress testing on finding these two additional safe bounds, we verify if the safe bound is $n_s/2$ or n_s. If the test passes at $n_s/2$ and fails at $(n_s + \Delta)/2$, the safe bound is $n_s/2$; otherwise, we continue checking n_s: if the test passes at n_s while failing at $n_s + \Delta$, the safe bound is verified to be n_s. If all the safe bounds match any row of Table 4.9, we claim this problem is from Class G and its detector is given in the fourth column. Otherwise, it falls in class X as discussed below.

4.3.3 Class *X*: Trigger Form Not General

Class X represents scaling problems that cannot be quantitatively expressed using a general form. Although Prob. 1, 2, 3, 4, 7 were reported, we cannot conclude that its trigger can be expressed in a general form like for Class G and thus we capture them in a restrictive condition.

Detector X: $n > B_h$ given s and P. Note it is not a general method; it works only within the restriction.

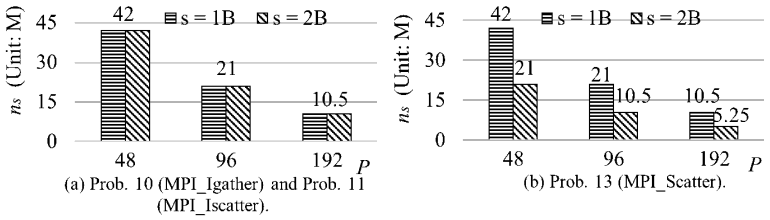

Figure 4.2: Safe bounds of G problems (Prob. 10, 11 and 13).

4.3.4 Case Studies: Class *G* and *X*

Detector D is sound enough based on proof. Here we show how to find detectors for Class G and Class X.

Class G. To check if a scaling problem is of Class G or not, we first *find* the safe bound at $(s = 1\text{B}, P = 48)$, and then *verify* the two safe bounds at $(s = 1\text{B}, P = 96)$ and $(s = 2\text{B}, P = 48)$. The detected safe bounds of Prob. 10, 11 and 13 are shown in Figure 4.2, where the required three as well as three additional safe bounds are shown so as to provide a clear picture of how the safe bounds vary given different (s, P) settings. By checking Table 4.9, we conclude that these problems are from Class G. However, their detectors are different. For Prob. 10 and 11, the detectors are the same:

$$G_e = nP > 2016\text{M}. \tag{4.21}$$

Prob. 13's detector is:

$$G_b = snP > 2016\text{MB}. \tag{4.22}$$

Class X. Similarly we found that the three safe bounds at $(s = 1\text{B}, P = 48)$, $(s = 2\text{B}, P = 48)$ and $(s = 1\text{B}, P = 96)$ are 124M, 68M and 62M respectively as shown in

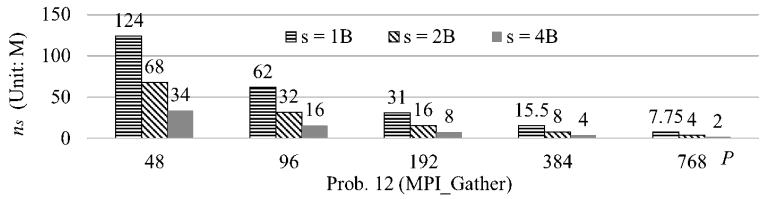

Figure 4.3: Safe bounds of an X problem (Prob. 12).

Figure 4.3. However, these do not map to any row in Table 4.9 and thus we classify this problem into Class X. Based on Figure 4.3, it follows:

$$\begin{cases} snP > 5952\text{MB} & \text{if } s = 1 \text{ and } P \geq 48 \\ snP > 6144\text{MB} & \text{if } s \geq 2 \text{ and } P \geq 96 \end{cases} \tag{4.23}$$

Note that users do not necessarily need to find the exact bound in all situations. Easily users can find a bound though overly restrictive. For example, an application uses the buggy MPI_Gather at $s = 1$B and $P = 96$. Based on testing it is easy to obtain 62M as the safe bound. Thus, we assume that the problem can occur if $n > 62$M.

4.4 Non-intrusive Avoidance

To avoid the risk of introducing other scaling problems, we keep our design clean via following protocols: (1) the avoidance of an MPI routine's scaling problem is based on the routine itself; (2) the avoidance uses the minimal number of MPI routines other than the target routine, i.e. other routines at most do some control messages' passing involving only a few bytes. For example, though avoiding MPI_Gather with MPI_Gatherv is easy, it it not allowed to avoid any problems existing in MPI_Gatherv.

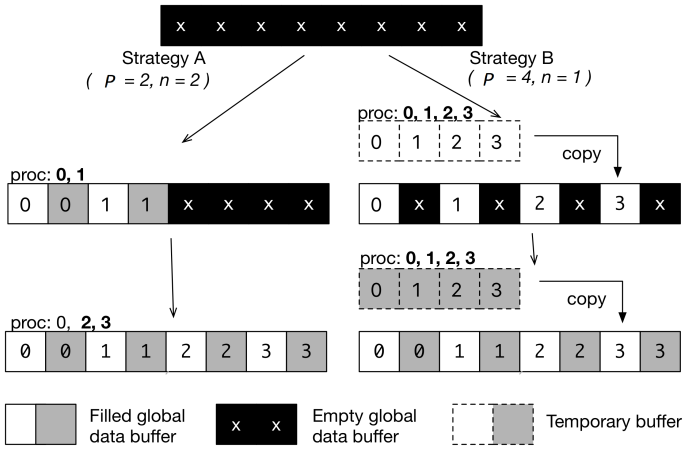

Figure 4.4: Illustration of the partitioning strategies for MPI_Gatherv ($P = 4$ and $n = 2$) by breaking down the filling process of the global data buffer. Process 0 is the root and the bug would be triggered when $nP > 4$.

4.4.1 Workaround 1: Communication Partitioning

Partitioning strategies. Consider a Type-3 scaling problem of Class G that manifests when $G_b > B_h$ (or $G_e > B_h$). An inherent workaround (*W1*) is to partition the communication such that for each sub-communication $nP \leq B_h$ (or $G_e \leq B_h$). Specifically, *W1* has two partitioning strategies: (A) shrink P while fixing n; and (B) shrink n while fixing P. Consider MPI_Gatherv, for which the scaling problem is triggered when $nP > 4$. Figure 4.4 shows a *simplified view* of how the two strategies are applied. *W1-A* creates two process groups – $\{0, 1\}$ and $\{0, 2, 3\}$. Since process 0 is the root that receives messages from all, it is present in every group. In the 2^{nd} group process 0 can be configured to send out nothing; thus the real number of processes participating in each sub-communication is still two, i.e., $P = 2$ and $n = 2$. With *W1-B*, $P = 4$ and $n = 1$ in each sub-communication. Since $nP = 4$ with either strategy, the scaling problem is avoided.

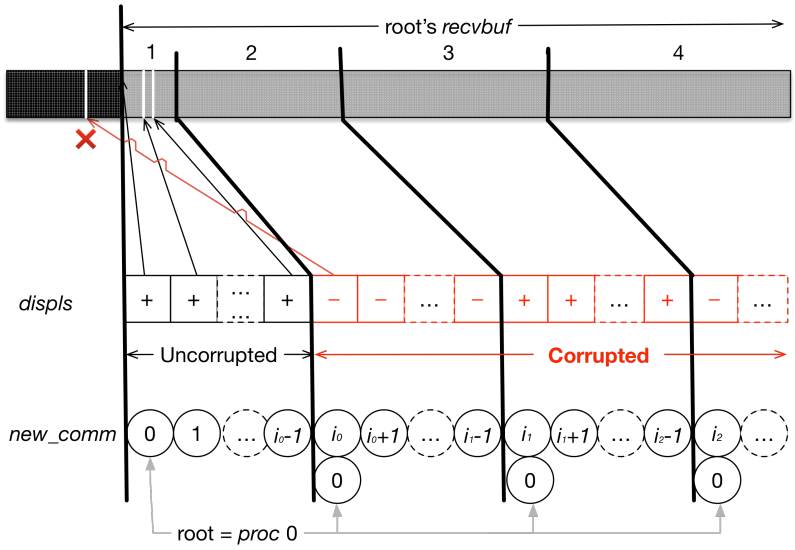

Figure 4.5: Workaround 1-A for MPI_Gatherv.

Applying Workaround 1-A to MPI_Gatherv Applying the workarounds to Class G and X is straightforward, but it involves the tricky issue of displacement array's corruption for Class D. We hence illustrate how *W1-A* works for MPI_Gatherv (Class D) here. As shown in Figure 4.5, one corrupted displacement array *displs* consists of at least two *segments* with all elements in each segment are either non-negative or negative. Each such segment maps to one segment of root process' *recvbuf* as well as a group of processes, in which the root process should be added if it is not included as it is the one that holds *recvbuf*. The communication then could be naturally partitioned, each of which is performed within one group of processes.

Constructing uncorrupted displacement array. Recall that we have recovered *displs* from corruption with all of its correct values stored in array *c*. For each sub-

communication, we construct a new displacement array ($displs2$) as

$$displs2[i] = \begin{cases} 0 & \text{if } i = 0, \\ c_{s_0+i} - c_{s_0} & \text{if } i > 0, \end{cases} \tag{4.24}$$

where the range $[s_0, s_1 - 1]$ depicts the process id range of one process group. In one run, we have $c_{s_1-1} - c_{s_0} \leq$ INT_MAX and thus $displs2$ would not be corrupted.

4.4.2 Workaround 2: Big Data Type

Building a big data type is a potential alternative strategy (*W2*) for scaling problems that are *unrelated to data type size s* such as Prob. 8, 9, and 10. With the newly created big data type of size d-bytes, an original message having x elements with each element accounting for y bytes can be converted to a new message containing xy/d elements with each element having d-bytes. That is, the number of elements in one message (n) is decreased by a factor of d. Suppose the safe bound limit for an s-irrelevant scaling problem is n_s. This could increase the safe bound from n_s to dn_s. In addition, *W2*'s performance is expected to be comparable to the original's as the cost of building new data type is trivial.

The size of big data type in byte (d) can be set as following: (1) $d = sn_s$ for *regular* collectives; and (2) $d = sn_{gcd}$ with n_{gcd} being the *greatest common divisor* of all values in $displs$ and $recvcounts$ for *irregular* collectives, which ensures that using the new data type the collective is able to work as intended. Note that for case (2) *W2* is effective only when $n_{gcd} > 1$.

Scaling problems		W1-A	W1-B	W2
Type-3	Class D	✔✘	✓	✘
	Class G	✔✘		
	Class X	✘		
Type-2		✘		

Table 4.10: Workarounds applicability: "✓" - apply; "✘" - does not apply; "✔✘" - apply with restrictions.

4.4.3 Applicability and Limitation

Table 4.10 summarizes the applicability of all strategies on various scaling problems. *W1-A* can tackle the majority of scaling problems of Class D and G from Type-3, its restriction is for MPI_Alltoall(I,v) as this routine has the highest communication complexity and partitioning the parallelism scale will only lead to complex error-prone logic. It cannot handle Class X as we use the detector $n > B_h$. It does not handle Type-2 as only message size matters for this type. *W1-B* that cuts message size is the most general avoidance that applies unconditionally. *W2* is less general compared with *W1-B* because of following limitations: (1) it only works for scaling problems that are unrelated to s; and (2) it does not work for irregular collectives when $n_{gcd} = 1$.

W2's limitation resides in its limited applicability. *W1* has limitations as well. First, non-blocking communication routines has been turned into its blocking communication using *W1-A* and *W1-B*. Second, additional memory overhead is incurred in the implementation of some workarounds like *W1-B* for MPI_Gather. Third, the performance of *W1-A* and *W1-B* is not as good as the performance of *W2*.

Scale ↓		Original		W1-A		W1-B		W2	
		n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M
P	192	10.5	2.21	256	54.00	256	54.0	272	57.38
	768	2.625	2.03	68	52.60	72	55.69	72	55.69

Table 4.11: Workarounds' effectiveness for MPI_Gatherv (D). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB.

4.4.4 Evaluation

We evaluate our non-intrusive workarounds based on 4 representative MPI routines. They stand for *all-to-one* and *all-to-all* — *one-to-all* is ignored as it is very similar to *all-to-one*, and also they represent *irregular*, *regular*, *blocking* and *non-blocking* collectives. Our default setting is the same as mentioned earlier—24 processes per node (1 process per core), $s = 1$B and $f = 16$.

Effectiveness

The effectiveness is evaluated by the degree to which a workaround can increase the *safe bounds* of the default buggy functions – the greater the safe bounds are increased the more effective is the workaround. In the evaluation, our workarounds increase the safe bounds significantly, but the workarounds' safe bounds are limited by the physical memory size. To show this point, we also report the *maximum memory consumption on one node*, denoted as R_M, which is calculated according to the MPI standard.

I. Class *D* (Prob. 9: MPI_Gatherv). As shown in Table 4.11, all three workarounds have comparable effectiveness, and their safe bounds are roughly 24 times the safe bound of the default MPI_Gatherv. *W1* and *W2* have comparable effectiveness. The workarounds do not go further because the physical memory limit is reached — note MPI

Scale ↓		Original		W1-A		W1-B		W2	
		n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M
P	192	10.5	2.21	256	54.00	256	54.00	272	57.38
	768	2.625	2.03	72	57.02	42	32.48	42	32.48

Table 4.12: Workarounds' Effectiveness for MPI_Igather (G). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB.

Scale ↓		Original		W1-B	
		n_s	R_M	n_s	R_M
P	192	31	6.75	240	50.63
	768	7.75	6.19	64	49.50

Table 4.13: Effectiveness of Workaround 1-B for MPI_Gather (X). The unit of n_s is 1 M, i.e. 2^{20}, and that of R_M is GB.

has hidden memory footprint besides the obvious R_M.

II. Class *G* (Prob. 10: MPI_Igather). Its evaluation is shown in Table 4.12. At scale $P = 192$, three workarounds are of comparable effectiveness and their safe bounds are 24+ times the default MPI_Igather's safe bound. At $P = 768$, *W1-A* is the best, *W2* and and *W1-B* are worse, where the first is limited by the memory size while the last two are not. The last two's worse performance is traced down to the error of *connection time out*, i.e., when too many processes connect to the root process that is only capable of responding a portion of the connection requests at a time due to the ongoing communication with large message sizes, some connections fail to be established within a time limit. This error doesn't negatively impact *W1-A* as each time it communicates with only a small portion of processes.

III. Class *X* (Prob. 12: MPI_Gather). The detection only works under a specific restriction as mentioned earlier. Here the restrictions are $s = 1$ and $P \geq 48$ and the

scaling problem manifests when $n \geq \frac{5952}{P}$. *W1-B* is the only workable solution. Table 4.13 shows *W1-B*'s safe bounds are 7+ times of the default's. As *W1-B* incurs 5.8 GB memory overhead, R_M is smaller compared with the previous experiments.

Performance

The performance is measured as *time cost* in seconds. As each process might have varying time costs in one run, we report both the average and the maximum. Given a (s, n, P) configuration, a collective is run Y times, where $Y = 500$ if $n <= 1M$ and $Y = 20$ otherwise. We evaluate the performance of workarounds using the *above three buggy collectives* first and then two correctly-functioning routines with *supposed* scaling problems for which all the workarounds apply.

I. Class *D*. For MPI_Gatherv, all workarounds are effective; they detect scaling problems of class D by detecting if the displacement array *displs* is corrupted, which involves checking all elements in *displs* and broadcasting the judgment to all P processes with respectively $O(P)$ and $O(logP)$ time complexity. Considering such overhead, we evaluate their performance both before and after the problem's occurrence. Figure 4.6 shows the comparison between the default MPI_Gatherv and *W1-A* before the problem occurs. Note all workarounds detect the scaling problem in the same way and thus have the same detection overhead before the problem's occurrence, so we only evaluate *W1-A*. We observe that the performance of *W1-A* is comparable to the default as the detection overhead is trivial. Figure 4.7 shows that the time costs of *W1-A* and *W1-B* grow linearly with message size as it cuts communication into roughly equal-sized pieces. *W2*'s performance is better as it retains the communication only by varying the parameter setting, but it should be noted

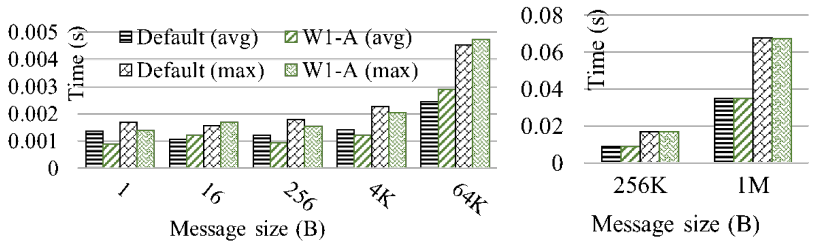

Figure 4.6: Performance comparison between W1-A and the default MPI_Gatherv (MPICH) before the scaling problem's occurrence.

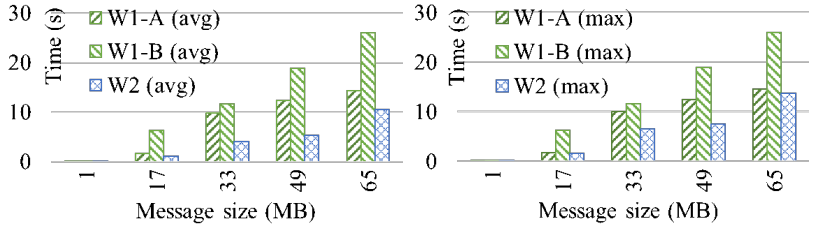

Figure 4.7: Performance comparison among the three workarounds for MPI_Gatherv (MPICH) whose scaling problem (Class D) is triggered once $sn > 2.625MB$ when $P = 768$.

it is not guaranteed to work for irregular collectives as explained previously. To make *W2* work in this case, we configure all processes transfer equal-sized messages in the experiment.

II. Class *G*. Since the problem detection is only based on checking an inequality which is far less overhead than the detection overhead discussed above, we only measure the performance after the problem's appearance. Figure 4.8 shows the performance comparison

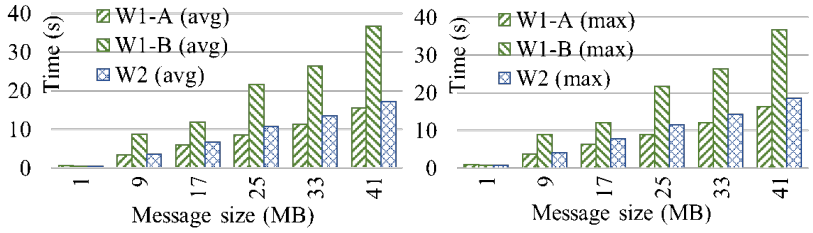

Figure 4.8: Performance comparison among the three workarounds for MPI_Igather (Open-MPI) whose scaling problem (Class G) is triggered once $sn > 2.625$MB when $P = 768$.

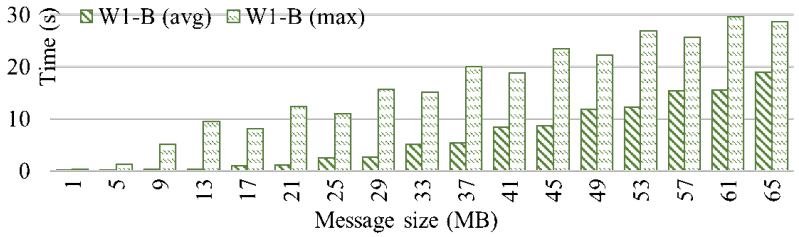

Figure 4.9: Performance trend of W1-B for MPI_Gather (MPICH) whose scaling problem (Class X) is triggered once $sn > 7.75$MB when $P = 768$.

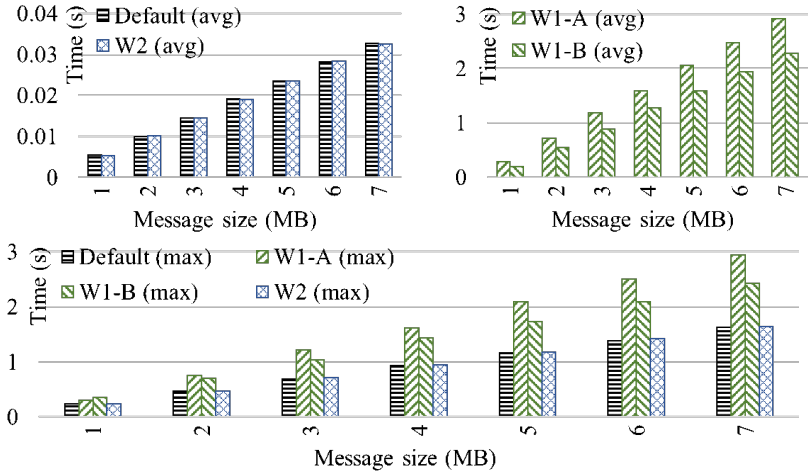

Figure 4.10: Performance comparison based on MPI_Gather (MPICH) supposing a Class G problem is triggered when $n > 128$K at $P = 768$.

of *W1-A*, *W1-B*, and *W2* for MPI_Igather. *W1-A* is better than *W2* by only a small margin, and *W1-B*'s performance is about half of *W2*'s.

III. Class *X*. As the detection is also trivial, the performance is measured only after the problem's appearance, which is shown in Figure 4.9.

IV. Evaluation of all workarounds based on correct MPI_Gather. To make sure *all workarounds apply*, we suppose a class G scaling problem would be triggered if $nP > 96$M. Figure 4.10 shows the performance comparison among all and the default. It is observed that: (1) For the default and *W2* the maximum time cost is about 50 times

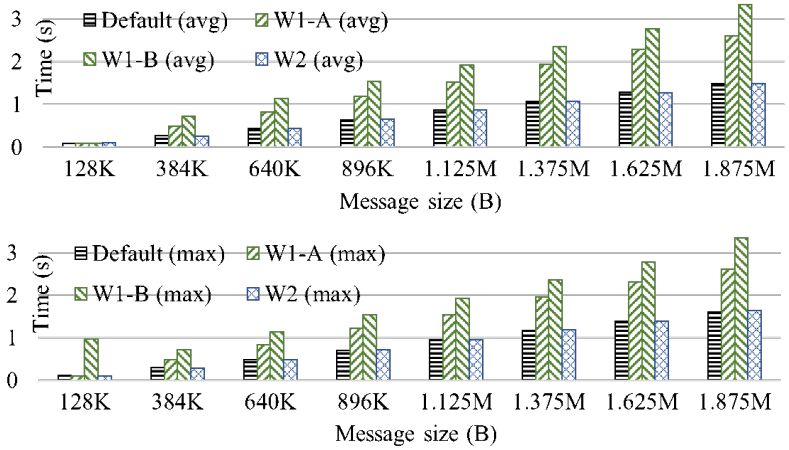

Figure 4.11: Performance comparison based on MPI_Allgatherv (MPICH) supposing a Class G problem is triggered when $n > 128\text{K}$ at $P = 768$.

the average, but for *W1-A* and *W1-B* the maximum is at most 1.2 times the average, which results from the fact that the partitioning method of *W1-A* and *W1-B* delays all the non-root processes while *W2* does not; (2) *W2* is of comparable performance to the default; (3) Based on the maximum, *W1-A*'s time cost is 1.8 times the default's and *W1-B*'s is 1.5 times the default's.

V. Evaluation of all workarounds based on correct MPI_AllGatherv. For the same reason, we assume a class G scaling problem would be triggered if $nP > 96M$. Figure 4.11 shows the performance comparison. We have following observations: (1) the average and the maximum has little difference as the communication is symmetrical, i.e., all the processes transfer the same amount of data; (2) *W2* and the default have comparable performance; (3) *W1-A* and *W1-B* respectively demands about 1.7 and 2.2 times the time cost of the default.

Evaluation summary. Before a scaling problem's occurrence, the performance of any workaround is comparable to that of the default. After its occurrence, *W2*'s perfor-

mance is comparable to the default's. *W1-A*'s and *W1-B*'s performance are worse because they partition the default communication, and their time cost increases linearly as the message size goes up. In conclusion, *W1-B* is the a general solution, and *W2* has the best performance.

4.5 Summary

We demonstrate the necessity of user-side testing. We show that testing with limited computing resources can manifest scaling problems based on the interplay between message size and parallelism scale. We provide a protection layer consisting of three potential avoidance strategies and evaluate its practicality based on representative MPI routines. Our strategies can also be easily applied to point-to-point communication.

Chapter 5

Hang Detection at Large Scale

Program hang, the phenomenon of unresponsiveness [122], is a common yet difficult type of bug in parallel programs. In large scale MPI programs, errors causing a program hang can arise in either the computation phase or the MPI communication phase. Hang causing errors in the computation phase include infinite loop [98] within an MPI process, local deadlock within a process due to incorrect thread-level synchronization [84], soft error in one MPI process that causes the process to hang, and unknown errors in either software or hardware that cause a single computing node to freeze. Due to communication dependency, an error triggered in one process (*faulty process*) gradually spreads to others, finally leading to a global hang. Errors in MPI communication phase that can give rise to a program hang include MPI communication deadlocks/failures.

5.1 Program Hang at Large Scale

It is widely accepted that some errors manifest more frequently at large scale both in terms of the number of parallel processes and problem size as testing is usually performed at small scale to manage cost and some errors are scale and input dependent [35, 86, 133, 134, 121]. A program hang at large scale freezes the execution and wastes all the requested computing resources in production runs based on batch job systems on supercomputers. The wastage could be significant espeically considering the it is not uncommon that thousands of cores or more need to be allocated for one job execution. Hence, it is urgent to devise a tool to detect hangs at runtime and terminates hanging job to avoid the wastage.

5.1.1 Challenge

Ad hoc *timeout* mechanism [11, 85, 84, 98] is a commonly used hang detection method; however, it is difficult to set an appropriate threshold even for users that have good knowledge of an application. This is because the optimal timeout not only varies across applications, but also with input characteristics and the underlying computing platform. Choosing a timeout that is too small leads to high false alarm rates and too large timeouts lead to long detection delays. The user may favor selecting a very large timeout to achieve high accuracy while sacrificing delay in detecting a hang. For example, IO-Watchdog [11] monitors writing activities and detects hangs based on a user specified timeout with 1 hour as the default. Up to 1 hour on every processing core will be wasted if the user uses the default timeout setting. Thus, a lightweight hang detection tool with high accuracy is urgently needed for programs encountering non-deterministic hangs or sporad-

ically triggered hangs (e.g., hangs that manifest rarely and on certain inputs). It can be deployed to automatically terminate erroneous runs to avoid wasting computing resources without adversely effecting the performance of correct runs.

5.1.2 Our Solution: ParaStack

To address the above need for *hang detection*, this chapter presents *ParaStack*, an extremely lightweight tool to detect hangs in a timely manner with high accuracy, negligible overhead with great scalability, and without requiring the user to select a timeout value. Due to its lightweight nature, *ParaStack* can be deployed in production runs without adversely affecting application performance when no hang arises. It handles communication-error-induced hangs and hangs brought about by a minority of processes encountering a computation error. For a detected hang, *ParaStack* provides direction for further analysis by telling whether the hang is the result of an error in the computation phase or the communication phase. For a computation-error induced hang, it pinpoints faulty processes.

ParaStack is a parallel tool based on stack trace that judges a hang by detecting dynamic manifestation of following pattern of behavior – **persistent existence of very few processes outside of MPI calls**. This simple, yet novel, approach is based upon the following observation. Since processes iterate between computation and communication phases, a persistent dynamic variation of the *count* of processes outside of MPI calls indicates a healthy running state while a continuous small count of processes outside MPI calls strongly indicates the onset of a hang. Based on execution history, *ParaStack* builds a runtime model of *count* that is robust even with limited history information and uses it to evaluate the likelihood of continuously observing a small count. A hang is verified if the

likelihood of persistent small count is significantly high. Upon detecting a hang, *ParaStack* reports the process in computation phase, if any, as faulty.

The above execution behavior based model is capable of detecting hangs for different target programs, with different input characteristics and sizes, and running on different computing platforms without any assistance from the programmer alike. *ParaStack* reports hang very accurately and in a timely manner. By monitoring only a constant number of processes, *ParaStack* introduces negligible overhead and thus provides good scalability. Finally, it helps in identifying the cause of the hang. If a hang is caused by a faulty process with an error, all the other concurrent processes get stuck inside MPI communication calls. If the error is inside communication phase, the faulty process will also stay in communication; otherwise, it will stay in computation phase. Simply checking whether there are processes outside of communication can tell the type of hang, communication-error or computation-error induced, as well as the faulty processes for a computation-error induced hang. The main contributions of *ParaStack* are:

- *ParaStack* introduces highly efficient non-timeout mechanism to detect hangs in a timely manner with high accuracy, negligible overhead, and great scalability. Thus it avoids the difficulty of setting the timeout value.

- *ParaStack* is a lightweight tool that can be used to monitor the healthiness of *production runs* in the commonly used batch execution mode for supercomputers. When there is a hang, by terminating the execution before the allocated time expires, *ParaStack* can save, on average, 50% of the allocated supercomputer time.

- *ParaStack* sheds light on the roadmap for a detected hang's further analysis by telling

whether it was caused by an error in computation or communication phase. In addition, it pinpoints faulty processes for a computation-error induced hang.

- *ParaStack* is integrated into two parallel job schedulers *Torque* and *Slurm* and we validated its performance on the world's current 2nd and 12th fastest supercomputers—*Tianhe-2* and *Stampede*. For a significance level of 0.1%, experiments demonstrate that *ParaStack* detects hangs in a timely manner at negligible overhead with over 99% accuracy. No false alarm was observed in correct runs taking about 66 hours in total at the scale of 256 processes and 39.7 hours at the scale of 1024 processes. In addition, *ParaStack* accurately identifies the faulty process for computation-error induced hangs.

5.1.3 The Case for ParaStack

Hang detection is of value to application *users* and *developers* alike. Application *users* usually do not have the knowledge to debug the application. In batch mode, when a hang is encountered, the application simply wastes the remainder of the allocated computing time. This problem is further exacerbated by the fact that users commonly request a bigger time slot than what is really needed to ensure their job can complete. If users are unaware of a hang occurrence, they may rerun the application with even a much bigger time allocation, which will lead to even more waste. By attaching a hang detection capability to a batch job scheduler with negligible overhead, ParaStack can help by terminating the jobs and reporting the information to users when it detects a hang. Thus the unnecessary waste of computing resources is avoided.

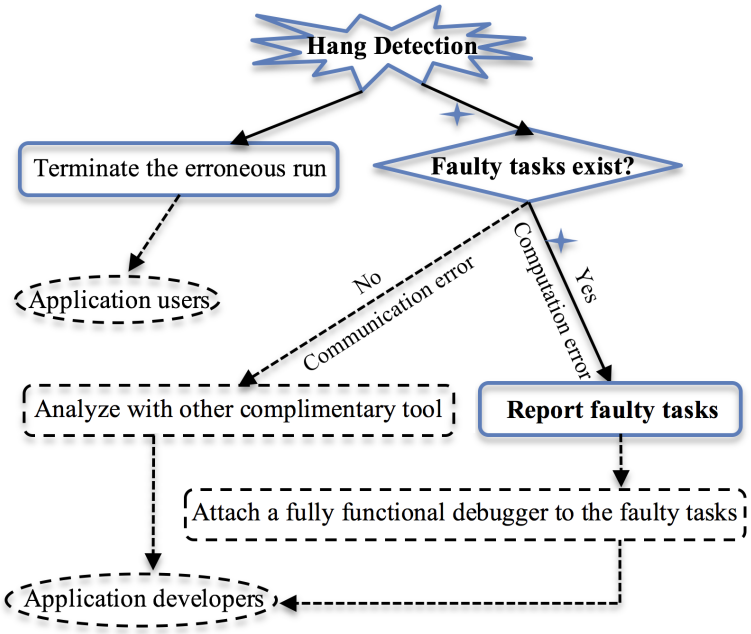

Figure 5.1: ParaStack workflow – steps with solid border are performed by ParaStack and those shown with dashed border require a complimentary tool.

Application *developers* need to detect a hang first and then debug based on the information given by *ParaStack*. First, knowing whether the hang-inducing error is in computation or communication sheds light on the direction for further analysis. To debug hangs due to communication error, such as global deadlock, is hard and it usually requires comparatively more heavyweight progress dependency analysis, communication dependency analysis or stack trace analysis. Since stack-trace analysis based tools such as STAT [35] do not require runtime information, they can be applied immediately after *ParaStack* reports a hang. In addition, the faulty process can be identified easily for a computation error induced hang, which benefits developers significantly by reducing from hundreds and thousands of suspicious processes to only one or a few.

The workflow of our tool is depicted in Figure 5.1. The path marked with blue

stars is the main focus of this chapter. If a hang happens and no faulty process is reported, we assume implicitly that the hang is caused by communication errors. For *ParaStack* users, debugging of a computation error induced hang has two phases: (1) monitoring the execution to detect hangs and report the faulty process with a lightweight diagnosis tool; and (2) debugging the faulty process with a fully functional debugger. *ParaStack* is an extremely lightweight tool for the first phase.

5.2 Lightweight Hang Detection

We begin by presenting the key observation that distinguishes the runtime behavior of a correctly functioning MPI program from one that is experiencing a hang. An MPI program typically consists of a trivial setup phase followed by a time-consuming loop-based solver phase where the latter is more error-prone. The solver loop can be viewed as consisting of a mix of computation code and communication code where the latter belongs to the MPI library. Depending upon the code being executed by a process, we classify the runtime state of the process as: ***IN_MPI*** if it is executing code in an MPI call; or ***OUT_MPI*** if it is executing non-MPI code. At any point in time, each process can be only in one state. Further ***OUT_MPI* significance**, denoted as S_{out}, is defined as the fraction of an application's parallel processes that are in state *OUT_MPI* at a given time. Next we argue that S_{out} can be used to distinguish between *healthy state* and *hang state*.

– *Healthy runtime state* is characterized by S_{out}'s periodic pattern. Parallel processes run in and out of MPI functions repeatedly in a healthy run. Thus, a healthy process frequently switches between states ***IN_MPI*** and ***OUT_MPI***. Because of the loop struc-

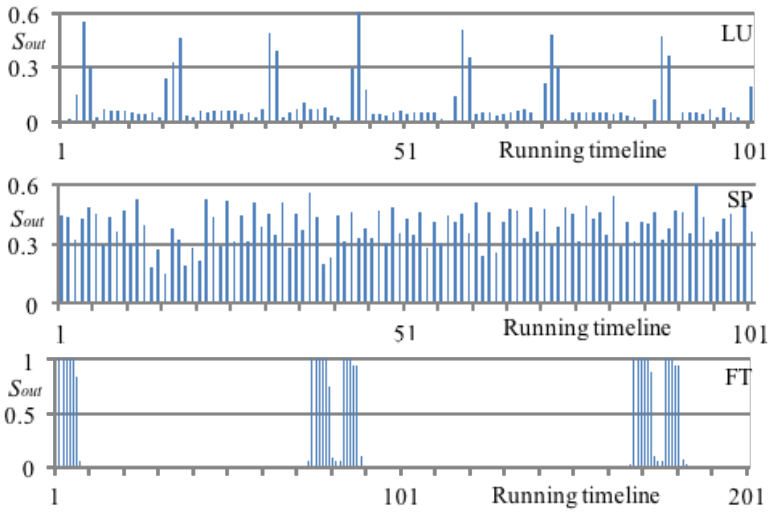

Figure 5.2: Dynamic variation of S_{out} observed from 3 benchmarks: LU, SP, FT from NPB suite. All are executed with 256 processes at problem size *D*.

ture, parallel processes are expected to show a repetitive pattern in terms of how they flip state from one to the other. Thus, S_{out} is expected to vary over time in a periodic pattern. Figure 5.2 shows the periodic variation of S_{out} in healthy executions of 3 benchmarks: LU, SP, and FT from NPB suite [17]. This is obtained by repeatedly checking S_{out} at fixed time interval of 1 millisecond. We see that the length of the period varies as it is influenced by factors such as problem size and application type.

– *Hang runtime state* is characterized by a persistently low S_{out}. If a hang is caused by a *computation error*, the majority of processes in state *IN_MPI* should form a tight communication dependency on the faulty processes and the faulty processes in state *OUT_MPI* should be in the minority. If a hang is caused by a *communication error*, all processes should be in state *IN_MPI* and thus S_{out} should be 0 persistently. Figure 5.3 plots S_{out} during a run of LU benchmark during which a hang is encountered – the dynamic variation ceases and S_{out} is very low after the hang's occurrence. *Thus, the health of an*

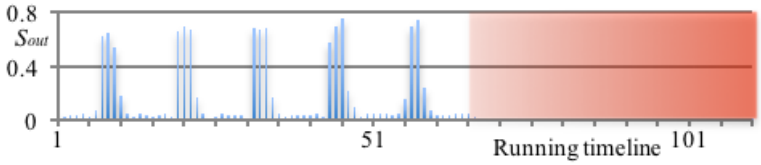

Figure 5.3: The S_{out} variation of a faulty run of LU, where a fault is injected on the left border of the red region.

application can be judged by looking for consecutive observations of very low S_{out}.

Depending upon whether the utilized function is blocking or not, we classify MPI communication styles into 3 types: *blocking* style, i.e. a blocking communication function; *half-blocking*, i.e. a non-blocking communication followed by a blocking function like MPI_Wait to wait for its completion; and *non-blocking* style where a non-blocking communication is performed followed by a check for completion using a busy waiting loop using non-blocking message checking function like MPI_Test. Our characterization of runtime state is able to detect hangs for programs only using the first two styles of communication. For a program that uses a mix of different communication styles, the lesser the use of third communication style the more useful is our approach. For example, HPL uses a mixed style with a small portion of the program in the third style, processes can get stuck at multiple sites upon a program hang and thus a significant fraction of processes would ultimately stay in blocking functions and thus be in *IN_MPI* while some may flip states forever in busy-waiting loops. Our observation is still useful in this case.

Putting S_{out} into Practice Precisely tracking S_{out} will require monitoring the runtime state of all processes continuously and this will lead to high overhead. To achieve our objective of developing a lightweight hang detection method, we neither monitor all processes nor

do we monitor their states continuously. In particular, we determine the state of *constant number of processes* (say C), at *fixed time intervals* (say I), and compute S'_{out} that denotes the fraction of C processes that are in state OUT_MPI. A hang is reported if S'_{out} is observed to be persistently low, i.e., no bigger than a threshold t, for K consecutive intervals.

Now the next challenge is determine a selection of the values for C, I, K and t. To simply the discussion, we fix C at 10 processes – this choice is out of performance considerations and its justification is given later in Section 5.2.3, and fix t at 0 as it is rare that the faulty process happens to be among the randomly selected C processes in a small number of runs. Let us first consider a simple scheme in which the hang detection algorithm a priori fixes the values of both I and K. In fact this scheme is similar in spirit to the commonly used fixed timeout methods [11, 85, 84, 98] that avoid the complexities of chosing the timeout value.

Next we studied the effectiveness of this simple scheme by studying its precision, i.e. studying: (a) accuracy of catching real hangs; and (b) false positive rate, i.e. detection of hangs when none exist. In this study we used two values for I (400ms and 800ms), two values for K (5 times and 10 times) and then ran experiments for three applications (FT, LU, SP) on two platforms (Tianhe-2 and Tardis). The results obtained are given in Table 5.1 and by studying them we observe the difficulty of setting the (I, K) parameters for different platforms, different input sizes, and different applications. In particular, we observe the following:

- (Platforms: Tianhe-2 vs. Tardis) Consider the case for FT at input size *D*. For (I_1, K_1) while on Tianhe-2 all actual hangs are correctly reported, on Tardis false hangs are

Platform →	Tianhe-2						Tardis								
Benchmark(Input size) →	FT(D)			FT(E)			FT(D)			LU(D)			SP(D)		
Metrics →	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*
$I_1 = 400ms,\ K_1 = 5\ times$	1.0	0.0	3.3	0.0	1.0	—	0.0	1.0	—	0.0	1.0	—	0.3	0.7	2.0
$I_2 = 400ms,\ K_2 = 10\ times$	1.0	0.0	8.1	1.0	0.0	10.9	0.9	0.1	6.5	1.0	0.0	5.3	1.0	0.0	5.1
$I_3 = 800ms,\ K_3 = 5\ times$	1.0	0.0	7.2	1.0	0.0	11.7	0.8	0.2	7.0	1.0	0.0	3.9	1.0	0.0	3.9
$I_4 = 800ms,\ K_4 = 10\ times$	1.0	0.0	13.2	1.0	0.0	17.4	1.0	0.0	10.2	1.0	0.0	10.7	1.0	0.0	8.6

Table 5.1: Adjusting the timeout method to various benchmarks, platforms and input sizes at scale 256 based on 10 erroneous runs per configuration. *Metrics*: *AC* – accuracy; *FP* – false positive rate; *D* – average response delay in seconds, i.e. the elapsed time from when the fault is injected to when a hang is detected.

reported during the correct execution phase (i.e., before a hang actually occurs) in all 10 runs.

- (Input sizes for FT: D vs. E) On Tianhe-2 though (I_1, K_1) has a 100% accuracy for FT at input size D, for input size E the accuracy drops to 0% and false positive rate goes up to 100%.

- (Target Application: LU and SP vs. FT) For parameter settings (I_2, K_2) and (I_3, K_3), on Tardis though the accuracy for LU and SP is 100%, the accuracy for FT is less and false alarms are reported.

Clearly the above results indicate that fixed settings of (I, K) are not acceptable and thus a more sophisticated strategy must be designed. We observe that no fixed setting of parameters will work for all programs, on all inputs, and different platforms. Therefore the choice of parameters must be made based upon on the runtime characteristics of an application on a given input and platform. We cannot leave this choice to the users as they are likely to resort to guessing the parameter settings and thus will not have any confidence in the results of hang detection.

Therefore the approach we propose is one that automates the selection and tuning of I and K at runtime such that hangs can be reported with high degree of confidence. In fact the approach we propose allows the user to specify the desired degree of confidence and our runtime method ensures that a hang's presence is verified to meet the specified desired degree of confidence. The details of this method are presented next.

5.2.1 Model Based Hang Detection Scheme

The basic idea behind our approach is as follows. We randomly sample S_{out} at runtime to build and maintain a model and detect hangs by checking S_{out} against the model.

Random sampling of S_{out}. Variation of S_{out} over time is composed of many small cycles, and all cycles exhibit similar trend over time. Suppose the cycle time is C_t. If we randomly take a sample from a time range of NC_t where $N \in \mathbb{N}^+$, no matter how N varies it is clear this randomly observed S_{out} will follow the same distribution denoted as $F(S_{out})$, considering the similarity across cycles. Such random sampling can be achieved by inserting a good *uniformly generated random* time step, denoted as r_{step}, that makes the next sample fall at any point in one or several cycles, between two consecutive samples.

Suppose I is the *maximum time interval*, and $rand(I)$ is a uniform random number generator over $[0, I]$. We make $r_{step} = rand(I) + I/2$ and thus the sampling interval ranges over $[I/2, 3I/2]$ with an average of I. An ideal model can be built either when $I = NC_t$ or when the I is way bigger than C_t so that the sampling is approximately random rather than time-dependent.

Automatically tuning *I*. Hand-tuning I is undesirable and impractical as C_t varies across different applications, input sizes, and underlying computing platforms. Instead, we can achieve approximate random sampling through enlarging I as below. We design an automatic method to enlarge the maximum interval I in the early execution stage by checking the samples' randomness. *If the sampling is statistically found to lack randomness, we double I, as a bigger I leads to better randomness, and then re-evaluate the randomness.* Below details the method we use to check the sampling's randomness.

Runs test [123] is a standard test that checks a randomness hypothesis for a two-valued data sequence. We use this to judge the randomness of a sample sequence. Given a sample sequence of S_{out}, we set the average of samples as *boundary*. Samples bigger than or equal to the boundary are coded as *positive* (+) and samples smaller than that as *negative* (-). A *run* is defined as a series of consecutive positive (or negative) values. Under the assumption that the sequence containing N_1 positives and N_0 negatives is random, the *number of runs*, denoted as R, is a random variable whose distribution is approximately normal for large runs test. Given a significance level 0.05, for small runs test ($N_1 \leq 20, N_0 \leq 20$), we can get a range for the *assumed correct* number of runs via table in [123], i.e. the non-rejection region. If R is beyond this range, we reject the claim that the sequence is random and thus relax I. On the other hand, if either $N_1 \leq 1$ or $N_2 \leq 1$ and thus the non-rejection region is not available, we also assume the sampling is not random to avoid the risk of failing to identifying a non-random sampling process.

For example, consider an MPI program running with 10 processes and thus the possible values of S_{out} are 0.1, 0.2, 0.3, ..., 1.0. There are a sequence 16 samples as follows

$$0.2\ \ 0.1\ \ 0.1\ \ 0.2\quad 0.1\ \ 0.1\ \ 0.0\ \ 0.0$$

$$0.8\ \ 0.9\ \ 1.0\ \ 0.8\quad 0.9\ \ 0.1\ \ 0.9\ \ 0.9,$$

which is equivalent to the two-valued sequence

$$- - - - - - - - - + + + + + - + + .$$

Its boundary is 0.44375 with $N_1 = 7$, $N_0 = 9$ and $R = 4$. The assumed correct range is

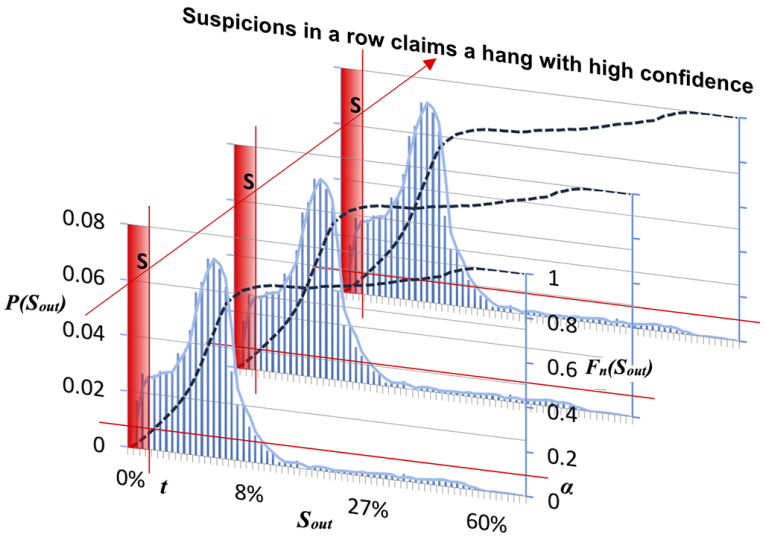

Figure 5.4: Hang detection. Three panels show the empirical distribution of randomly sampled S_{out} of LU, where the red region shows the suspicion region, the blue curve shows the probability density function $P(S_{out})$, and the dashed black curve shows the cumulative distribution function $F_n(S_{out})$. The red arrow crosses the suspicion region 3 times meaning 3 consecutive observations of suspicion.

(4, 14) and the number of runs 4 is not inside the range, so we claim the sampling is not random and double I.

Note the model needs to be renewed when I is doubled. The size of old samples collected at average time interval I is 2 times of the size if the old samples were collected at interval $2I$. Therefore, we cut the sample size by half.

Suspicion of hang. As samples accumulate, an empirical cumulative distribution function, denoted as $F_n(S_{out})$, can be built, where n is the number of samples. Given a probability $\hat{p}$, we can obtain $t = F_n^{-1}(\hat{p})$. A *suspicion* is defined as $S_{out} \leq t$, i.e. a very low

S_{out}. The observed values can be classified into a pair of opposite random events:

$$\begin{cases} A : \text{Suspicion} & \text{if } S_{out} \leq t, \\ \overline{A} : \text{Non-suspicion} & \text{if } S_{out} > t, \end{cases}$$

Note $\hat{p}$ is selected dynamically to ensure robustness at various sample sizes and will be discussed in Section 5.2.2.

Significance test of hang. A single suspicion does not justify a hang's occurrence; instead, a continuous detection of suspicions indicates a hang with high confidence. We can quantify the number of suspicions (A) before the first observation of a non-suspicion ($\overline{A}$) as a *geometric distribution*. The probability of $Y = y$ observations of event A before the first observation of $\overline{A}$ can be expressed as follows:

$$P(Y = y) = q^y \cdot (1 - q)$$

where q is an estimation of the *true suspicion probability*, denoted as p, by adapting $\hat{p}$ and will be discussed in Section 5.2.2. Let us consider the following null and alternate hypothesis:

$$\begin{cases} H_0 : & \text{The MPI application is healthy,} \\ H_1 : & \text{The MPI application has hung.} \end{cases}$$

Under H_0, the probability to observe at least k consecutive As is

$$\begin{aligned} P_{H_0}(Y \geq k) &= 1 - \textstyle\sum_{y=0}^{k-1} q^y \cdot (1 - q) \\ &= q^k. \end{aligned}$$

Given the confidence level $1-\alpha$, we reject H_0 and accept H_1 if $P_{H_0}(Y \geq k) \leq \alpha$, i.e., as below:

$$\begin{aligned} q^k &\leq \alpha \\ \Rightarrow \quad k &\geq \lceil log_q \alpha \rceil \ . \end{aligned}$$

Hence a hang would be reported at a confidence level of $1-\alpha$ if $\lceil log_q \alpha \rceil$ times of consecutive suspicions are encountered as depicted in Figure 5.4. The theoretical worst case time cost required to detect a hang is $I \cdot \lceil log_q \alpha \rceil$, considering a few *normal suspicions* in the correct phase may appear before a hang really appears.

5.2.2 Robust Model with a Limited Sample Size

Ideally, we would have $\hat{p} \approx p$ if the sample size is large enough. We can just apply $q = \hat{p}$ to the model. However, the problem is that the sample size can not be large enough as the sample size always grows from 0, and the assignment $q = \hat{p}$ thus would only make a bad hang detection model. To overcome this difficulty, we introduce a method for a achieving a credible q for each level of sample size.

Since we only care about *suspicion* versus *non-suspicion*, the sampling can be viewed as a Bernoulli process, i.e. $X_i \overset{i.i.d.}{\sim} Ber(p)$, where X_i is the i-th sample. By the rule of thumb [25], when $n\hat{p} > 5$ and $n(1-\hat{p}) > 5$, $\hat{p}$ follows

$$\hat{p} = \frac{\sum_{i=1}^{n} X_i}{n} \overset{\circ}{\sim} N(p, \frac{p(1-p)}{n}).$$

Its 95% confidence interval is $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$. If we estimate p with an error no bigger than e, i.e. $\hat{p} \in [p-e, p+e]$, at 95% confidence, we have $1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \leq e$. The minimal

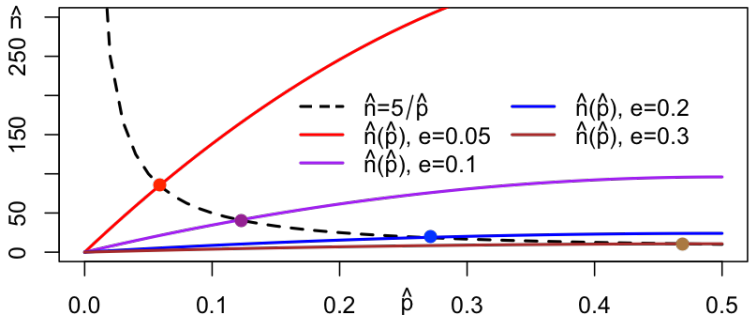

Figure 5.5: Relation among sample size, suspicion probability and tolerance error, where $\hat{n}(\hat{p}) = \frac{3.8416}{e^2}\hat{p}(1-\hat{p})$.

sample size to justify $\hat{p}$ is

$$\hat{n} = max\{\frac{5}{\hat{p}}, \frac{5}{1-\hat{p}}, \frac{3.8416}{e^2}\hat{p}(1-\hat{p})\},$$

where $0 < \hat{p} < 1$. Because $\hat{p}$ and $1-\hat{p}$ are exchangeable and $\frac{5}{\hat{p}} > \frac{5}{1-\hat{p}}$ in $(0, 0.5]$, we only study

$$\hat{n} = f_{max}(\hat{p}) = max\{\frac{5}{\hat{p}}, \frac{3.8416}{e^2}\hat{p}(1-\hat{p})\}, \text{ where } \hat{p} \in (0, 0.5].$$

where $\hat{p}\ in(0, 0.5]$ and f_{max} is the function that gets the maximum between the given two terms.

Given a tolerance error e, our goal is to get an acceptable $\hat{p}$ that can be justified by the smallest sample size n, where the *smallest* ensures the model as soon as possible even with a small sample size. We provide 4 acceptable tolerance levels, 0.3, 0.2, 0.1 and 0.05, to study the relation among suspicion probability, tolerance error and sample size as shown in Figure 5.5. Given e, let's denote the minimal $\hat{n}$ as $\hat{n}_m$ and the $\hat{p}$ that minimizes $\hat{n}$ as $\hat{p}_m$. With e equaling 0.3, 0.2, 0.1 and 0.05, we get $(\hat{p}_m, \hat{n}_m)$ respectively as $(0.47, 11)$,

$(0.27, 19)$, $(0.12, 42)$ and $(0.06, 86)$. These points specify a path demanding the least sample size to step from a larger tolerance error to a smaller one, i.e. from 0.3 to 0.05. At 95% confidence, we estimate p as below:

$$\begin{cases} p \in [0.17, 0.77] & \text{when } 11 \leq n < 19, \\ p \in [0.07, 0.47] & \text{when } 19 \leq n < 42, \\ p \in [0.02, 0.22] & \text{when } 42 \leq n < 86, \\ p \in [0.01, 0.11] & \text{when } n \geq 86. \end{cases}$$

It coincides with our intuition that a smaller $\hat{p}$ with a smaller e must be justified by a larger n.

However, the model is discrete and very likely such $\hat{p}_m$ does not exist. We thus need to find the sub-optimal $\hat{p}$, denoted as $\hat{p}_{m'}$, around $\hat{p} = \hat{p}_m$, which ensures a sub-minimum $\hat{n}$, denoted as $n_{m'}$. With $t_1 = max\{X\}$, where $F_n(X) < \hat{p}_m$, and $t_2 = min\{X\}$, where $F_n(X) \geq \hat{p}_m$, we have

$$\hat{n}_{m'} = min\{f_{max}(F_n(t_1)), f_{max}(F_n(t_2))\},$$

upon which $\hat{p}_{m'}$ is known. With e equal to 0.3, 0.2, 0.1, 0.05, we can respectively obtain $(\hat{p}_{m'}, \hat{n}_{m'})$ as $(\hat{p}_{m',0.3}, \hat{n}_{m',0.3})$, $(\hat{p}_{m',0.2}, \hat{n}_{m',0.2})$, $(\hat{p}_{m',0.1}, \hat{n}_{m',0.1})$, $(\hat{p}_{m',0.05}, \hat{n}_{m',0.05})$, where $\hat{n}_{m',0.3} < \hat{n}_{m',0.2} < \hat{n}_{m',0.1} < \hat{n}_{m',0.05}$. Therefore, we would have a $\hat{p}$ with a known maximal

error at 95% confidence for each level of sample size:

$$\begin{cases} \hat{p}_{m',0.3} \in [p-0.3, p+0.3] & \text{when } n \in [\hat{n}_{m',0.3}, \hat{n}_{m',0.2}), \\ \hat{p}_{m',0.2} \in [p-0.2, p+0.2] & \text{when } n \in [\hat{n}_{m',0.2}, \hat{n}_{m',0.1}), \\ \hat{p}_{m',0.1} \in [p-0.1, p+0.1] & \text{when } n \in [\hat{n}_{m',0.1}, \hat{n}_{m',0.05}), \\ \hat{p}_{m',0.05} \in [p-0.05, p+0.05] & \text{when } n \geq \hat{n}_{m',0.05}. \end{cases}$$

Robust model. The value of $(\hat{p}_{m'}, \hat{n}_{m'})$ is continuously updated as the sample size increases. At each sample size level, a suspicion is defined by the obtained credible $\hat{p}_{m'}$. Because of the maximum error e, we might underestimate p ($\hat{p}_{m'} < p$) and undermine the hang detection accuracy. To avoid this, we make $q = \hat{p}_{m'} + e$. Because the confidence of $\hat{p}_{m'} \in [p-e, p+e]$ is 95%, we claim $q \geq p$ with 97.5% confidence.

Before the hang detection is performed, ParaStack needs to accumulate at least $\hat{n}_{m',0.03}$ random samples to build a model. The model building time is thus $\hat{n}_{m',0.03} \cdot I$. Since different applications may have different appropriate values of I that assure randomness, the model building time also varies from one application to another.

5.2.3 Lightweight Design Details

One monitor per node can be launched to examine the runtime state of all processes on a local node. But checking the call stack of all processes to sample S_{out} can slowdown the target application's execution. Hence, following lightweight strategy is introduced.

***CROUT_MPI* significance**. We monitor only a constant number, say C, of processes instead of all. Accordingly, we define *CROUT_MPI Significance*, denoted as

S_{crout}, as the fraction of processes at OUT_MPI in a Randomly selected C processes. The hang detection scheme is still valid by checking S_{crout} as the idea of looking for a rare event remains unchanged.

Since only C processes need to be checked and some of them might coexist on the same node, they *at most* occupy C compute nodes, each of which requires one monitor actively checking the selected processes. We thus say monitors on these nodes are *active* and the others are *idle*. This design makes our tool extremely lightweight because: (1) only C processes' states need to be checked at a time cost of several microseconds per check; (2) communication is only required in a very limited scope of no more than C *active* monitors; and (3) the already trivial time cost can be possibly overlapped by target applications' idle time.

Parameter Setting. (1) The lightweight design requires an appropriate setting of C and I. A larger C leads to more overhead, and a smaller initial value of I also does so though it will be enlarged at runtime. In addition, a small value for C like 2 or 3 can flatten the variation of S_{crout} and thus diminishes the flexibility of adjusting $\hat{p}$ that ensures model robustness. Therefore, we set C to 10 first and then find I with initial value of 400 milliseconds to satisfy the above requirements. (2) We perform the *runs test* every 16 samples until randomness is ensured as that is large enough for runs test and small enough to ensure ParaStack has the smallest sample size required to check hangs. (3) We set $\alpha = 0.1\%$, which is statistically highly significant and implies 99.9% confidence. Note this is the only parameter that is tailored by the users.

Prevention of a corner case failure. Rarely a corner case arises due to the

dynamic adjustment of what defines a suspicion to ensure model's robustness. When a suspicion is defined as $S_{crout} = 0$ and one faulty process, whose state is OUT_MPI after a hang happens, is one of the C processes being monitored, i.e. $S_{crout} \neq 0$, neither suspicions nor hangs will be observed. This corner case failure can be avoided by monitoring two *disjoint* random process sets, since the faulty process cannot be present in both sets. Of course more sets are required to be resilient for the case containing multiple faulty processes. *ParaStack* alternates between the two sets using each for a fixed number of observations. Since ideally $q \leq 0.77$ and $log_{0.77}0.001 = 26.5$, the maximal times of suspicions required to verify a hang is 27. ParaStack alternates between two sets every 30 times to ensure it has enough time to find hangs while monitoring the process set with $S_{crout} = 0$ before switching to the other one with $S_{crout} \neq 0$.

Transient slowdowns at large scale. As noted in recent works, on large scale systems, the system noise can sometimes lead to a substantial transient slowdown of an application [73, 105]. We also occasionally encountered transient slowdowns on Tianhe-2 – typically in less than 4 runs out of a total of 50 runs. It is important not to confuse a transient slowdown with a hang. We observe that a transient slowdown is distinguishable from a hang because, unlike a hang, it is characterized by the presence of a few processes stepping through the code slowly. This transient-slowdown-specific effect can be identified if any of the following is true: (1) at least one process passes through different MPI functions; (2) at least one process steps in and out of MPI functions other than MPI_Iprobe, MPI_Test, MPI_Testany MPI_Testsome and MPI_Testall, i.e. a process running in a busy-waiting loop stepping across non-MPI code and a function like MPI_Test is treated as staying in the MPI

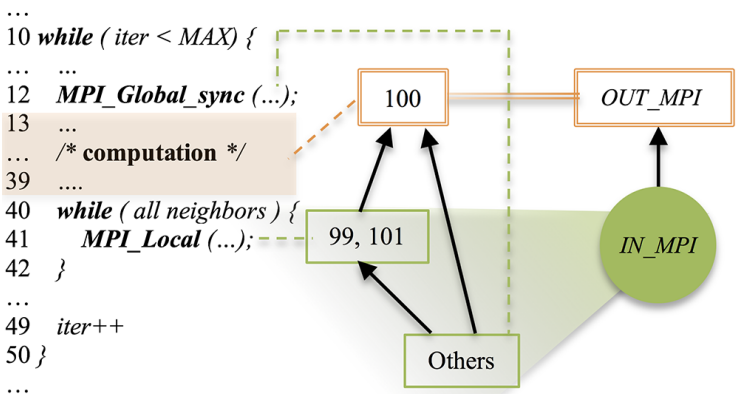

Figure 5.6: Faulty process identification for *computation-error induced hangs*. On the left is an MPI program skeleton, which hangs due to a computation error in process 100. Traditionally, the faulty process can be detected based on the progress dependency graph as shown in the middle. Our technique greatly simplifies the idea by just checking runtime states as shown on the right.

function. Thus we check if such slowdown-specific effect exists based on two stack traces of each target process upon a hang report from the model-based mechanism. If it exists, we report a transient slowdown rather than a hang and resume monitoring; otherwise, we report a hang to users.

5.3 Identifying Faulty Process

Once a hang is detected, *ParaStack* reports the processes in *OUT_MPI* as faulty. The reported processes are claimed to contain the root cause of a hang that results from a computation error. If no process is reported, we claim the hang is a result of a communication error. Next we focus on locating the faulty process for a computation error induced hang.

On the left in Figure 5.6 we show the solver code skeleton of a typical MPI program that is expected to run at large scale. In addition to the computation, all pro-

cesses perform both local communication and synchronization-like global communication. Synchronization-like global communication stands for a communication type that works like a synchronization across all processes such that no process can finish before all enter into the function call. For example, *MPI_Allgather* falls into this category, but *MPI_Gather* does not. In an erroneous run, process 100 fails to make progress due to a computation error, so its immediate neighbors, processes 99 and 101, wait for it at the local communication, which in turns causes all the others to hang at the global communication. Process 100 thus should be blamed as the *faulty process* for this hang. Locating this faulty process would take programmers a giant step closer to the root cause considering hundreds and thousands of suspicious processes are eliminated. Traditional progress-dependency-analysis methods [85, 84, 98] are very effective in aiding the identification of faulty process for general hangs but they involves complexities like recording control-flow information and progress comparison as shown in the middle of Figure 5.6. But for *computation-error induced hangs* these complexities are not necessary. ParaStack instead is inherently simple for this case.

Identification. Across *all* processes, we identify processes in state *OUT_MPI* as the faulty ones for a computation-error induced hang since all the other concurrent ones in *IN_MPI* would wait for the faulty processes. This can be achieved by simply glancing at the state of each process. As shown on the right in Figure 5.6, process 100 is easily located as it is the only one staying in state *OUT_MPI*.

Busy waiting loop based non-blocking communication. If a hang occurs in an application with busy waiting loops, in addition to persistently finding faulty processes in *OUT_MPI*, we may also occasionally find a few non-faulty in *OUT_MPI*. For example,

HPL has its own implementation of collective communication based on busy waiting loops, which can make a few non-faulty processes step back and forth in a track trace rooted at such HPL communication functions when a hang appears. This can mislead ParaStack into believing that such non-faulty processes are faulty. To avoid this, we check every process's state several times and then select the ones that are in OUT_MPI persistently.

5.4 Implementation

ParaStack is implemented in C and conforms to the MPI-1 standard, by which we can ensure the maximum stability by only using a few old, yet good, widely-tested MPI functions while avoiding newly-proposed more error-prone functions. It was tested on Linux/Unix systems and integrated with popular batch job schedulers *Slurm* and *Torque*. It is easily usable by MPI applications using mainstream MPI libraries like MVAPICH, MPICH, and OpenMPI.

Job submission. We provide batch job submission command for Slurm and Torque. It processes users' allocation request, and executes the application and ParaStack concurrently. It ensures only one monitor per node is launched.

Mapping between MPI rank and process ID. ParaStack finds all the processes belonging to the target job by its command name using the common Linux/Unix command *ps*. Users submit a job by specifying the number of nodes and processes per node. Under this setting, the MPI rank assignment mechanism implies two rules: (1) MPI rank increases as process id increases on the same node; and (2) MPI rank increases as node id, in a ordered node list, increases. Suppose the number of target processes per node is ppn and a monitor's id is i. The MPI processes from rank $i * ppn$ to $(i+1) * ppn - 1$ shares

the same node with Monitor i that does the local mapping by simply sorting process ids.

Hang detection. (1) ParaStack suspends and resumes a processes' execution using *ptrace*, and resolves the call-chain using *libunwind*. (2) To obtain the runtime state, we examine stack frames to check if they *start* with *'mpi'*, *'MPI'*, *'pmpi'*, or *'PMPI'* until the backtrace finishes or such relation is found. If found, the state is *IN_MPI*; otherwise it is *OUT_MPI*. This works as mainstream MPI libraries use the above naming rule and users rarely use function names starting with such strings. (3) *Idle monitors* wait for messages in a busy waiting loop consisting of a hundreds-of-milliseconds-sleep and a nonblocking test to avoid preemption.

5.5 Discussion

We discuss handling of complex situations by ParaStack.

Multi-threaded MPI program. A hybrid parallel program, using MPI+OpenMP or MPI+Pthreads, can have both thread-level and process-level parallelism. (1) For thread level MPI_THREAD_SINGLE and MPI_THREAD_FUNNE- LED, only the master thread communicates. Thus, ParaStack works by simply monitoring the master thread. (2) For more progressive mode, MPI_THREAD_SERIALIZED and MPI_THREAD_MULTIPLE, ParaStack must be adapted by redefining the runtime state of a process as: if at least one thread from a process is in MPI communication, we say this process is in *IN_MPI*; otherwise, it is *OUT_MPI*. Hence, a hang can still be captured by the fact that too few *processes* are in *OUT_MPI* persistently.

Applications with multiple phases. An application may alternate among

Scale	256	1024	4096	8192	16384
BT, CG LU, SP	*D*	*E*		—	—
FT	*D,E*	*E*	—	—	—
MG	*E*	—	—	—	—
HPL	$8*10^4$	$2*10^5$	$2.5*10^5$	$3*10^5$	$3.5*10^5$
HPCG	*64 * 64 * 64*		—	—	—

Table 5.2: Default input sizes used by each application at various running scales. Inputs *D* and *E* are the two largest inputs that come with the benchmarks. The input size for HPL specifies the width of a square matrix and the input size for HPCG specifies the local domain dimension.

several phases with differing behaviors leading to imprecision in the S_{crout} model. However, ParaStack can be easily adapted by constructing separate models for different phases if the application is instrumented to inform ParaStack of phase changes during execution. ParaStack can build separate models by sampling each of the phases and using them for respective phases.

Applications with load imbalance. ParaStack is developed to detect hangs for applications with good load balance and is not suitable for applications with severe load imbalance. For applications with *severe load imbalance*, near the end of execution, a few heavy-workload processes may be running. Thus our model based mechanism can fail. We ignore this situation because applications with severe load imbalance should not be deployed at large scale so as to avoid computing resources waste. For moderate load imbalance, we can apply the technique of detecting transient slowdowns as the load imbalance is also characterized by the effect that a few processes are still running slowly.

5.6 Experimental Evaluation

Computing platforms. We evaluate ParaStack on three platforms: *Tardis*, *Tianhe-2* and *Stampede*. Tardis is a 16-node cluster, with each node having 2 AMD Opteron 6272 processors (with *32 cores* in all) and 64GB memory. Tianhe-2 is the *2nd* fastest supercomputer in the world [31], with each node having 2 E5-2692 processors (with *24 cores* in all) and 64GB memory. Stampede is the *12th* fastest supercomputer in the world [31], with each node having 2 Xeon E5-2680 processors (*16 cores* in all) and 32GB memory. Infiniband is used for all. We allocate respectively 8 nodes—256 (8*32) processes—on Tardis, 64 nodes—1,024 (64*16) processes—on Tianhe-2, and up to 1024 nodes—16,384 (1024*16) processes—on Stampede.

Applications and input sizes. We use six NAS Parallel Benchmarks (NPB: BT, CG, FT, MG, LU and SP) [17], High Performance Linpack (HPL) [9], and High Performance Conjugate Gradient Benchmark (HPCG) [53] for evaluation. The execution of these widely used benchmarks consists of a trivial setup phase and a time-consuming iterative solver phase. Though HPCG has multiple phases, all phases are iterative. As ParaStack is developed to monitor long-running runs, we use large available input sizes indicated in Table 5.2 by default unless otherwise specified. In our evaluation, we ignore MG due to its short execution time on both Stampede and Tianhe-2, and ignore FT on Stampede as it crashes at large scale due to memory limit. We did not inject errors in HPCG on Stampede and Tianhe-2 as it has multiple iterative steps and our random error injection technique is not readily applicable.

Fault injection. On Tardis, to simulate a hang, we suspend the execution of a

Time interval	10 ms	100 ms
O_t	50.88s	7.52s
n	18220	1870

Table 5.3: For an execution of HPL on a 15000*15000 matrix, the clean run on average takes 185.05 seconds. O_t is the total stack trace overhead due to n stack trace operations.

randomly selected process by injecting a long sleep in a random invocation of a random user function as faults are more likely to be in the application than in well-known libraries [98]. We use *gprof* to collect all the *user functions*. *Dyninst* [24] is used to statically inject errors, i.e. long enough sleep calls, in application binaries. We discard the cases where error appears in the first 20 seconds of execution because real-world HPC applications spend the majority of time in the later solver phase and building our model takes around 20 seconds. Note our tool targets hangs in the middle of long runs such as those reported in [2, 3] and the model building time is trivial in comparison to the program execution time. On Stampede and Tianhe-2 we inject errors in the source code and simulate a hang by injecting a long sleep call in a randomly selected iteration of a randomly selected process.

5.6.1 Hang Detection Evaluation

I. Overhead. To begin with, we measured the overhead of stack trace for a single process running HPL, a highly compute intensive application. We executed 5 clean runs and 5 runs with stack trace using time intervals of 10ms and 100ms. The average cumulative total overhead (O_t) and number of stack trace operations (n) are given in Table 5.3. As we can see, the overhead is high for interval of 10ms – a 50.88 seconds increase over clean run that takes 185.05 seconds. However, for the interval of 100ms the overhead is low – 7.52 seconds. Thus, for I of 100ms or higher we can expect our tool to have very low overhead.

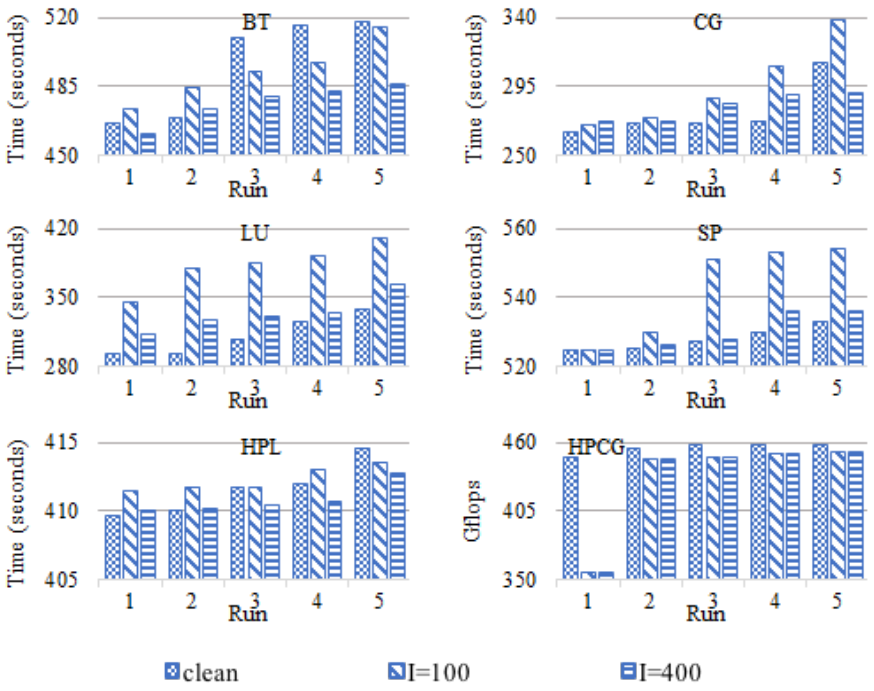

Figure 5.7: Performance comparison of running applications with ParaStack (I = 100 ms), with ParaStack (I = 400 ms) and without ParaStack (clean) on Stampede at scale 1024 based on 5 runs in each setting. The performance is evaluated as *GFLOPS* for HPCG and as *time cost in seconds* for all the others, and the the 5 runs are ordered by performance.

Now we study the impact of using *ParaStack* on runtimes for all applications under two I settings of 100ms and 400ms at scales of 256 and 1024 processes. Note I does not change in this study – we disable the automatic adjustment of I. Experiment results are based on 5 runs at each setting. Table 5.4 shows results at scale 256 and it shows that ParaStack has negligible impact on applications' performance in either setting. At scale 1024, we separately present the performance for each of the 5 runs in each setting on Stampede and Tianhe-2 as the performance variations due to system noise are greater than the prior experiment. On Stampede, Figure 5.7 shows the performance for I = 400ms is often better than that with I = 100ms, and is almost the same as that of clean runs (except for LU). Since Tianhe-2 suffers less system noise due to its lower utilization rate

Benchmark	BT		CG		FT		LU		MG		SP		HPL		HPCG	
Metric	P	S	P	S	P	S	P	S	P	S	P	S	P	S	P	S
clean	336.7	1.0	132.0	1.1	178.8	0.3	247.8	2.9	347.3	0.5	511.1	0.3	277.8	0.8	29.1	0.1
I=100	336.4	0.6	131.6	0.2	179.5	0.2	247.8	0.6	347.0	0.5	510.3	0.4	277.7	0.5	29.1	0.1
I=400	336.8	1.4	132.4	0.6	179.07	0.7	246.6	0.6	347.1	0.3	511.0	0.6	277.2	0.4	29.1	0.1

Table 5.4: Performance comparison of running applications with ParaStack ($I = 100ms$), with ParaStack ($I = 400ms$) and without ParaStack (clean) on Tardis at scale 256. *P*erformance is measured by the delivered *GFLOPS* for HPCG and by the *time cost in seconds* for the others, and *S*tandard deviation of the performance is shown.

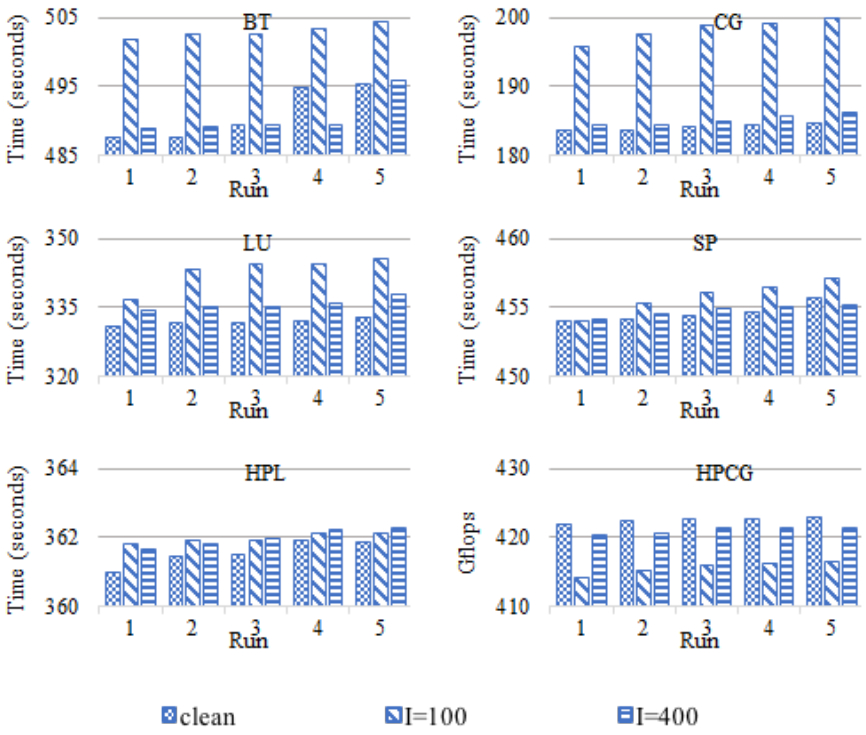

Figure 5.8: Performance comparison of running applications with ParaStack ($I = 100$ ms), with ParaStack ($I = 400$ ms) and without ParaStack (clean) on Tianhe-2 at scale 1024.

than Stampede, the performance variation on it is less. Hence we can expect Tianhe-2 better captures ParaStack's overhead. On Tianhe-2, Figure 5.8 clearly shows that $I = 400$ms is always better than $I = 100$ms, and introduces a slight overhead compared with the clean runs. Table 5.5 shows the overhead for each application. The overhead with $I = 400$ms is at most 1.14%, which is always better than the overhead in the other setting. Hence for the rest of the experiments we use the setting of $I = 400$ms.

Benchmark	BT	CG	LU	SP	HPL	HPCG
I=100	2.44%	7.61%	3.35%	0.26%	0.12%	1.64%
I=400	-0.08%	0.55%	1.14%	0.04%	0.12%	0.35%

Table 5.5: ParaStack's Overhead on Tianhe-2 at scale 1024 based on the average of 5 runs.

Platform	Tardis		Tianhe-2		Stampede	
# runs	100		50		20	
Scale	*256*		*1024*		*1024*	
Metric	Time(s)	AC_h	Time(s)	AC_h	Time(s)	AC_h
BT	336	99%	487	100%	495	100%
CG	132	100%	177	100%	278	100%
FT	179	98%	100	100%	—	—
LU	247	98%	328	98%	311	100%
MG	347	100%	—	—	—	—
SP	511	100%	454	100%	528	100%
HPCG	—	100%	—	—	—	—
HPL	277	99%	362	100%	411	100%

Table 5.6: Accuracy of hang detection. The rough time cost of a correct run is shown.

II. False positives were evaluated using 100 correct runs of each application at scale 256 on Tardis taking about 66 hours, 50 correct runs for BT, CG, FT, LU, SP, HPCG, and HPL at scale 1024 on Tianhe-2 taking about 27.9 hours, and 20 correct runs for BT, CG, LU, SP, HPCG, and HPL at scale 1024 on Stampede taking about 11.8 hours. *The false positive rate was observed to be 0% when the theoretical false positive rate is* $\alpha = 0.1\%$. In addition, no false positives were observed even in all erroneous runs performed in experiments presented next.

III. Accuracy refers to the effectiveness of *ParaStack* in detecting hangs in erroneous runs. Let the total number of faulty runs be T and the total number of times that the hang can be detected correctly be T_h. The accuracy is defined as T_h/T. Table 5.6 shows the accuracy based on 100 erroneous runs at scale 256 on Tardis, 50 erroneous runs at scale 1024 on Tianhe-2 and 20 runs at scale 1024 on Stampede. *ParaStack* misses only 6 times out of 800 runs at scale 256. In these cases, hangs happen very early (even before *ParaStack* has collected enough samples to build an accurate model) and thus I is continuously enlarged

Scale↓	Metric↓	BT	CG	FT	LU	SP	HPL
1024	D	7.2	18.8	8.8	9.0	4.8	6.8
	S	7.3	14.7	7.3	4.2	2.2	3.3

Table 5.7: Response delay on Tianhe-2: D is the average response delay in seconds; S is the standard deviation.

Scale↓	BT		CG		LU		SP		HPL	
	D	S	D	S	D	S	D	S	D	S
1024	7.1	4.5	7.6	4.5	7.8	5.9	4.1	1.2	5.0	2.5
4096	5.4	3.6	24.1	13.1	4.3	1.3	3.7	2.0	5.6	4.7

Table 5.8: Response delay on Stampede: D is the average response delay in seconds and S is the standard deviation.

and the probability of $S_{crout} = 0$ is increased. Hence there is not enough time to verify the hang before the allocated time slot expires. One hang in LU is also missed at scale 1024 on Tianhe-2; for all other runs at scale 1024, the accuracy is 100% on Stampede and Tianhe-2.

Due to the high cost, a limited number of experiments was conducted. At *scale 4096*, we studied ParaStack's accuracy based on 10 erroneous runs for BT, CG, LU, SP, and HPL. For BT, LU, and HPL, $AC_h = 1$; for CG and SP, AC_h equals 0.8 and 0.9 respectively. Also, as the later two take less time, errors are more likely to happen earlier. At *scale 8192*, the accuracy based on 5 erroneous runs of HPL is $AC_h = 5/5$. At *scale 16384*, the accuracy based on 3 erroneous runs of HPL is $AC_h = 3/3$.

IV. Response delay is the elapsed time from a hang's occurrence to its detection by *ParaStack*. For the erroneous runs where hangs are correctly identified by *ParaStack*, we collected the response delays for all applications. Figure 5.9 shows the response delay distribution for 100 erroneous runs at scale of 256 on Tardis. Table 5.7 shows the average response delay and the standard deviation based on 50 erroneous runs at scale of 1024 on

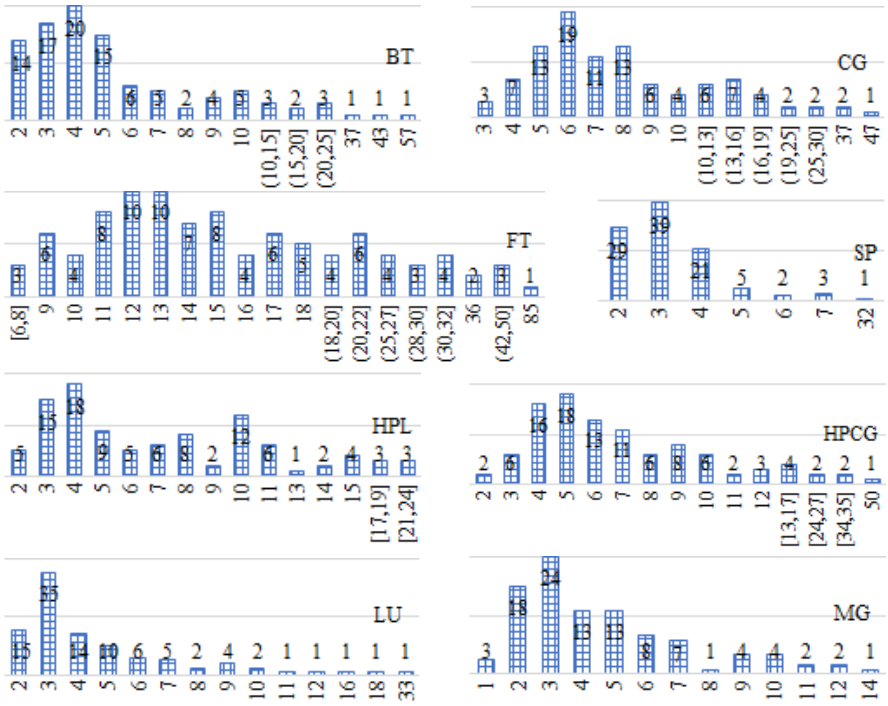

Figure 5.9: The response delay of hang detection based upon 100 erroneous runs for each application at scale of 256 on Tardis. *The horizontal axis represents response delay in seconds and the vertical axis represents the number of times ParaStack identifies a hang with the corresponding delay.*

Tianhe-2. Table 5.8 shows the average response delay and the standard deviation based on 20 erroneous runs at scale of 1024 and 10 erroneous runs at scale of 4,096 on Stampede. At *scale 8,192* the response delay for 5 erroneous runs of HPL are 5, 6, 14, 16, and 17 seconds. At *scale 16,384* response delays for 3 erroneous runs are 6, 7, and 10 seconds. *ParaStack* commonly detects a hang with a delay of no more than 1 minute at various scales. We observe that the response delay not only varies across applications, it also differs from one hang to another for a given application. As we know, the response delay in worst case is $I \cdot \lceil log_q \alpha \rceil$. The variation of q depending upon sample size and the adaptation of I leads to the variation in response delay.

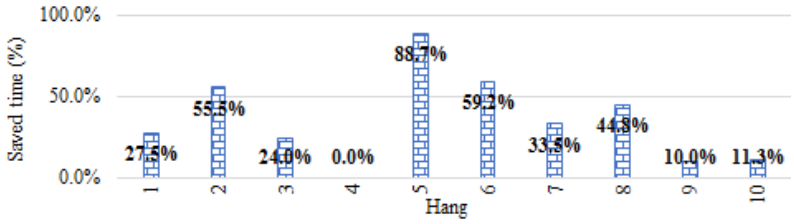

Figure 5.10: The percentage of time savings ParaStack brings to application users in batch mode based on 10 erroneous runs of HPL with the average percentage equal to 35.5%.

V. ParaStack enabled time savings for application users. Supercomputers typically charge users in *Service Units* (*SUs*) [19, 26]. The total number of *SUs* charged for a job is equal to the product of the number of nodes occupied, the number of cores per node, and the elapsed wallclock time of the job. Application users run their job assuming absence of hangs; thus, when running an application in batch mode, the allocated time will be wasted if a hang arises and the user is charged for it. *ParaStack* saves this cost by terminating the application upon a hang. To quantify the time saving, we ran HPL 10 times using a problem size 100,000 with a (uniform) random error injected in the iterative phase. The correct run takes around 518 seconds, so users are inclined to request a larger time slot – conservatively let us assume a 10-minute time slot is requested. The percentage of time ParaStack saves is shown in Figure 5.10. For the 10 runs, on average the time saved is 35.5%. With increasing number of tests, the average time saved will approach 50%. Because ParaStack detects a hang soon after its occurrence, if hang is expected to happen randomly during execution, the average time at which the program is terminated is about half of the execution time.

VI. ParaStack vs. timeout. Unlike timeout method, ParaStack can report hang according to the user specified confidence which automatically adjust parameters like

Platform	Bench.	P			P*		
		AC	FP	D	AC	FP	D
Tianhe-2	FT(D)	1.0	0.0	4.8	1.0	0.0	3.5
	FT(E)	1.0	0.0	29.4	1.0	0.0	14.9
Tardis	FT(D)	1.0	0.0	14.0	0.9	0.0	25.2
	LU(D)	1.0	0.0	4.5	1.0	0.0	1.1
	SP(D)	1.0	0.0	3.3	1.0	0.0	1.0

Table 5.9: ParaStack's generality for variation of platforms, benchmarks and input sizes at scale 256 based on 10 erroneous runs per configuration. Notes: (1) P stands for the default ParaStack with I being initialized as 400ms; $P*$ stands for ParaStack with I being initialized as 10ms. (2) AC, accuracy; FP, false positive rate; D, average response delay.

sampling interval, what defines a suspicion, how many times of suspicions confirm a hang. Our experimental results in Table 5.1 already demonstrated the drawbacks of timeout-based mechanism. In contrast, ParaStack's default configuration shows 100% accuracy and 0% false positive rate (see Table 5.9). Even though we initialize I with a very small value that does not deliver random sampling – say $I = 10$ in comparison to the default value of $I = 400$, ParaStack's effectiveness ($P*$) still compares well with the default (P) as shown in Table 5.9. This is because ParaStack has capability of adapting I automatically so as to ensure random sampling. In short, the key advantage of *ParaStack* is that it reports hang based on runtime history with high confidence $1 - \alpha$ while traditional timeout method is based upon guesses as shown in Table 5.1.

5.6.2 Faulty Process Identification

We evaluate the the effectiveness of faulty process identification using two metrics: *faulty process identification accuracy* (AC_f); and *faulty process identification precision* (PR_f). As the faulty process identification is only performed after a hang is detected, this evaluation is based on the same experiment as conducted in the hang detection accuracy

Platform	Tardis		Tianhe-2		Stampede	
Scale	*256*		*1024*		*1024*	
Metric	AC_f	PR_f	AC_f	PR_f	AC_f	PR_f
BT	$^{99}/_{99}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
CG	$^{100}/_{100}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
FT	$^{97}/_{98}$	0.99	$^{50}/_{50}$	1.0	—	—
LU	$^{98}/_{98}$	1.0	$^{49}/_{49}$	1.0	$^{20}/_{20}$	1.0
MG	$^{100}/_{100}$	1.0	—	—	—	—
SP	$^{100}/_{100}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
HPCG	$^{100}/_{100}$	1.0	—	—	—	—
HPL	$^{99}/_{99}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0

Table 5.10: Evaluation of faulty process identification.

evaluation. Recall, T_h denotes the total number of times that the hang is detected correctly. Let the number of times that the faulty process is found be T_f out of T_h times, and let x_i be the number of processes reported as faulty ones in the *i-th* run. For the *i-th* run, if the true faulty process is in this report, we say its precision (p_i) is $p_i = 1/x_i$ in this single run; otherwise, $p_i = 0$. The 2 metrics are defined as $AC_f = T_f/T_h$ and $PR_f = \frac{1}{T_h}\sum_{i=1}^{T_h} p_i$.

Table 5.10 gives results based on 100 erroneous runs at scale of 256 on Tardis, 50 erroneous runs at scale of 1024 on Tianhe-2, and 20 erroneous runs at scale 1024 on Stampede. In terms of accuracy, *ParaStack* misses the faulty process once at scale 256. Because this is a rare occurrence, we can handle it by printing debugging information for further analysis. The precision of faulty process identification for FT is approximately 99.0% as *ParaStack* misses the faulty process once out of 98 runs. The precision for all other applications is 100%.

At *scale 4096*, ParaStack's effectiveness based on 10 erroneous runs for BT, CG, LU, and SP is $AC_f = 1.0, \text{and } PR_f = 100\%$; for HPL, $AC_f = PR_f = 0.9$. At *scale 8192*, *ParaStack*'s effectiveness based on 5 erroneous runs of HPL is $AC_f = 5/5, \text{and } PR_f =$

86.7% as in one run ParaStack identifies 3 processes as faulty which includes the real faulty process while it precisely identifies the real faulty process the other 4 runs. At *scale 16384*, ParaStack's effectiveness based on 3 erroneous runs of HPL is $AC_f = 3/3$, and $PR_f = 100\%$.

5.7 Summary

By observing S_{crout}, ParaStack detects hangs with high accuracy, in a timely manner, with negligible overhead and in a scalable way. Based on the concept of *runtime state*, it sheds light on the roadmap for further debugging. It does not require any complex setup and supports mainstream job schedulers – *Slurm* and *Torque*. Its compliance with MPI standard ensures its portability to various hardware and software environments.

Chapter 6

Related Work

6.1 General Bug Detection

Various tools have been developed to aid the detection of general software bugs in MPI programs. We classify these tools into three categories: (1) *testing* that detects bugs via iteratively executing the program and aims to satisfy some criteria of code coverage like branch coverage, statement coverage, and non-determinism coverage due to message interleaving, (2) *static methods* that detect MPI errors by analyzing the code without the need of executing it, and (3) *dynamic methods* that check the correctness of an MPI program by executing it.

Testing. The study of *testing* in the filed of HPC is scarce. Some notable works in this less-studied area include: message perturbation to improve the testing coverage of non-determinism [128], Automated Testing System (ATS) for regression testing [5], Fortran-TestGenerator for generating unit tests for legacy HPC applications written in Fortran [75],

and GKLEE for concolic testing for GPU programs [89, 93]. However, none performs code coverage based testing for MPI applications — code coverage based testing can manifest software bugs via achieving a high coverage and thus improves the software quality. COMPI hence fills the gap and performs branch coverage based testing for MPI applications.

Research works on concolic testing [116, 65] are most closely related to our COMPI tool. Concolic testing, also known as dynamic symbolic execution, automatically generates test inputs via combining symbolic execution dynamically with concrete execution. Since its birth, concolic testing has been a great success to test a variety of *sequential* programs [36, 107, 43, 55, 103, 80]. Also concolic testing has been applied to shared-memory parallel programs including GPU programs [89, 93, 94, 48], multi-threading programs in C and Java [114, 115, 56]. None of the above tackle distributed-memory applications. jCUTE [112] instead applies concolic testing to boost the branch coverage as well as to detect deadlock in distributed Java programs. While jCUTE does not tackle MPI programs that are commonly used in HPC area, MPISE [61] and MPI-SV [131] deals with the non-determinism of message passing in MPI programs and mainly focus on communication deadlock detection based on concolic testing, and they are both built on top of CLOUD9 [41] — a parallel version of concolic testing tool KLEE [43]. Complimentary to MPISE and MPI-SV, COMPI [91], built on top of concolic testing tool CREST [42], performs *branch coverage* based testing *efficiently* for *real-world MPI applications* and aims to detect runtime errors unrelated to non-determinism.

Static methods. Existing static methods [117, 119, 132, 118, 54] mainly focus on the detection of illegal use of MPI functions and communication deadlocks. Among these, quite

a few methods including MPI-Spin [118], TASS [119], and CIVL [132], make use of symbolic execution. These works are complimentary to our approach as they employ symbolic execution alone as opposed to concolic testing. Concolic testing distinguishes itself from traditional symbolic execution via simplification of symbolic constraints by using concrete values. The cost of concolic testing is the sacrifice of completeness as some constraints can be lost or not recorded precisely due to the use of concrete values. What distinguishes our COMPI tool from above works is that it focuses on efficiently performing branch coverage based testing of real-world MPI applications.

Dynamic methods. Most existing bug detections tools for HPC programs are dynamic methods. DAMPI [127], Intel Message Checker [51], Intel Trace Analyzer and Collector [102], ISP [125], Marmot [83], MUST [72, 71], and Umpire [126] detects deadlocks and incorrect MPI usage based on runtime information collected by intercepting MPI library. Besides, dynamic approaches are also used to detect defects inside MPI library such as data movement anomaly and synchronization error [44, 63, 46] as well as non-deterministic message race [109]. Unlike above, our COMPI tool does not aim to uncover a specific class of bug; instead, it aims to improve the code quality via achieving high branch coverage and can detect software bugs that lead to abnormal execution termination as well as program hangs in the process of achieving a high coverage. Hermes [79] detect communication deadlocks using dynamic symbolic verification that is very similar to concolic testing. COMPI differs from it in that the goal of COMPI is to improve code quality of real-world MPI programs using branch-coverage based testing.

6.2 Tackling Scaling Bugs

To aid the development of MPI applications, developers have to deal with the challenge of scaling bugs. Such challenge involves three steps: (1) how to *manifest* a scaling bug, (2) how to *diagnose* it to find the root cause, and (3) how to *fix* it.

Manifestation. To manifest scaling bugs, many research relies on model-based prediction [133, 134, 88]. Zhou et al. [133, 134] predict the happening of scaling problems based on a model built from bug-free runs at small scale. Laguna and Schulz [88] predict the scale-dependent integer overflow bugs at code-level in large-scale parallel applications based upon a model of small scale runs that monitoring integer operations related to the number of processes and the input size. Techniques in [97, 39, 87] debug large-scale applications by deriving their normal timing behavior and looking for deviations from it. Unlike these, we propose the feasibility of testing to manifest scaling bugs with the use of MPI collectives and justify that testing is not necessarily expensive by using only a small number of processes at the expense of a large message size.

Diagnosing. It is very common that MPI application developers still use use traditional debugging tools such as GDB [8] to diagnose scaling bugs. Following this line, DDT [1] and TotalView [32] are fully functional debuggers that allow developers interact with many processes instead of just one. However, interacting with a huge number of processes usually overwhelm users. Many lightweight diagnosing tools [34, 85, 84, 98], though only applicable to hangs and performance slowdowns, have been developed to aid the identification of root causes. Our work tackling scaling bugs with the use of MPI collectives is complimentary to

these as it aims to find an integrated solution that detects and bypasses scaling problems of the MPI libraries without the need of root cause diagnosis.

Fixing. Fixing a scaling bug is challenging due to the aggregated complexity of technical difficulty, history reason (MPI backward compatibility), and responsibility issue (who should fix the bug). Hammond et al. [68] extends MPI to support the need of sending a message having a large element count that exceeds INT_MAX based on building big data types. Our work used the idea of big data types as one of our approaches to solve a different problem: (1) the element count does not disobey the MPI standard, i.e. it is smaller than INT_MAX; and (2) we provide non-intrusive workarounds for more than integer overflow, e.g. the workarounds can also work for environment-dependent scaling bugs.

6.3 Techniques for Handling a Program Hang

At various running scales, a variety of tools exist for detecting as well as diagnosing communication deadlocks — a special case of hang — and hangs.

Handling hangs at small scale. At small scale, existing work focus on detecting communication-deadlock using methods like time-out [83, 64, 101], communication dependency analysis [126, 69], and formal verification [125]. These tools either use an imprecise timeout mechanism or precise but centralized technique that limits scalability. MUST [72, 71] claims to be a scalable tool for detecting MPI deadlocks at large scale, but its overhead is still non-trivial as it ultimately checks MPI semantics across all processes. These non-timeout methods do not address the full scope of hang detection, as

they do not consider computation-error induced hangs. Compared with those, our work, ParaStack, deals with both kinds of hangs. Besides, the advantage of ParaStack over these non-timeout tools is that it incurs *negligible overhead* to detect hangs; the advantage of non-timeout tools is they are precise and potentially gives detailed insights to remove the errors. Also, ParaStack is better than time-out methods as it avoids the difficulty of parameter setting.

Handling hangs at large scale. At large scale, a few recent efforts focus on detecting and diagnosing hangs. STAT [35] divides tasks (process/thread) into behavioral equivalence classes using call stack traces. It is very useful for further analysis once *ParaStack* identifies a hang, especially when hangs are hard to reproduce. STAT-TO [34] extends STAT by providing temporal ordering among processes that can be used to identify the least-progressed processes; however, it requires expensive static analysis that fails in the absence of loop-ordered-variables. AutomaDed [85, 84] draws probabilistic inference on progress dependency among processes to nominate the least-progressed task, but it fails to handle loops. *Prodometer* [98] performs highly accurate progress analysis in the presence of loops and can precisely pinpoint the faulty processes of a hang. Prodometer's primary goal is to *diagnose* cause of hangs by giving useful progress dependency information. In contrast, *ParaStack*'s main aim is to *detect* hangs with high confidence in production runs. Thus Prodometer's capabilities are complementary to our tool.

Chapter 7

Conclusions and Future Work

7.1 Contributions

This dissertation contributes various dynamic bug detection techniques pre- and post-deployment for both developers and users of MPI applications. For developers, we provide an automated testing tool to aid the bug detection in their software development. For users, we provide a testing suite helping users test suspicious MPI collectives and an avoidance framework helping avoid detected scaling bugs without requiring any user effort; also we devise a hang detection tool helping users save a great amount of computing resources in presence of a hang at large scale production runs. Specifically, we highlight following contributions.

Concolic Testing for MPI Applications

We develop COMPI, the first concolic testing framework for MPI applications. First, we provide an automated testing framework for MPI programs — COMPI 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, we make COMPI a very practical testing tool via three techniques: input capping, two-way instrumentation, and constraint set reduction.

In addition, we improve the usability of COMPI via addressing two issues: input values generated by COMPI do not deliver cost-effective testing, and COMPI does not support floating-point arithmetic. We address the first issue via proposing a novel input tuning technique. To address the second issue, we enable handling of *floating point* data types and operations and demonstrate that the efficiency of constraint solving can be improved if we rely on the use of reals instead of floating point values.

Tackling Scaling Bugs

The complexity of MPI collectives is directly impacted by both parallelism scale and problem size, and their use often triggers scaling problems. Fixing a scaling problem is challenging, and thus it usually takes much time for users to wait for an official fix, which sometimes is even not possible due to the difficulty of bug reproduction, root-cause identification, and fix development. To improve users' productivity, we first establish the necessity of user side testing and justifies that testing is feasible at moderate scale computing platforms. Next we provide a protection layer to avoid scaling bugs non-intrusively — once the protection layer detects a condition that triggers a scaling problem it avoids the bug by either (1) chopping the communication into smaller ones or (2) building big data types. Our work hence provides an immediate remedy when an official fix is not readily available.

Hang Detection at Large Scale

While program hangs on large parallel systems can be detected via the widely used timeout mechanism, it is difficult for the users to set the timeout. To address the above problems with *hang detection*, this thesis presents *ParaStack*, an extremely lightweight tool to detect hangs based on runtime history in a timely manner with high accuracy, negligible overhead with great scalability, and without requiring the user to select a timeout value. For a detected hang, it further sheds light on the roadmap for further debugging: (1) it tells whether such hang is caused by a computation error or not, and (2) it reports the root-cause process for a hang induced by a computation error. ParaStack is integrated into two parallel job schedulers Torque and Slurm and we validated its performance on two world-leading supercomputers.

7.2 Future Work

Verification of MPI Programs

Our current work, COMPI, cannot avoid limitations. These limitations shed light on the roadmap of our future work. First, more efficient search strategy that guides the exploration of execution tree is in need. BoundedDFS is a very traditional search strategy, and is adopted by COMPI only because it is the only strategy that can make the testing pass the sanity check of MPI programs. It is known there exist an array of more efficient strategies for the concolic testing of sequential programs. We believe combining BoundedDFS with other more efficient strategies would lead to a more efficient strategy that works for MPI programs. Second, supporting other mainstream programming languages in the field of

HPC like C++ and Fortran is important to enable the verification of more real-world MPI programs. Third, enabling symbolic execution across different processes is necessary. This is because the values of variables marked as symbolic in one process could also determine if some branches could be explored or not in other processes, considering the values of the marked variables could be passed across processes. Fourth, a better constraint construction method is in need. COMPI simplifies the constraints via replacing symbolic expressions with concrete values. The simplification cause imprecise constraints (e.g., multiplication of two symbolic expressions like $x * y$ can be simplified as $2x$ supposing the concrete value of y is 2), or even loss of constraints (e.g., division operations is not even recorded symbolically). The imprecision and loss of constraints impair the symbolic reasoning and thus can cause some branches not to be covered. Fifth, lightweight instrumentation is possible using taint analysis. COMPI relies on instrumentation such that symbolic execution code can be inserted, but the instrumentation performed at code level is too heavy. With taint analysis, we can only instrument instructions related to the use of variables being marked symbolically, which would avoid a significant portion of instrumentation and thus enable faster execution of programs in testing.

Also, advances in areas outside of HPC also inspire potential research directions. First, combining concolic testing with random testing [96, 67] has proved to be more effective to manifest defects in sequential programs: random testing can quickly reach execution states deep in the execution tree, and symbolic execution can thoroughly explore the neighborhood of the states. We would also like to study the effectiveness of such combination for MPI applications. Second, concolic testing can also be used to check specific type of

software bugs such as integer overflow. Via inserting conditional statements that checks if integer overflow occurs among integer operations, the symbolic reasoning can easily check if any inputs can lead to integer overflow based on the constraints issued by the inserted conditional statements.

Tackling Scaling Bugs

One way to fix a scaling bug is to eradicate the bug from its root. Statistics has proved its importance to aid the diagnosing of a scaling bug [39, 87, 85, 84, 90], i.e., extracting the root cause from the massive amount of debugging information at large scale runs. Future work could explore the applicability of statistics and machine learning techniques on scaling bugs of various kinds other than hangs.

The other way is to propose techniques that mitigate the bugs negative impact when the first solution is not possible. In this way, hard-to-fix scaling bugs can be fixed indirectly. Future work could explore such indirect solution's application scenarios other than MPI collectives.

Beyond MPI

It has become the mainstream for supercomputers to be infused with the power of both traditional CPUs and co-processors. To make full use of their combined power, hybrid parallel programming models have been proposed, e.g., MPI + CUDA and MPI + OpenMP. The correctness support for HPC applications demands the progress on not only each programming model but also their combinations. In the future, we hope to go beyond MPI to provide correctness support for other popular programming interfaces as well as their combinations.

Bibliography

[1] Allinea ddt. URL: http://www.allinea.com/products/ddt.

[2] Bug occurs after 12 hours. URL: https://github.com/open-mpi/ompi/issues/81/.

[3] Bug occurs after 200 iterations. URL: https://github.com/open-mpi/ompi/issues/99.

[4] Can we count on mpi to handle large datasets? URL: http://blogs.cisco.com/performance/can-we-count-on-mpi-to-handle-large-datasets.

[5] Catching bugs with the automated testing system. URL: https://computation.llnl.gov/research/mission-support/WCI/automated-testing-system.

[6] Code hangs when variables value increases. URL: https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/601182.

[7] Floating point exception in susy-hmc. URL: https://github.com/daschaich/susy/issues/16.

[8] Gdb: The gnu project debugger. URL: http://www.gnu.org/software/gdb.

[9] Hpl: a portable implementation of the high-performance linpack benchmark for distributed-memory computers. URL: http://www.netlib.org/benchmark/hpl/.

[10] Intel mpi benchmarks user guide. URL: https://software.intel.com/en-us/imb-user-guide.

[11] Io-watchdog. URL: https://code.google.com/p/io-watchdog/.

[12] Mpi: A message-passing interface standard version 3.1. URL: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[13] Mpi code hangs when send/recv large data. URL: https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/610561.

[14] Mpi has bad performance in user mode. URL: https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/607259.

[15] Mpi_bcast in intel mpi library 2017 hangs on large user-defined datatypes. URL: https://software.intel.com/en-us/articles/intel-mpi-library-2017-known-issue-mpi-bcast-hang-on-large-user-defined-datatypes.

[16] Mpi_get_count and large messages. URL: http://trac.mpich.org/projects/mpich/ticket/1005.

[17] Nas parallel benchmarks. URL: https://www.nas.nasa.gov/publications/npb.html.

[18] Ocaml. URL: https://ocaml.org/.

[19] Ohio supercomputer center's charging policy. URL: https://www.osc.edu/supercomputing/software/general#charging.

[20] Ompi 1.4.3 hangs in imb gather. URL: https://github.com/open-mpi/ompi/issues/125.

[21] Ompi-1.7 mpi_alltoallv hangs. URL: https://github.com/open-mpi/ompi/issues/1620.

[22] Ompi v1.6 running out of registered memory. URL: https://svn.open-mpi.org/trac/ompi/ticket/3134.

[23] Osu micro-benchmarks. URL: http://mvapich.cse.ohio-state.edu/benchmarks.

[24] Paradyn project: Dyninst. URL: http://www.paradyn.org/html/manuals.html#dyninst.

[25] Probability theory and mathematical statistics: normal approximation to binomial. URL: https://onlinecourses.science.psu.edu/stat414/node/179.

[26] San diego supercomputer center's charging policy. URL: http://www.sdsc.edu/support/user_guides/comet.html#charging.

[27] Sloccount. URL: https://www.dwheeler.com/sloccount/.

[28] Solve the 1d time dependent heat equation using mpi. URL: http://people.sc.fsu.edu/~jburkardt/c_src/heat_mpi/heat_mpi.html.

[29] Stampede user guide [online]. URL: https://portal.tacc.utexas.edu/user-guides/stampede.

[30] Three segmentation fault bugs in susy-hmc. URL: https://github.com/daschaich/susy/issues/15.

[31] Top500 list. URL: http://www.top500.org/lists/2014/11/.

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

[33] The yices smt solver. URL: http://yices.csl.sri.com/.

[34] 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 *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–11, Nov 2009.

[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 *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[36] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.

[37] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=998675.999437.

[38] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983. doi:10.1147/sj.223.0229.

[39] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong Ahn, and Martin Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 231–240, July 2010.

[40] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[41] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198. ACM, 2011.

[42] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, Sept 2008. doi:10.1109/ASE.2008.69.

[43] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[44] Zhezhe Chen, J. Dinan, Zhen Tang, P. Balaji, Hua Zhong, Jun Wei, Tao Huang, and Feng Qin. Mc-checker: Detecting memory consistency errors in mpi one-sided applications. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 499–510, New Orleans, LA, Nov 2014. IEEE Computer Society.

[45] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

[46] Zhezhe Chen, Xinyu Li, Jau-Yuan Chen, Hua Zhong, and Feng Qin. Syncchecker: Detecting synchronization errors between mpi applications and libraries. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 342–353, May 2012.

[47] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/351240.351266`, `doi:10.1145/351240.351266`.

[48] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the sixth conference on Computer systems*, pages 315–328. ACM, 2011.

[49] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[50] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[51] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of mpi programs with intel® message checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[52] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, Aug 2012.

[53] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[54] Alexander Droste, Michael Kuhn, and Thomas Ludwig. Mpi-checker: static analysis for mpi. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 3. ACM, 2015.

[55] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162. ACM, 2007.

[56] Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 353–365. ACM, 2014.

[57] Justin E Forrester and Barton P Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68. Seattle, 2000.

[58] G. C. Fox and S. W. Otto. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[59] Xianjin Fu, Zhenbang Chen, Hengbiao Yu, Chun Huang, Wei Dong, and Ji Wang. Symbolic execution of mpi programs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 809–810. IEEE Press, 2015.

[60] Xianjin Fu, Zhenbang Chen, Hengbiao Yu, Chun Huang, Wei Dong, and Ji Wang. Symbolic execution of mpi programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 809–810, Piscataway, NJ, USA, 2015. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2819009.2819178`.

[61] Xianjin Fu, Zhenbang Chen, Yufeng Zhang, Chun Huang, Wei Dong, and Ji Wang. Mpise: Symbolic execution of mpi programs. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 181–188. IEEE, 2015.

[62] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[63] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, New York, NY, USA, 2007. ACM.

[64] James Coyle Jim Hoekstra Glenn R. Luecke, Yan Zou and Marina Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[65] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1065010.1065036`, `doi:10.1145/1065010.1065036`.

[66] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012. URL: `http://doi.acm.org/10.1145/2090147.2094081`, `doi:10.1145/2090147.2094081`.

[67] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[68] Jeff R Hammond, Andreas Schäfer, and Rob Latham. To int_max... and beyond!: exploring large-count support in mpi. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 1–8. IEEE Press, 2014.

[69] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, Jan 2006.

[70] Dustin Heaton and Jeffrey C Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207–219, 2015.

[71] Tobias Hilbrich, Bronis R. de Supinski, Wolfgang E. Nagel, Joachim Protze, Christel Baier, and Matthias S. Müller. Distributed wait state tracking for runtime mpi deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. ACM.

[72] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for mpi deadlock detection. In *Proceedings of the 23rd International Conference on Supercomputing*, number 10, pages 296–305, New York, NY, USA, 2009. ACM.

[73] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[74] C. Hovy and J. Kunkel. Towards automatic and flexible unit test generation for legacy hpc code. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, pages 1–8, Nov 2016. `doi:10.1109/SE-HPCCSE.2016.005`.

[75] Christian Hovy and Julian M Kunkel. Towards automatic and flexible unit test generation for legacy hpc code. In *SE-HPCCSE@ SC*, pages 1–8, 2016.

[76] Arne Johanson and Wilhelm Hasselbring. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering*, 2018.

[77] Upulee Kanewala and James M Bieman. Testing scientific software: A systematic literature review. *Information and software technology*, 56(10):1219–1232, 2014.

[78] Diane Kelly and Rebecca Sanders. The challenge of testing scientific software. In *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)*, pages 30–36. Citeseer, 2008.

[79] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. Dynamic symbolic verification of mpi programs. In *International Symposium on Formal Methods*, pages 466–484. Springer, 2018.

[80] Yunho Kim, Moonzoo Kim, Young Joo Kim, and Yoonkyu Jang. Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1143–1152. IEEE, 2012.

[81] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. Automated unit testing of large industrial embedded software using concolic testing. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 519–528. IEEE, 2013.

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

[83] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. Marmot: An mpi analysis and checking tool. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 493–500, 2003.

[84] Ignacio Laguna, Dong Ahn, Bronis de Supinski, Saurabh Bagchi, and Todd Gamblin. Diagnosis of performance faults in large scale mpi applications via probabilistic progress-dependence inference. 2014.

[85] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 213–222, New York, NY, USA, 2012. ACM.

[86] Ignacio Laguna, Dong H Ahn, Bronis R de Supinski, Todd Gamblin, Gregory L Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, et al. Debugging high-performance computing applications at massive scales. *Communications of the ACM*, 58(9):72–81, 2015.

[87] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. Large scale debugging of parallel tasks with automaded. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 50:1–50:10. ACM, 2011.

[88] Ignacio Laguna and Martin Schulz. Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In *Proceedings of the International Conference*

for High Performance Computing, Networking, Storage and Analysis, page 19. IEEE Press, 2016.

[89] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. Gklee: concolic verification and test generation for gpus. In *ACM SIGPLAN Notices*, volume 47, pages 215–224. ACM, 2012.

[90] Hongbo Li, Zizhong Chen, and Rajiv Gupta. Parastack: Efficient hang detection for mpi programs at large scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 63:1–63:12, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3126908.3126938`, `doi:10.1145/3126908.3126938`.

[91] Hongbo Li, Zizhong Chen, and Rajiv Gupta. Compi: Concolic testing for mpi applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 865–874. IEEE, 2018.

[92] Hongbo Li, Zizhong Chen, Rajiv Gupta, and Min Xie. Non-intrusively avoiding scaling problems in and out of mpi collectives. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 415–424. IEEE, 2018.

[93] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in cuda programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[94] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical symbolic race checking of gpu programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 179–190. IEEE Press, 2014.

[95] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 601–612. IEEE, 2017.

[96] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 416–426. IEEE, 2007.

[97] Alexander V Mirgorodskiy, Naoya Maruyama, and Barton P Miller. Problem diagnosis in large-scale computing environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 88. ACM, 2006.

[98] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 193–203. ACM, 2014.

[99] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[100] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 902–912. IEEE, 2015.

[101] P. Ohly and W. Krotz-Vogel. Automated mpi correctness checking: What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, 2007.

[102] Patrick Ohly and Werner Krotz-Vogel. Automated mpi correctness checking: What if there was a magic option. In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, pages 19–25, 2007.

[103] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 73–82. IEEE Computer Society, 2011.

[104] Zhengbin Pang, Min Xie, Jun Zhang, Yi Zheng, Guibin Wang, Dezun Dong, and Guang Suo. The th express high performance interconnect networks. *Frontiers of Computer Science*, 8(3):357–366, 2014.

[105] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55. IEEE, 2003.

[106] Prakash Prabhu, Hanjun Kim, Taewook Oh, Thomas B Jablin, Nick P Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, David Walker, Yun Zhang, et al. A survey of the practice of computational science. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[107] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 186–196. ACM, 2010.

[108] Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Martin Schulz. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 62:1–62:12, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2807591.2807642`, `doi:10.1145/2807591.2807642`.

[109] Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, Martin Schulz, and Christopher M. Chambreau. Noise injection techniques to expose subtle and unintended message races. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 89–101, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3018743.3018767`, `doi:10.1145/3018743.3018767`.

[110] David Schaich and Thomas DeGrand. Parallel software for lattice n= 4 supersymmetric yang–mills theory. *Computer Physics Communications*, 190:200–212, 2015.

[111] David Schaich and Thomas DeGrand. Parallel software for lattice n= 4 supersymmetric yang–mills theory. *Computer Physics Communications*, 190:200–212, 2015.

[112] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.

[113] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.

[114] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[115] Koushik Sen and Gul A Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical report, 2006.

[116] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1081706.1081750`, `doi:10.1145/1081706.1081750`.

[117] Stephen F Siegel. Model checking nonblocking mpi programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–58. Springer, 2007.

[118] Stephen F Siegel. Verifying parallel programs with mpi-spin. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 13–14. Springer, 2007.

[119] Stephen F Siegel and Timothy K Zirkel. Tass: The toolkit for accurate scientific software. *Mathematics in Computer Science*, 5(4):395–426, 2011.

[120] Anthony Skjellum, Nathan E Doss, and Kishore Viswanathan. Inter-communicator extensions to mpi in the mpix (mpi extension) library. Technical report, Citeseer, 1994.

[121] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[122] Xiang Song, Haibo Chen, and Binyu Zang. Why software hangs and what can be done with it. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, June 2010.

[123] Frieda S Swed and Churchill Eisenhart. Tables for testing randomness of grouping in a sequence of alternatives. *The Annals of Mathematical Statistics*, 14(1):66–87, 1943.

[124] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1792786.1792798`.

[125] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. Isp: A tool for model checking mpi programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, New York, NY, USA, 2008. ACM.

[126] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[127] A. Vo, G. Gopalakrishnan, R.M. Kirby, B.R. de Supinski, M. Schulz, and G. Bronevetsky. Large scale verification of mpi programs using lamport clocks with lazy update. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE Computer Society, 2011.

[128] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis De Supinski, and Andreas Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 27–36. ACM, 2006.

[129] X. Wu, Z. Xu, D. Yan, T. Wu, J. Yan, and J. Zhang. The floating-point extension of symbolic execution engine for bug detection. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 265–272, Dec 2016.

[130] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey Voelker. Mpiwiz: Subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 251–260, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1504176.1504213`, `doi:10.1145/1504176.1504213`.

[131] Hengbiao Yu. Combining symbolic execution and model checking to verify mpi programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 527–530. ACM, 2018.

[132] Manchun Zheng, Michael S Rogers, Ziqing Luo, Matthew B Dwyer, and Stephen F Siegel. Civl: formal verification of parallel programs. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 830–835. IEEE, 2015.

[133] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 85–96. ACM, 2011.

[134] Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. Wukong: automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 131–142. ACM, 2013.