System Support for Improving the Reliability of MPI Applications and Libraries

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Zhezhe Chen, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Dissertation Committee:

Prof. Feng Qin, Advisor
Prof. Gagan Agrawal
Prof. Christopher Stewart
© Copyright by

Zhezhe Chen

2013
Abstract

Although the Message Passing Interface (MPI) is widely used to develop parallel programs on computing systems such as clusters, MPI applications and libraries suffer from bugs. A recent survey of MPI users indicated that programmers had problems with different types of MPI errors such as non-deterministic errors and distributed operation ordering errors. Furthermore, the underlying MPI library implementations also suffer from software bugs. For example, more than 500 bug tickets are created for various versions of Open MPI each year. Similarly, about 350 bug tickets are reported for MPICH2 each year. These MPI bugs can severely affect the availability and productivity of MPI programs.

This dissertation proposes runtime support to improve the reliability of MPI applications and libraries. Our approach is to build software tools using system methods including instrumentation-based profiling and runtime analysis to help detect software bugs in MPI applications and libraries. First, this dissertation proposes a method called FlowChecker to detect communication errors in MPI libraries. The main idea of FlowChecker is to extract program intentions of message passing and to check whether these intentions are fulfilled by the underlying MPI libraries, i.e., whether messages are delivered from the specified sources to the specified destinations. Second, this dissertation proposes another approach called SyncChecker to detect synchronization errors between MPI applications and libraries. SyncChecker tracks relevant memory
accesses in the MPI applications and corresponding message send/receive operations in the MPI libraries. Then it checks whether the correct execution order between the MPI application and the MPI library is enforced by the MPI completion check routines. Finally, this dissertation proposes an approach called MC-Checker to detect memory consistency errors in MPI applications with one-sided communication. MC-Checker first performs online instrumentation and logs relevant runtime events such as MPI one-sided calls and local load/store operations to the trace files. Then MC-Checker performs offline analysis and checks the operations against compatibility tables to see whether there are memory consistency errors.

We have designed and implemented software prototypes for the proposed methods and evaluated them with real-world and injected bugs on popular MPI libraries, including MPICH2, MVAPICH2, and Open MPI, and different MPI applications such as Athena and octopus. The experimental results show that the approaches proposed in this dissertation can effectively detect and locate bugs in MPI applications and libraries. They also provide useful diagnostic information to help programmers improve reliability of MPI applications and libraries. Furthermore, our tools incur low or moderate overhead which indicates that our tools are applicable for production runs or software testing phases. The results also demonstrate that the runtime support in these methods is very effective for improving the reliability of MPI applications and libraries.
Dedicated To

My Love, Lingdi Zhao
Acknowledgments

I am very happy and lucky to have Professor Feng Qin as my advisor. He is bright, smart, generous, inspiring, diligent, and open-minded. He is always the one who can give me guidance and help when I need. In addition, he takes good care of and respects his students. He always puts students’ interests in the first place. Furthermore, he is very nice and considerate. I really had a very good time during my five year PhD studies with him as my advisor. Those wonderful experiences will always live in my mind.

I am also greatly indebted to Professor Gagan Agrawal, Professor Christopher Stewart, and Professor Jesse Kwiek, not only for their time and effort in serving on my dissertation committee but also for their generous help during my PhD studies. I also thank other professors who gave me kind help and/or passed their knowledge to me. An incomplete list of them includes Professor Xiaodong Zhang, Professor P. Sadayappan, Professor Radu Teodorescu, Professor Yuan F. Zheng, Professor Tamal Dey, Professor Dong Xuan, Professor Srinivasan Parthasarathy, Professor Neelam Soundarajan, and Professor Atanas Rountev.

I also thank my groupmates, Qi Gao, Wenbin Zhang, Mai Zheng, Xinyu Li, Jau-Yuan Chen, and Dachuan Huang, who gave me help in my research projects. I also truly thank Lina Fu and Xuejun Wen, who picked me up in the airport when I first arrived in Columbus and took me for shopping every week before I owned a car. They
gave me much convenience and help. I would like to thank my language partners Douglas Trimble and Jasmine Sims who helped me improve my spoken English. Douglas also helped me improve my driving skills so that I could pass the driving test and build a good and safe driving habit. Thanks also go to my roommate Zhichun Zhang. We lived together and helped each other in the past four years. I also thank Ania Mitros, Seth LaForge, Piotr Mitros, and Stefanie Tellex for giving me help during my applying for graduate schools and inviting me to their homes for visit. I would also like to thank Adam Champion, Dacong Yan, Yang Zhang, Nan Deng, Boxuan Gu, Jin Teng, Xiaojia Zhao, Bin Ren, Xin Wang, Yuji Wang, Hui Zhang, Yilin Ma, and Honglei Zhu for their help in various aspects.

I am grateful to Dr. Pin Zhou and Scott Guthridge in IBM Research for being my mentors when I was doing internships. It was a great experience working with them and other members in their teams in those summers. It really helps me broaden my knowledge scope and makes me learn a lot from them.

Finally, I would like to thank my parents, Xiangcun Chen and Youzhi Chen, who nurture and cultivate me and give me a good environment during my growing up. They also teach me how to be a good person. They always support me and believe me in every possible way. I also thank my sister Yuanyuan Chen, who grows up with me and gives me and our family a lot of support. Thanks also go to my brother Xuexue Chen who grows up and plays with me together. My last special thank will go to my darling wife Lingdi Zhao, also named as Lefei Zhao. She is kind, smart, hardworking and easy going. She tolerates all of my shortcomings. She always gives me support during my difficulties. My life becomes more colorful and interesting with her.
Vita

2006 ........................................ Bachelor of Science
Zhejiang University
Hangzhou, China

2008 ........................................ Master of Science
Zhejiang University
Hangzhou, China

2012 ........................................ Master of Science
The Ohio State University
Columbus, OH, USA

2008 - present ............................. Graduate Student
The Ohio State University
Columbus, OH, USA

Publications

Research Publications


**Fields of Study**

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Our Approach</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Detecting Bugs in MPI libraries via Message Flow Checking</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Detecting Synchronization Errors between MPI Applications and Libraries</td>
<td>6</td>
</tr>
<tr>
<td>1.2.3 Detecting Memory Consistency Errors in MPI One-sided Applications</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>11</td>
</tr>
<tr>
<td>2. Related Work</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Bug detection for parallel and distributed programs</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Problem diagnosis in large systems</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Synchronization bug detection</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Communication error detection</td>
<td>14</td>
</tr>
<tr>
<td>2.5 General software bug detection</td>
<td>14</td>
</tr>
</tbody>
</table>
4.6.2 Case Studies ........................................... 74
4.6.3 Runtime Overhead .................................... 78
4.6.4 Effects of the Optimizations ............................ 79
4.7 Summary .................................................. 81

5. Bug Detection in MPI One-sided Application ................. 83
   5.1 Main Idea of MC-Checker ............................... 85
       5.1.1 Memory Consistency Errors ...................... 85
       5.1.2 Bug Detection Process ......................... 88
   5.2 Design and Implementation ............................ 89
       5.2.1 Design Overview and Challenges ................. 89
       5.2.2 ST-Analyzer: Identifying Relevant Memory Accesses 92
       5.2.3 Profiler: Collecting Runtime Information ...... 93
       5.2.4 DN-Analyzer: Analyzing Traces and Detecting Memory Consistency Errors 94
   5.3 Evaluation Methodology ................................ 106
   5.4 Experimental Results .................................. 107
       5.4.1 Overall Effectiveness ............................ 107
       5.4.2 Case Studies ..................................... 109
       5.4.3 Runtime Overhead ............................... 112
   5.5 Summary ................................................ 114

6. Final Remarks ............................................ 116

Bibliography .................................................. 118
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 MPI libraries and bug cases used in the evaluation. Lib means Library, LOC means Lines of Code, and Desc. means Description.</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Overall results of FlowChecker for bug detection.</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Disk space overhead of FlowChecker. T-Size means trace size, E-Time means execution time, and G-Rate means growth rate of the trace. Trace size reported here is per process.</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Evaluated applications and synchronization errors. Note that Athena-r1086 means the MPI application Athena with the revision number 1086. “r” has the same meaning in the names of other applications.</td>
<td>70</td>
</tr>
<tr>
<td>4.2 Overall effectiveness of SyncChecker for bug detection</td>
<td>72</td>
</tr>
<tr>
<td>5.1 The operation compatibility table within an epoch. The operations in the first row will be executed before the operations in the first column. BOTH means both overlapping and non-overlapping memory operations are permitted. NOVL means only non-overlapping operations are permitted, i.e., operations that access overlapping buffers conflict with each other.</td>
<td>96</td>
</tr>
<tr>
<td>5.2 Operation compatibility table across processes. BOTH means both overlapping and non-overlapping operations are permitted. NOVL means only non-overlapping operations are permitted. X means combining these operations is erroneous.</td>
<td>103</td>
</tr>
<tr>
<td>5.3 Evaluated applications. Note that some bug IDs use the date of the applications and some use svn revision or version number.</td>
<td>106</td>
</tr>
<tr>
<td>5.4 Overall effectiveness of MC-Checker</td>
<td>107</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Three types of data corruption bugs in MPI programs</td>
<td>2</td>
</tr>
<tr>
<td>3.1 An example to illustrate the main idea of FlowChecker. The MP-intention here is {A1→D1, A2→D2} and the message flows are A1→B1→C1→D1 and A2→B2→C2→D2.</td>
<td>18</td>
</tr>
<tr>
<td>3.2 A simplified bug case from an MPI library</td>
<td>19</td>
</tr>
<tr>
<td>3.3 FlowChecker design overview</td>
<td>21</td>
</tr>
<tr>
<td>3.4 An example of an MP-intention. The MP-intention is implied by a group of\texttt{MPI_Gathers} with non-contiguous data in two processes. MP-Extractor represents the MP-intention as two MPI call-pairs, each with two transmission units.</td>
<td>27</td>
</tr>
<tr>
<td>3.5 An example of data source propagation for two transmission units. A, B, C, and D are four data movement operations. Buffers 1, 2, 3, 4, 6, 7, 9, and 10 store message data specified by the MPI application, while buffers 5 and 8 store control data used by the underlying MPI library.</td>
<td>30</td>
</tr>
<tr>
<td>3.6 Correctness Checking by MF-Checker</td>
<td>32</td>
</tr>
<tr>
<td>3.7 Bug case 1: Datatype construction error in Open MPI (But ID: #209)</td>
<td>40</td>
</tr>
<tr>
<td>3.8 Bug case 2: Incorrect memory copy direction in MPICH2 (Bug ID: #280)</td>
<td>42</td>
</tr>
<tr>
<td>3.9 Bug case 3: Incorrect Sending Buffer Size in MVAPICH2 (Bug ID: 03/2007)</td>
<td>44</td>
</tr>
</tbody>
</table>
3.10 Bug report for the bug case in Open MPI (Bug ID #689) . . . . . . 45
3.11 Runtime overhead of FlowChecker. Baseline is the native execution
without applying FlowChecker. . . . . . . . . . . . . . . . . . . . . . 46
3.12 Scalability study of FlowChecker. Baseline is the native execution
without applying FlowChecker. . . . . . . . . . . . . . . . . . . . . . 47
4.1 Six scenarios of the execution orders among the runtime events (at
both MPI application and library levels) in nonblocking communication. 54
4.2 Design overview of SyncChecker . . . . . . . . . . . . . . . . . . . . 56
4.3 The state machine for detecting synchronization errors . . . . . . . 61
4.4 Bug case: Sending buffer overwritten in Athena (written in C) . . . . 75
4.5 Bug case 2: Sending buffer overwritten in octopus, which is written in
Fortran. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.6 Runtime overhead of SyncChecker. “Native” means execution without
applying SyncChecker, “SyncChecker-P” means execution with Pro-
filer only, “SyncChecker-PA” means execution with both Profiler and
Analyzer enabled. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.7 Optimization effects of runtime overhead with Profiler enabled only.
The runtime overhead of “> 200” incurred by Profiler-basic is be-
cause we terminate the program execution after that long time. “Na-
tive” means execution without applying SyncChecker, “Profiler-basic”
means execution without any optimization, “Profiler-ER” means exe-
cution with ER optimization enabled, “Profiler-ER-AT” means execu-
tion with ER and AT optimizations enabled, “SyncChecker-P” means
execution with ER, AT and MR optimizations enabled. . . . . . . . . 80
4.8 The numbers of profiled memory accesses per process with different
level of optimizations. For clarity, we put a tiny bar with “0” on
the top indicating no memory access. Also note that y-axis is in a
logarithmic scale. “Profiler-basic”, “Profiler-ER”, “Profiler-ER-AT”,
and “SyncChecker-P” have the same meanings as those in Figure 4.7. 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Examples of memory consistency errors. The bold operations are conflicting operations without happens-before or consistency order.</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Design overview of MC-Checker</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>Memory consistency errors across processes. We abbreviate MPI_Barrier, MPI_Win_lock, MPI_Win_unlock MPI_Put, and MPI_Get as Barrier, lock, unlock, Put, and Get, respectively.</td>
<td>99</td>
</tr>
<tr>
<td>5.4</td>
<td>Bug case 1: Conflicting <strong>MPI_Get</strong> and load operations in BT-broadcast. Note that the load operation of <strong>check</strong> is conflicting starting from the second iteration of while loop.</td>
<td>110</td>
</tr>
<tr>
<td>5.5</td>
<td>Bug case 2: Conflicting operations in lockopts. load/store in section A is concurrent with put in section D.</td>
<td>111</td>
</tr>
<tr>
<td>5.6</td>
<td>Runtime overhead of MC-Checker. “Native” means execution without MC-Checker. “MC-Checker” means execution with MC-Checker’s Profiler.</td>
<td>113</td>
</tr>
<tr>
<td>5.7</td>
<td>Scalability study on benchmark pingpong</td>
<td>114</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

1.1 Motivation

The Message Passing Interface (MPI) [3] is widely used to develop parallel programs on computing systems such as clusters. This is evidenced by a plethora of MPI applications across many disciplines such as astronomy, bioinformatics, weather forecasting, and financial modeling [8]. As clusters continue to be a major component of High Performance Computing (HPC) environments [1], MPI is becoming increasingly prevalent.

Although MPI is very popular for writing parallel programs, MPI applications and libraries suffer from bugs. A recent survey [29] of MPI users indicated that programmers had problems with different types of MPI errors such as non-deterministic errors and distributed operation ordering errors. Furthermore, the underlying MPI library implementations [6, 7, 36, 74] also suffer from software bugs. For example, more than 500 bug tickets are created for various versions of Open MPI [36] each year. Similarly, about 350 bug tickets are reported for MPICH2 [6] each year. These MPI bugs and errors severely affect the availability and productivity of MPI programs. They can lead programs to hang, crash, or generate incorrect results.
However, it is challenging to detect bugs in MPI applications and libraries. First, some of the bugs may be non-deterministic. They may manifest themselves in some program runs but not in others. Also, some of the bugs may depend on the MPI libraries. They may be manifested only with some particular MPI libraries. Second, it is hard to tell where the bugs are even if they manifest. In some cases, we do not know whether the bugs are in the MPI applications or in the MPI libraries.

In particular, we are especially interested in a type of bugs called data corruption. Many data corruption bugs are related to specific MPI semantics and only reside in MPI programs. There are three types of data corruption bugs in MPI programs depending on their origins. Figure 1.1 shows these three types of bugs. The first type of bugs reside in MPI applications and are irrelevant to any MPI calls. We call them non-MPI bugs. The second type of bugs are located in the MPI interface between MPI applications and libraries. We call them MPI-interface bugs. The third type of bugs are located in MPI libraries and are irrelevant to MPI interfaces. We call them MPI-library bugs. The non-MPI bugs have no difference from data corruption...
in other normal programs. We can use tools such as Valgrind [61] to detect them. We will not discuss them in this dissertation.

There are few works for detecting MPI-interface and MPI-library bugs. MPI-interface bugs are more MPI-specific. This type of bugs occur in the interface between MPI applications and libraries. When an MPI application makes MPI calls, the MPI application passes its parameters into the MPI library. When the MPI calls return, the MPI library will send some outputs as the parameters of MPI calls to the MPI application. When an MPI application performs MPI calls, some data corruption may occur in the parameters of MPI calls. There are four scenarios for this type of bugs: 1) the MPI application corrupts data due to MPI application bugs, 2) the MPI application corrupts data due to MPI library bugs, 3) the MPI library corrupts data due to MPI application bugs, and 4) the MPI library corrupts data due to MPI library bugs. MPI-library bugs are data corruptions lying in the MPI library. There are two types of MPI-library bugs. In one type, the corrupted data or buffers are directly related to MPI message delivery. In the other type, data corruption is not related to the buffers for MPI message delivery. This type of bugs can also be treated as normal data corruption bugs. In this dissertation, we will focus on MPI-interface and MPI-library bugs relevant to MPI message delivery.

1.2 Our Approach

In this dissertation, we propose system support to improve the reliability of MPI applications and libraries. The approach is to build software tools by using system methods to help detect and/or manifest software bugs in MPI applications and libraries.
By using system approach, we propose novel methods for bug detection in MPI applications and libraries. Our approaches need to have the following features:

- High effectiveness: The approaches in this dissertation should effectively detect bugs in MPI applications and libraries.

- High efficiency: The approaches in this dissertation for production runs should incur very low overhead so they improve reliability without impacting program runtime. The approach proposed for testing phases should incur moderate overhead, which makes it feasible for testers to find bugs in testing phase.

- Easy to use: The approaches in this dissertation should be automatic approaches for bug detection. In addition, they had better not require source code or modification thereof. Furthermore, they should provide important and useful diagnostic information to help developers and testers detect bugs.

- High portability: The approaches in this dissertation had better be independent of MPI implementations and programming languages. They should be useful for bug detection in different MPI libraries. They should also support bug detection for MPI applications written in different programming languages.

In this dissertation, we propose three approaches to handle reliability problems in MPI applications and libraries. To make sure MPI libraries correctly deliver message data from one process to another in production runs, we propose the first approach called FlowChecker to detect communication bugs by checking message flows in the MPI libraries. This type of bugs is MPI-library bugs relevant to MPI message delivery. This approach can also detect some MPI-interface bugs in the fourth scenario. To improve the reliability of MPI applications in the testing phase, we propose the second
approach called SyncChecker to detect synchronization errors. This type of bugs are
the first scenarios of MPI-interface bugs. To further detect data corruption bugs in
MPI applications with one-sided communication, we propose the third approach called
MC-Checker to detect memory consistency errors by leveraging static and dynamic
approaches. More specific description of these three approaches are shown below.

1.2.1 Detecting Bugs in MPI libraries via Message Flow Checking

To improve the reliability of MPI libraries, this dissertation presents FlowChecker,
a low-overhead method for detecting communication-related bugs. A communication-
related bug in this dissertation refers to a bug that causes all or part of user messages
not to be delivered from the sources to the destinations as specified by the MPI ap-
lications. The main idea of FlowChecker is to extract program intentions of message
passing (MP-intentions) and to check whether the message flows that occur in the
underlying MPI libraries correctly fulfill the MP-intentions. If any MP-intention is
not fulfilled, FlowChecker reports a software bug and provides relevant diagnostic
information. In this dissertation, message flow refers to a series of operations (e.g.,
network send/receive) within MPI libraries that transmit message data from a sending
process to a receiving process.

More specifically, FlowChecker performs online profiling and offline trace analysis
to detect bugs in MPI libraries. To perform online profiling, FlowChecker instruments
the binary code of both MPI applications and MPI libraries. During program exe-
cution, FlowChecker logs MPI function calls (e.g., MPI_Send and MPI_Gather) at the
application level and data movement operations at the library level into trace files.
Data movement refers to the movement of a chunk of contiguous data from a source
buffer to a destination buffer [37]. Examples of data movement include memory copy and network send/receive.

From the MPI calls in the trace files, FlowChecker extracts MP-intentions (e.g., a pair of matched MPI _Send and MPI _Recv function calls made by the MPI application). For each MP-intention, FlowChecker tracks the corresponding message flows by following the relevant data movement operations starting from the sending buffers. If the message data are not correctly delivered to the receiving buffers as indicated by the MP-intention, FlowChecker reports the bug and provides diagnostic information, such as faulty MPI functions or incorrect data movements, to help pinpoint the root causes.

Based on the above ideas, we have implemented a prototype of FlowChecker on Linux. We have evaluated FlowChecker with five real-world and two injected bug cases from three popular MPI libraries, including Open MPI [36], MPICH2 [6], and MVAPICH2 [7].

### 1.2.2 Detecting Synchronization Errors between MPI Applications and Libraries

In this dissertation, we present a new technique called SyncChecker to detect synchronization errors between MPI applications and the underlying MPI libraries. Our main idea is to check, at runtime, whether the reuse of a message buffer in an MPI application is properly enforced to occur after the corresponding message data have been transferred by the MPI library. If not, SyncChecker reports this synchronization error with detailed diagnostic information to help developers understand the root cause and fix the bug. The specific implementation of SyncChecker we report in this dissertation focuses on nonblocking MPI calls such as MPI _I send and MPI _Irecv. We
do not see any particular difficulty in applying the core idea to handle RMA calls since they also follow the semantics of nonblocking communication.

To detect such synchronization errors, SyncChecker performs online profiling on relevant runtime events, then conducts analysis on-the-fly to reason about the event orders. More specifically, SyncChecker first instruments the MPI application for profiling nonblocking MPI calls (e.g., `MPI_Isend`) and relevant memory accesses. In addition, SyncChecker profiles data movement operations in the MPI library to track whether the message data have been transferred. Data movement operations refer to operations that move a chunk of data from a source buffer to a destination buffer, such as memory copy and network send/receive.

After collecting the runtime events, SyncChecker performs the detection process on-the-fly. For each nonblocking call, SyncChecker identifies the message buffer as well as the execution order of the relevant runtime events, including accesses to the buffer and invocations of MPI completion checks by the MPI application, and messages sent or received by the MPI library. SyncChecker reports a synchronization error if the execution order between the buffer accesses by the MPI application and message send/receive events by the underlying MPI library is not enforced by MPI completion checks. Furthermore, SyncChecker provides detailed diagnostic information, including line numbers, function names, and source file names of synchronization error related runtime events mentioned above and their incorrect execution orders.

It is challenging to perform the above tasks. First, SyncChecker needs to handle complex MPI semantics. For example, MPI supports many datatypes, ranging from simple primitive datatypes such as `MPI_INT` to derived datatypes such as the ones specifying non-contiguous memory regions. Additionally, the MPI library often issues
a vast number of data movement operations. Some are related to nonblocking calls whereas others are not. Second, memory access profiling can easily induce prohibitive runtime overhead as indicated by previous software instrumentation tools [61].

Based on the above ideas, we have implemented a prototype of SyncChecker on Linux and evaluated it with seven bug cases including five introduced by the developers of the software and two injected by us. The bugs reside in four different types of MPI applications, including (1) Athena [14], a grid-based application for astrophysical magnetohydrodynamics, (2) octopus [64], a simulator for electron-ion dynamics, (3) Boost-app, an application based on the Boost.MPI library [39], and (4) Sort, a sorting algorithm using MPI. Our experiments have shown that SyncChecker effectively detects all the evaluated bugs and reports detailed diagnostic information.

1.2.3 Detecting Memory Consistency Errors in MPI One-sided Applications

In this dissertation, we present a technique called MC-Checker to detect memory consistency errors in MPI one-sided applications. Memory consistency error occurs when two memory operations, which can be MPI one-sided calls such as `MPI_Put` or local load/store from one or two processes, conflict with each other, leading the programs to an undefined state. To detect memory consistency errors in MPI one-sided applications, the main idea is to efficiently identify the execution epochs and concurrent execution regions, and to check the one-sided operations and/or local memory accesses within an epoch or within a concurrent region across processes. The checking is performed against the compatibility tables, which are constructed based on MPI specification [5]. If a pair of operations conflict, MC-Checker will report the
error and provide diagnostic information to help programmers further locate and fix the bug.

MC-Checker instruments the program before program execution and logs relevant events to trace files at run time. Then it performs offline analysis on the trace files to detect memory consistency errors. Specifically, MC-Checker first instruments relevant runtime events including MPI one-sided routines (e.g. `MPI_Put`), MPI datatype construction routines (e.g. `MPI_Type_struct`), MPI general synchronization routines (e.g. `MPI_Barrier`), and MPI basic support routines (e.g. `MPI_Comm_rank`). In addition, MC-Checker needs to instrument memory load/store accesses because they may conflict with one-sided operations and lead to memory consistency errors.

While tracing provides us program execution information, instrumenting all memory load/store accesses will cause prohibitive runtime overhead. To address this issue, MC-Checker performs static analysis to identify relevant memory accesses so that it only instruments and profiles identified memory load/store accesses. As a result, MC-Checker can achieve lower runtime overhead without losing its correctness checking capability.

Based on the collected runtime trace, MC-Checker will detect memory consistency errors within one process and across processes. It first checks conflicting operations within an epoch in one process. More specifically, it needs to identify the execution region of an epoch. After that, MC-Checker examines the operations of one-sided communication calls that may access local buffers and also checks relevant local memory load/store accesses. If any two operations have buffer overlaps and conflict with each other, MC-Checker will report the bug with diagnostic information.
In addition, MC-Checker checks conflicting operations across processes. In particular, it first needs to match MPI synchronization calls (e.g. `MPI_Barrier`) between processes. Then MC-Checker will identify execution regions in any two processes that could occur concurrently, i.e., the happens-before [46] relation does not hold in the regions. After that, MC-Checker will check the operations in the concurrent regions against the compatibility table to see whether the operations conflict. Once a conflict is found, MC-Checker will report the error.

It is challenging to perform the above tasks due to complicated MPI semantics, especially complicated one-sided semantics. First, MC-Checker needs to handle datatypes ranging from primitive contiguous datatypes such as `MPI_INT` to derived non-contiguous datatypes. Second, MC-Checker needs to process communicator and group management MPI calls to gather knowledge about communication group scopes. Third, MC-Checker needs to handle different types of epoch regions and different communication modes (i.e., active and passive modes). Fourth, matching MPI synchronization calls is also challenging because MPI has many types of synchronization operations. Finally, profiling each memory load/store access is very expensive. MC-Checker needs to take some approach to reduce the profiling overhead. We address these challenges in Section 5.2.

Based on the above idea, we have implemented a prototype of MC-Checker in Linux and evaluated it with three real-world and two injected bugs in five MPI one-sided applications. Our experiments show that MC-Checker can effectively detect memory consistency errors in the evaluated MPI applications.
1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 discusses related work. Chapter 3 discusses communication bugs detection in MPI libraries via message flow checking. Chapter 4 talks about synchronization error detection between MPI applications and libraries. Chapter 5 presents an approach by leveraging static and dynamic methods to detect memory consistency errors in MPI one-sided applications. Chapter 6 concludes this dissertation.
Chapter 2: Related Work

This dissertation related to the previous work in five categories: 1) bug detection for parallel and distributed programs, 2) problem diagnosis in large systems, 3) synchronization bug detection, 4) communication error detection, 5) general software bug detection.

2.1 Bug detection for parallel and distributed programs

Many approaches have been proposed for detecting bugs in parallel and distributed programs by checking program runtime information [29, 34, 37, 41, 45, 52, 77]. In addition to these dynamic approaches, researchers have explored formal verification and model checking methods [72, 78] for detecting bugs such as deadlocks in parallel and distributed programs. Moreover, interactive parallel debuggers [11, 13, 18, 32, 55] leverage automated information collection, aggregation, and visualization to help programmers manually locate root causes of software bugs in parallel programs.

2.2 Problem diagnosis in large systems

Research has been conducted on problem diagnosis for large-scale systems [10, 20, 24, 42, 47, 56, 57, 67, 81, 83]. They mainly focus on identifying root causes of program failures or performance degradation via statistical methods or machine learning
techniques. For example, Falcon locates faulty memory-access interleavings by cleverly identifying the correlation between the interleavings and pass/fail execution. Maruyama and Matsuoka exploit the correlation between the function traces and normal/failed runs for localizing faults [56]. Carrozza et al. leverage Support Vector Machine classifiers to detect and locate software faults in complex safety-critical software systems [22].

2.3 Synchronization bug detection

Much research has been conducted on detecting synchronization bugs (e.g., data races [19,27,51,71,82], atomicity violations [35,49,79], and order violations [38,50,67]) in multi-threaded programs. These approaches can be classified into two categories: dynamic and static approaches. Dynamic approaches [35, 61, 71] typically track all memory accesses at run time and detect ill-synchronized accesses. While these approaches can detect general non-deterministic bugs, they are not suitable for handling the synchronization errors in MPI nonblocking communication. This is mainly because these approaches only focus on low-level memory accesses without understanding complex semantics of MPI programs. For example, without capturing MPI semantics, it is difficult to know when exactly the message data in the buffer has been copied out or sent over the network. Furthermore, these tools incur prohibitive overhead due to fine-grained memory access monitoring [61], although sampling [19] or new hardware [49,50] can alleviate this situation. On the contrary, our approach exploits semantic-relevant dynamic optimizations to significantly reduce runtime overhead.
2.4 Communication error detection

Another closely related method is the checksum feature provided in several MPI libraries [7, 15, 36], which checks the integrity of data in communication to detect network hardware errors. Although checksums may help uncover bugs in the communication layer of MPI libraries, this does not work if an error is introduced before the checksum calculation at the sending side or after checksum matching at the receiving side. Moreover, for a detected bug, the checksum feature only reports a checksum failure for some invocation of an MPI function, providing limited help in finding the root cause. Furthermore, the checksum feature can significantly slow down the communication since it needs to perform computation on the data content, e.g., generating a CRC code [68].

2.5 General software bug detection

There are many research studies on detecting software bugs in general systems. Examples include program assertions [12, 43], static analysis methods [17, 28, 31, 33], dynamic checking approaches [40, 61, 71], model checking [80], and formal verification [60, 75]. Unlike these studies, we solely focus on communication-related bugs in MPI libraries by exploiting program intentions of message passing and message flows. Additionally, researchers have studied parallelizing runtime bug detection on multi-cores [76] and better bug reporting and understanding [23, 73]. These studies can be used to improve our work.
Chapter 3: Bug Detection in MPI Libraries

In this chapter, we present FlowChecker, a low-overhead method for detecting communication-related bugs in MPI libraries. A communication-related bug in this chapter refers to the bug that causes all or part of user messages not to be delivered from the sources to the destinations as specified by the MPI application. The main idea of FlowChecker is to extract program intentions of message passing (MP-intentions) and to check whether the message flows that occur in the underlying MPI libraries correctly fulfill the MP-intentions. If any MP-intention is not fulfilled, FlowChecker reports a software bug and provides relevant diagnostic information. In this chapter, message flow refers to a series of operations (e.g., network send/receive) within MPI libraries that transmit message data from a sending process to a receiving process.

Based on the above ideas, we have implemented a prototype of FlowChecker on Linux. We have evaluated FlowChecker with five real-world and two injected bug cases from three popular MPI libraries, including Open MPI [36], MPICH2 [6], and MVAPICH2 [7]. Unlike previous approaches, FlowChecker has the following advantages:
• To the best of our knowledge, FlowChecker is the first automatic method for detecting communication-related bugs in MPI libraries by checking the correctness of message flows, a key aspect of these libraries. Our experimental results show that FlowChecker can effectively detect all seven evaluate bug cases from three widely-used MPI library implementations and help further pinpoint the root causes of the bugs.

• FlowChecker incurs low runtime and disk space overhead. Our experiments with High Performance Linpack (HPL) [9] and NAS Parallel Benchmarks (NPB) [16] show that the online profiler of FlowChecker incurs 0.9-5.6%, 0.9-8.1%, and 1.6-9.7% runtime overhead on Open MPI, MPICH2, and MVAPICH2, respectively. Furthermore, our results show that the trace size per process grows moderately, averaging 3.01 MB/min, 1.77 MB/min, and 10.08 MB/min on Open MPI, MPICH2, and MVAPICH2, respectively.

• FlowChecker is library-independent. This is because its message flow checking mechanism is independent of any particular communication algorithms used in MPI library implementations. Current implementation of FlowChecker supports MPI libraries that use TCP/IP or InfiniBand network protocols.

• FlowChecker requires no modification of source code. By using Pin [54], a dynamic binary instrumentation tool, FlowChecker works with binary code directly, requiring no modification of the source code of the MPI applications or MPI libraries. Likewise, FlowChecker requires no re-compilation of the MPI applications or MPI libraries.
FlowChecker provides informative bug reports. It provides programmers with accurate diagnostic information: the failed MPI calls, the ranks of failed processes, data and position information at message flow broken points. Such diagnostic information can help programmers understand root causes and fix the bug.

The rest of this chapter is organized as follows. Section 3.1 presents the main idea of FlowChecker and a real-world bug case. Section 3.2 describes the design and implementation of FlowChecker, followed by key design issues in Section 3.3. Then, Section 3.4 discusses the evaluation methodology, followed by the experiment results in Section 3.5. Finally Section 3.6 summarizes this chapter.

3.1 Main Idea of FlowChecker

3.1.1 Main Idea

The main idea of FlowChecker is to check whether the underlying MPI libraries correctly deliver the messages from the sources to the destinations as specified by the MPI applications. More specifically, FlowChecker first extracts the intentions of message passing (MP-intentions) from the MPI applications. Normally, MP-intentions are implied by the MPI function calls and the corresponding arguments made in the MPI applications. For example, FlowChecker can extract MP-intentions based on a matched pair of \texttt{MPI\_Send} and \texttt{MPI\_Recv} function calls.

Second, for each MP-intention, FlowChecker tracks the corresponding message flows by following the relevant data movement operations starting from the sending buffers at the source process. Data movement operations move data from one memory location to another within one process or between two processes [37]. Examples of
Figure 3.1: An example to illustrate the main idea of FlowChecker. The MP-intention here is \{A1\rightarrow D1, A2\rightarrow D2\} and the message flows are A1\rightarrow B1\rightarrow C1\rightarrow D1 and A2\rightarrow B2\rightarrow C2\rightarrow D2.

Data movement include memory copy and network send/receive. This step allows FlowChecker to understand how the MPI libraries perform message delivery.

Finally, FlowChecker checks message flows that are established at the second step against the MP-intentions extracted at the first step. If any mismatch is found, FlowChecker reports a bug and provides further diagnostic information such as faulty MPI functions or incorrect data movements.

Figure 3.1 illustrates the main idea of FlowChecker. In this example, process 1 invokes \texttt{MPI\_Send} to send a message stored at the buffer \{A1, A2\} to process 2, while process 2 invokes \texttt{MPI\_Recv} to receive the message and store the message at the buffer \{D1, D2\}. To deliver the message, the underlying MPI library first packs message data at the buffer \{A1, A2\} into the buffer \{B1, B2\}, then sends the data to the buffer \{C1, C2\} at process 2, and finally unpacks the data into the buffer \{D1, D2\} at process 2.

To handle the case in Figure 3.1, FlowChecker first extracts the MP-intention, which is \{A1\rightarrow D1, A2\rightarrow D2\}, based on the matched \texttt{MPI\_Send} and \texttt{MPI\_Recv} at process 1 and process 2, respectively. Then FlowChecker tracks the message flows, A1\rightarrow B1\rightarrow C1\rightarrow D1 and A2\rightarrow B2\rightarrow C2\rightarrow D2, in the underlying MPI library by analyzing...
the data movement operations `memcpy`, `send`, and `recv`. Finally, FlowChecker checks whether the message is correctly delivered by comparing the message flows against the MP-intention.

Programmers can easily make mistakes in the above communication steps due to rich MPI semantics. Examples of such mistakes include miscalculation of memory addresses in the buffer and incorrect datatype constructions. As a result of these mistakes, message data are often not delivered to destinations as specified by MPI applications. FlowChecker detects this type of bugs, referred to as communication-related bugs, and helps pinpoint the root causes of the bugs by reporting exact locations of incorrect message flows.
3.1.2 A Real-World Bug Case

Figure 3.2 (a) shows a simplified bug case extracted from Open MPI, a popular MPI library implementation. The function transmits a chunk of data from the sending buffer to the receiving buffer. However, the data are sent to a wrong position in the receiving buffer due to miscalculation of the destination address. The bug is at line 2, where the variable offset stores the number of elements with the datatype of MPI_INT. The library developers forgot to consider the size of MPI_INT, a 4-byte-long datatype. As a result, the data are sent to base+offset instead of base+offset*4, which is expected by MPI applications.

As shown in Figure 3.2 (b), the bug manifests itself at the last step of the message flow in the MPI library implementation. FlowChecker can identify the bug since the message is not delivered to the correct destination base+offset*4, as specified by the MPI application (we do not show the MPI application here for simplicity). Note that the bug will always lead to incorrect message flow, which is an invariant instead of an anomaly. Therefore, this bug cannot be detected by previous statistics-based approaches [37, 57].

3.2 FlowChecker Design and Implementation

3.2.1 Design Overview and Challenges

FlowChecker consists of four major components: Profiler, MP-Extractor, MF-Tracker, and MF-Checker. As shown in Figure 3.3, Profiler logs communication-related events at the MPI library and application levels into trace files. MP-Extractor extracts the MP-intentions specified by the MPI application. MF-Tracker tracks the message flows that occur in the MPI library. MF-Checker compares the extracted
MP-intentions and the corresponding message flows. If a mismatch is found, MF-Checker generates a bug report that provides detailed diagnostic information. Among the four components, Profiler is an online component, executing together with MPI applications and MPI library. The other three components analyze the trace files offline.

There are three key design challenges as follows. We address them in Sections 3.2.2, 3.2.3, and 3.2.4, respectively.

1. **How to profile the program execution efficiently?**

Profiler must incur low overhead since it is expected to be deployed at production runs. This requires Profiler to collect as few events as possible. Yet Profiler needs to collect as many events as possible so that other components can infer MP-intentions and actual message flows. Furthermore, it is desirable to make no modification to the source code of MPI applications and MPI library implementations because we may not have the source code of MPI applications, e.g., proprietary software.
(2) How to represent MP-intentions effectively and efficiently? In order to represent MP-intentions, the MP-Extractor needs to understand the MPI standard, which provides a wide range of MPI semantics. On the one hand, MPI semantics can be as simple as point-to-point transmission of message data that are stored in contiguous buffers. On the other hand, MPI semantics can be as complex as collective communication of user-defined non-contiguous datatypes. Furthermore, the representation should facilitate correctness checking by comparing the MP-intention with actual message flows. Therefore, it is important to represent MP-intentions in a general and efficient way.

(3) How to track actual message flows in MPI libraries? Tracking actual message flows is a difficult task without knowledge of library implementations. First, it is challenging to identify data movements that are relevant to an MPI call due to the huge number of data movement operations. Additionally, MPI applications may reuse the same buffer during execution. Second, to track message flows, FlowChecker needs to handle various types of data movement operations, especially for data movement over the network and shared memory channels. Third, different communication algorithms and optimizations for implementing an MPI function in different MPI libraries often complicate message flow tracking. For example, to implement `MPI_Gather`, a straightforward algorithm is to gather data directly from all processes. An alternative is the hierarchical approach, which gathers data to some intermediate processes that serve as local leaders and then transfers data to the root process. Moreover, MPI libraries often perform data movement operations that are related to non-MPI-application data such as control data for bookkeeping. These spurious data movement operations should be pruned to avoid interference with message flow checking.
3.2.2 Profiler: Collecting Communication Traces

Profiler logs communication and message flow events into trace files during program execution. More specifically, Profiler logs MPI function calls invoked in MPI applications as well as data movement operations made by MPI libraries. In addition to capturing the essence of MP-intentions and actual message flows, these coarse-grained events make Profiler efficient. In particular, Profiler logs three types of MPI calls. The first type is MPI communication routines such as \texttt{MPI\_Send} and \texttt{MPI\_Gather}. This type of routines are needed because they are the message flow sources and/or destinations. The second type is MPI datatype manipulation routines. Example includes \texttt{MPI\_Type\_contiguous} and \texttt{MPI\_Type\_struct}. They create a new datatype from primitive type (e.g., \texttt{MPI\_INT}) or existing user-defined datatypes. These routines are required because the message data may be stored in a memory region specified by the derived datatype. The third type is MPI basic supporting routines such as \texttt{MPI\_Comm\_rank} and \texttt{MPI\_Comm\_size}. This type of routines should be recorded since they contain the current process information and the whole communication group process information, which are required for checking message flow correctness.

At the MPI library, FlowChecker must know how the data is moved from source buffer to destination buffer. To profile such information, FlowChecker instruments data movement operations such as memory copy and network send/receive. Memory copy (e.g., \texttt{memcpy}) moves the data from one location to another within the same process. Network send and receive (e.g., \texttt{writev} and \texttt{readv}) moves the data from one process to another. All these routines are required to track message flows in the MPI library.
The trace grows moderately since Profiler only records necessary communication and data movement events. Its growth rate largely depends on the communication patterns of parallel applications and MPI library implementations. Our experimental results with HPL and NPB on three MPI libraries show that 4.95 MB of disk space on average can store a minute’s trace for each process (more details in Section 3.5.6). Since the space overhead is moderate, we did not apply any optimization techniques to the current implementation of Profiler. If writing trace causes large runtime overhead or large storage overhead, we can address this issue by leveraging various techniques used in previous work, such as lightweight file systems [66], node-local storage [21], and ScalaTrace [63].

The current implementation of Profiler uses Pin [54], a lightweight binary instrumentation tool, to instrument binary code of MPI libraries and MPI applications. As a result, Profiler requires neither source code modification nor re- compilation of MPI libraries and MPI applications.

3.2.3 MP-Extractor: Extracting MP-intentions

MP-intentions are implied by MPI calls made by the MPI applications. To extract MP-intentions, MP-Extractor first distills MPI calls from the collected trace files and store them in the MPI call list for each process. Then it matches corresponding MPI calls from different processes by following MPI standards, e.g., using the tags in MPI calls and the order of MPI calls. For example, MP-Extractor matches `MPI.Recv` with the corresponding `MPI.Send`. Similarly, it matches the group of corresponding `MPI.Gathers` in different processes. This matching method was also used by previous
work on detecting bugs in MPI applications [45,72]. After this step, each MP-intention consists of a pair or a group of matched MPI calls.

To facilitate message flow checking, MP-Extractor represents MP-intentions in a general and efficient way. More specifically, MP-Extractor utilizes *MPI call-pairs* to uniformly handle point-to-point and collective MPI calls. An MPI call-pair is a pair of corresponding MPI calls that transfer message data between two processes. For example, a matched MPI Send and MPI Recv is one MPI call-pair. As for a group of corresponding MPI Gathers in N processes, there are N MPI call-pairs, each with two MPI Gathers that transfer message data from a process to the root process.

Algorithm 1 shows how to match MPI calls. Lines 2-11 match point-to-point MPI calls and lines 12-32 match collective MPI calls. For a point-to-point MPI send call $f_1$, line 4 checks whether the src and tag of the MPI receive call $f_2$ are equal to the rank and tag of $f_1$. If yes, the call pair $\{f_1, f_2\}$ is stored into callpairlist and $f_1$ and $f_2$ are removed from their original lists. For a collective MPI call $f_1$, if $f_1$ is not invoked in the root process, we just skip that collective function call (lines 13-15). This function call will be processed when we handle the corresponding collective function call whose process is the root process. Lines 16-30 identify corresponding collective MPI calls with $f_1$ from all other ranks within the same communicator. The corresponding collective MPI calls should have the same function name as $f_1$ (line 21) and the same root process as $f_1$’s invoking process (line 24). After finding a corresponding collective MPI call, we add the call pair $\{f_1, f_2\}$ to the callpairlist and remove $f_2$ from its original list (lines 25-26). After processing the collective MPI calls corresponding to $f_1$ in all other ranks within the same communicator, we remove $f_1$ from its original list (line 31).
Algorithm 1 Matching MPI calls

1: for each MPI call $f_1$ in MPI call list $list_1$ of rank $n$ do
2:   if $f_1$ is a point-to-point MPI send then
3:     for each point-to-point MPI receive $f_2$ in MPI call list $list_2$ of rank $f_1$.dst do
4:       if $n = f_2$.src and $f_1$.tag = $f_2$.tag then
5:         callpairlist ← callpairlist + \{f_1, f_2\}
6:         list_1 ← list_1 \{f_1\}
7:         list_2 ← list_2 \{f_2\}
8:         break
9:     end if
10:   end for
11: end if
12: if $f_1$ is a collective MPI call then
13:   if $f_1$.root $\neq n$ then
14:     continue
15:   end if
16:   for each rank $m$ in all of the ranks in the $f_1$.comm do
17:     if $m = n$ then
18:       continue
19:     end if
20:     for each collective MPI call $f_2$ in MPI call list $list_2$ of rank $m$ do
21:       if func name($f_1$) $\neq$ func name($f_2$) then
22:         continue
23:       end if
24:       if $f_2$.root = $f_1$.root then
25:         callpairlist ← callpairlist + \{f_1, f_2\}
26:         list_2 ← list_2 \{f_2\}
27:       break
28:     end if
29:   end for
30: end for
31: list_1 ← list_1 \{f_1\}
32: end if
33: end for

It is worth to note that the wild cards such as MPI\_ANY\_SOURCE and MPI\_ANY\_TAG is a little tricky to handle and cannot be directly applied with Algorithm 1. This is because the wild card does not specify the source process of a message and thereby an MPI receive call that use the wild cards can be matched to a MPI send call from any process. In order to handle this case, FlowChecker tracks the value of MPI\_Status for each MPI receive call. In particular, when a MPI receive call completes, FlowChecker logs the real source and tag from the struct members of MPI\_Status. Then FlowChecker
Figure 3.4: An example of an MP-intention. The MP-intention is implied by a group of MPI\_Gathers with non-contiguous data in two processes. MP-Extractor represents the MP-intention as two MPI call-pairs, each with two transmission units.

treats the wild card MPI receive calls as regular ones and apply Algorithm 1 for matching MPI calls.

For each MPI call-pair, MP-Extractor further utilizes transmission units to handle both contiguous and non-contiguous data. A transmission unit represents transmission of a chunk of contiguous message data between two processes. More specifically, a transmission unit stores five key attributes of data transmission: the sending process (src\_rank), the base address of the sending buffer (src\_base), the receiving process (dst\_rank), the base address of the receiving buffer (dst\_base), and the length of the contiguous data (len). For transmission of non-contiguous data, MP-Extractor represents it as multiple transmission units. Figure 3.4 shows the representation of a group of corresponding MPI\_Gathers with non-contiguous data in two processes.
3.2.4 MF-Tracker: Tracking Message Flows

For each transmission unit in an MP-intention, MF-Tracker tracks the corresponding message flow from a sending buffer to a receiving buffer. A straightforward way is to understand and follow the communication algorithms implemented by the underlying MPI library. However, different MPI libraries may implement different communication algorithms for an MPI call, which makes this method not portable.

Instead, MF-Tracker uses a library-independent approach to track the message flow for a transmission unit. The main idea is to label and propagate the data source for each chunk of contiguous memory data involved in the message flow. A data source is defined as a 3-tuple $\langle \text{rank}, \text{base}, \text{len} \rangle$, which refers to the sending buffer in a transmission unit. More specifically, for each transmission unit, MF-Tracker first initializes the data source of the data stored in the sending buffer with the rank of the sending process, the starting address, and the length of the sending buffer. Then, MF-Tracker identifies subsequent data movement operations that are relevant to the transmission unit and propagates the data sources for each identified data movement operation. MF-Tracker continues to perform the previous step until no subsequent data movement operations can be found. Since this tracking method does not rely on any particular communication algorithm, FlowChecker is independent of MPI libraries.

Identifying subsequent data movement operations. For each chunk of data labeled with a valid data source, MF-Tracker identifies all its subsequent data movement operations. A data movement operation belongs to this group if the source buffer in the data movement operation overlaps with the buffer that holds the chunk
of data. For a data movement operation that moves data within one process, MF-Tracker simply calculates the intersection between its source buffer and the buffer holding the chunk of data.

The situation becomes complex when data movement operations involve the network (e.g., recv) or shared memory channels. The main reason is that these data movement operations have source buffers in different processes or have no explicit source buffers. To address this issue, MF-Tracker pre-processes the trace files by matching the corresponding data movement operations (e.g., send and recv) over network or shared memory channels. More specifically, for TCP/IP network operations, MF-Tracker models each connection as a couple of file descriptors in the sending and receiving processes after analyzing the POSIX-level socket management calls (e.g., connect and accept). Based on the modeled connections, MF-Tracker matches the corresponding POSIX-level data communication calls (e.g., send and recv). Similarly, for InfiniBand network operations, MF-Tracker models each connection as a couple of Queue Pair Numbers by analyzing network management calls (e.g., ibv_modify_qp). As for data movements through shared memory channels, MF-Tracker keeps track of the shared memory region by monitoring relevant operations (e.g., mmap and munmap) and matches the memory copies that operate on the same shared memory regions.

It is inefficient and inaccurate to search the entire trace files for the initial data movements that move data out of the sending buffer in a transmission unit. This is mainly because there are a huge number of data movements and MPI applications may reuse the same buffers during execution. To address this issue, MF-Tracker leverages the MPI standard and treats blocking and non-blocking MPI calls differently. For
Figure 3.5: An example of data source propagation for two transmission units. A, B, C, and D are four data movement operations. Buffers 1, 2, 3, 4, 6, 7, 9, and 10 store message data specified by the MPI application, while buffers 5 and 8 store control data used by the underlying MPI library.

For non-blocking MPI calls such as MPI_Isend, the MPI standard requires that the message data in the sending buffer are consumed before the corresponding MPI_Wait returns. Therefore, FlowChecker attempts to identify the initial data movement after the non-blocking MPI call and before the return of the corresponding MPI_Wait.

**Propagating the data source information.** For each subsequent data movement, MF-Tracker propagates the data source of the data stored in the source buffer to the data stored in the destination buffer. Figure 3.5 shows an example of the data source
propagation for two transmission units that involve four data movement operations $A$, $B$, $C$, and $D$. After two data movements $A$ and $B$, which pack non-contiguous data, the data sources of buffer 1 and buffer 2 are propagated to buffer 3 and buffer 4, respectively. Similarly, the data movement $C$ propagates the data sources of buffer 3 and buffer 4 to buffer 6 and buffer 7, respectively. Note that MF-Tracker treats buffer 3 and buffer 4 as two pieces of data since they contain different data source information, even though these two buffers are contiguous.

More often than not, MPI libraries transmit various control data along with MPI application data via memory copy or over the network. To handle these spurious data movements, MF-Tracker labels them with “invalid” data source so that they will not interfere with message flow tracking of MPI application data. Specifically, if the source buffer in a data movement operation contains partial data that have no data source label, this part of data will be labeled as “invalid”. As shown in Figure 3.5, buffer 5 and buffer 8 are labeled as “invalid” data sources in the data movement $D$.

In case of a data buffer being corrupted by another data movement, MF-Tracker labels the data source of the buffer as “invalid” data source. For example, if buffer 6 is corrupted by some data movement operations after the data movement $C$, MF-Tracker will faithfully label the data source of buffer 6 as “invalid” data source and then propagate to buffer 9. This propagation will facilitate FlowChecker for detecting such data corruption.
3.2.5 MF-Checker: Checking Message Flows and Reporting Errors

Once message flow tracking has been performed for a transmission unit, MF-Checker checks whether the message data in the sending buffer are delivered to the receiving buffer. Specifically, after MP-intention and message flows are extracted, MF-Checker compares the data sources of the receiving buffer with the sending buffer in the transmission unit. If they exactly match, MF-Checker determines that the transmission unit is fulfilled correctly by the underlying MPI library. If all transmission units of an MP-intention are fulfilled, the correctness checking for this MP-intention is done. Otherwise, MF-Checker reports a software bug and provides relevant diagnostic information.

Figure 3.6 shows the correctness checking by MF-Checker. From the MP-Extractor, FlowChecker knows that the MP-intention is that the data stored in buffer 1 should be send to buffer 2. After the message flow tracking, MF-Tracker propagates the data source of buffer 1 to the buffer 2. Then MF-Checker check whether the data source
of buffer 2 match with the data source of buffer 1. If yes, it means the MP-intention is fulfilled correctly by MPI library. If not, there must exists communication related bugs in the MPI library.

There are two types of mismatches between the intended sending buffer and the data source of the receiving buffer in a transmission unit. The first type occurs when the message flow is broken somewhere before reaching the receiving buffer. In this situation, MF-Tracker cannot identify the subsequent data movement operations at a certain step before the data reach the receiving buffer. The second type occurs when the data source of the receiving buffer does not match the sending buffer although the message flow reaches the receiving buffer. For this case, MF-Checker traces back through the message flow and reports the first data movement with a mismatch.

To help diagnose the bug, MF-Checker provides a bug report with detailed diagnostic information. More specifically, the bug report contains the following information: a) the failed MPI call-pairs. MF-Checker reports the MPI call-pairs which are unsatisfied by the message flows in the MPI libraries. b) the sources of the message data which are not delivered to the receiver. the message data source includes the source buffer address and the length of the data. c) the broken point data and location information. MF-Checker provides the broken point data addresses and the data lengths. In addition, MF-Checker provides the relevant data movement operations which contain the data and the callers of these data movement operations. These information can help programmer locate the root cause of the bugs. d) the most possible positions the data should arrive at if there is no bug in the program. The position information includes the relevant data movement operations and their caller.
With these information, developers can easily identify the root cause of the bug or narrow down the searching scope of the bug (more details in Section 3.5).

3.3 Issues and Discussion

Application-level bugs: When the MPI applications themselves have bugs, e.g., the buffer of MPI \texttt{Isend} is used in the subsequent MPI \texttt{Send} before the corresponding MPI \texttt{Wait}, the behavior of MPI libraries is undefined by the MPI standard. These may cause FlowChecker to report spurious errors. To handle this situation, previous approaches on detecting MPI application bugs [29,45,52,77] can be applied together with or before FlowChecker.

Message corruption bugs: FlowChecker can detect the communication bugs at the MPI library level that deliver the data to the incorrect destination or the bugs that corrupt message data via profiled data movements. However, FlowChecker cannot detect general message corruption bugs caused by incorrect data assignments such as buffer overflow. To address this issue, one way is to incorporate message checksum mechanisms. Another way is to enhance FlowChecker by tracking data movements at finer granularity such as byte level, where we need to address the challenges of large runtime and space overhead.

Data movements via value assignments: Although MPI libraries usually use the standard C library functions such as \texttt{memcpy} and \texttt{memmove} for data movements within a process, there are cases where data movements are implemented by value assignments. For example, MPI \texttt{Reduce} requires calculation on the data being transmitted. After calculating the data values, MPI libraries can directly assign them to the destination buffer. Even some general memory copies can be implemented by value assignments.
To address this issue, FlowChecker can leverage static analysis techniques [58] to identify such value assignments and transform them as data movements. More specifically, data flow analysis can help identify the sources and destinations of value assignments. If the source and/or destination addresses are adjacent, we know that they are a continuous data chunk that forms a larger grained data movement. Based on this information, we can turn on a flag to instruct FlowChecker to profile these value assignments and to mark them as a special data movement. An alternative is to rely on developers to provide relevant hints if predefined routines are used for moving data. In particular, developers provide the relevant routine names and parameters to Profiler so that these data movement routines can be logged for further analysis.

Data movements via other networks: The current implementation of FlowChecker traces data movements over TCP/IP and InfiniBand networks, which are the commonly used methods for network communication in High Performance Computing Systems. However, some MPI libraries use other types of high-performance networks such as Myrinet [59] and Quadrics [70]. To support these networks, we can extend FlowChecker by profiling and analyzing operations that use these network protocols.

Unsupported MPI features: Our current prototype of FlowChecker checks the most commonly used MPI features, such as point-to-point communication, collective communication, and user-defined data types, in MPI standard [3]. Advanced features such as one-sided communication and MPI-I/O, are not supported. To support one-sided communication, we need to modify device drivers or firmware of network interface cards to expose RDMA operations to the user-level process on the receiver side. These features are left for our future work.
3.4 Evaluation Methodology

Our experiments are conducted on a 64-processor cluster with 32 nodes. Each node is equipped with two processors, 8 GB memory, and 204 GB hard drive. Each processor is a 2.4 GHz AMD Opteron with 1 MB L2 cache. These nodes are connected by two network cards, one Gigabit Ethernet card and one InfiniBand card. The operating system running on the cluster is Linux 2.6.18. We use the cluster for online profiling. For analyzing the trace, we use a machine that has a 2.0 GHz Intel Xeon quad-core processor, 6 MB L2 cache, 16 GB memory, and 1 TB hard drive. We implement the online profiler of FlowChecker using Pin [54], a lightweight dynamic binary instrumentation tool, with Probe mode. For simplicity, we implement the trace-analyzing components in single thread, which can be improved by parallelizing the tasks in our future work.

We evaluate the effectiveness of FlowChecker with five real-world and two injected bug cases from three widely-used MPI library implementations, including Open MPI [36], MPICH2 [6], and MVAPICH2 [7]. The evaluated seven bug cases represent different types of mistakes, including a datatype construction error, packing/unpacking mismatch, incorrect pointer offset/address, incorrect memory copy direction, and incorrect sending buffer size. Additionally, to evaluate FlowChecker’s functionality of handling the wild card receive, we inject two bugs in Open MPI and use the MPI applications containing the wild card receive to trigger the bugs. We name these two bugs with bug ID #inj-1 and #inj-2 respectively. Table 3.1 shows the seven bug cases used in the evaluation. Note that we use the ticket numbers in their bug reporting systems as the bug IDs for Open MPI and MPICH2. As for

1 These bug cases do not imply the reliability of the MPI library implementations by any means.
Table 3.1: MPI libraries and bug cases used in the evaluation. Lib means Library, LOC means Lines of Code, and Desc. means Description.

MVAPICH2, the bug was discussed in the mailing list in March, 2007 and thereby we use the date as the bug ID.

We download the bug-triggering input (i.e., the driving MPI application) from the bug reporting web site for each bug case. To simulate the real-world scenarios of bug occurrences, we mix the bug-triggering input with normal inputs that do not trigger the bug. A normal input consists of point-to-point and collective MPI function calls.

We evaluate the runtime overhead and disk space overhead incurred by the online profiler of FlowChecker using recent versions of three MPI libraries (Open MPI-1.3.3, MPICH2-1.2.1, and MVAPICH2-1.4). Our driving MPI applications are High Performance Linpack [9] with the input sizes of 30,000, 40,000, and 50,000, and six NAS Parallel Benchmarks [16] with class C inputs in this set of experiments. To evaluate runtime overhead, we run each benchmark on top of each MPI library in two configurations, one with FlowChecker and the other without FlowChecker, each for 15 times, and calculate the average execution time for each configuration. To evaluate the disk space overhead, we run each benchmark on top of each MPI library once and calculate the size of the trace file for each process as well as the total size of the aggregated trace file for all processes.
<table>
<thead>
<tr>
<th>MPI Lib</th>
<th>Bug ID</th>
<th>Detected?</th>
<th>Pinpoint Root Causes?</th>
<th># of Processes</th>
<th>Trace Size</th>
<th>Execution Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open MPI-1.2</td>
<td>#209</td>
<td>Yes</td>
<td>No</td>
<td>64</td>
<td>31MB</td>
<td>7.99</td>
</tr>
<tr>
<td></td>
<td>#689</td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
<td>31MB</td>
<td>7.24</td>
</tr>
<tr>
<td></td>
<td>#1157</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>484KB</td>
<td>5.49</td>
</tr>
<tr>
<td>Open MPI-1.4.2</td>
<td>inj-1</td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
<td>35MB</td>
<td>10.34</td>
</tr>
<tr>
<td></td>
<td>inj-2</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
<td>1.1MB</td>
<td>8.52</td>
</tr>
<tr>
<td>MPICH2-1.0.6</td>
<td>#280</td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
<td>13MB</td>
<td>2.15</td>
</tr>
<tr>
<td>MVAPICH2-0.9.8</td>
<td>03/2007</td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
<td>25MB</td>
<td>7.70</td>
</tr>
</tbody>
</table>

Table 3.2: Overall results of FlowChecker for bug detection.

We evaluate the scalability of FlowChecker by using MVAPICH2 library with the same MPI application settings as those for runtime overhead experiments. In particular, we run these benchmarks with different processes, starting from 4. Due to limited resources of the cluster we are using, the maximum number of processes are set as 64. Note that FlowChecker logs the runtime events into local disk for each process, the file system is not expected to become a bottleneck when the scale gets larger. Some benchmarks can only be run on the number of processes which is square number. Others can only be run on the number of processes which is the power of 2. We get the runtime overhead for each benchmark on these different numbers of processes and present the results.

3.5 Experimental Results

3.5.1 Overall Results

Table 3.2 shows the overall results of FlowChecker for detecting bugs. For each bug case, we measure whether FlowChecker can detect the bug or not and whether the generated bug report pinpoints the root cause of the bug or not. Pinpointing root causes here means locating the exact function that contains the root cause of the
bug. In this table, we also report the number of processes running for triggering the bug. Additionally, this table presents the total size of the aggregated trace file from all processes as well as the trace size per process. Furthermore, the last two columns report the execution time for profiling the program and analyzing the aggregated trace file, respectively.

FlowChecker is effective in detecting bugs and pinpointing root causes of the bugs in MPI libraries. Table 3.2 shows that FlowChecker detects all seven evaluated bugs in the three MPI library implementations. Additionally, FlowChecker locates the exact functions that contain the bug for six out of seven bug cases. In the case that FlowChecker cannot pinpoint the root cause, the reported broken message flow helps developers locate the root cause, which is in a datatype construction function. FlowChecker is effective because it accurately captures the actual message flows in the underlying MPI libraries and checks them against the programmers’ intentions.

Table 3.2 also shows that FlowChecker’s detection capability does not depend on the size of traces or the scale of systems. For example, FlowChecker can detect bugs that manifest themselves in 64 processes as well as the bug being triggered in one process. This is mainly because FlowChecker is a rule-based method instead of a statistics-based method [84]. On the contrary, previous statistics-based approaches [37, 57] require large amount of trace data to detect bugs, which may not work for some MPI library bugs that only generate small trace files.

FlowChecker requires moderate disk space for trace files. Table 3.2 shows that the sizes of trace files range from 208 KB to 496 KB per process with the profiling time ranging from 2.15 seconds to 7.99 seconds. The main reason is that FlowChecker only
Figure 3.7: Bug case 1: Datatype construction error in Open MPI (Bug ID: #209)

profiles data movement related functions and MPI function calls. Section 3.5.6 shows similar experimental results using HPL and NPB on recent versions of MPI libraries.

FlowChecker quickly detects bugs in MPI libraries and generates bug reports. Our experimental results show that it takes FlowChecker from 0.02 seconds to 4.71 seconds to analyze trace files aggregated from all of the processes, whose sizes range from 484 KB to 31 MB. The analysis time is less than the profiling time in all the cases except for one bug case on MPICH2. This indicates that it is possible to streamingly analyze the trace concurrently with the profiling.

3.5.2 Case Studies for Effectiveness of FlowChecker

This subsection presents three representative bug cases, one from each MPI library.

Bug Case 1: Datatype Construction Error in Open MPI

This bug was found in Open MPI version 1.2. It can be triggered by invoking MPI_Send to send messages with a non-contiguous datatype. Once the bug occurs,
the receiving buffer cannot receive the data as expected, which likely causes silent errors in MPI applications.

The driving MPI application contains one bug-triggering input (i.e., one pair of \texttt{MPI\_Send} and \texttt{MPI\_Recv} with a non-contiguous datatype), mixed with diversified normal inputs. After being applied to this case, FlowChecker detects this bug and reports one unfulfilled MP-intention that consists of the exact pair of \texttt{MPI\_Send} and \texttt{MPI\_Recv} that triggers the bug. As shown in Figure 3.7, the transmission unit $B$ (buffer 2 $\rightarrow$ 8) fails due to incorrect message flow in the MPI library.

Among the seven evaluated bug cases, this is the only one in which FlowChecker does not pinpoint the root cause. However, FlowChecker provides useful information to help developers determine the root cause. As shown in Figure 3.7, FlowChecker cannot directly identify the message flow corresponding to the failed transmission unit $B$ since there is no data movement from buffer 2 to buffer 8. Nonetheless, FlowChecker reports the relevant transmission unit $A$ and its correct message flow buffer 1 $\rightarrow$ 3 $\rightarrow$ 5 $\rightarrow$ 7. By examining FlowChecker’s report, developers can quickly notice that the failed MPI call-pair has a message flow buffer 2$'$ $\rightarrow$ 4 $\rightarrow$ 6 $\rightarrow$ 8$'$ occurring together with the message flow for the transmission unit $A$. In fact, this observation leads the developers to the root cause, which is treating a non-contiguous datatype as a contiguous datatype mistakenly in the datatype construction function.

This bug may not be detected by prior statistics-based approaches [37,57] if only tracking the communication from the source to the destination buffers. The reason is that this bug neither causes any observable change in the execution time of each function in the path nor skews the data movement chain distribution. However, the corrupted data may lead to incorrect decision in the following programming execution
Figure 3.8: Bug case 2: Incorrect memory copy direction in MPICH2 (Bug ID: #280)

and thereby alter program flows. In this scenario, statistics-based approaches can detect this bug.

**Bug Case 2: Incorrect Memory Copy Direction in MPICH2**

This bug was found in MPICH2 version 1.0.6. It can be triggered by transmitting a sizable non-contiguous datatype in `MPI_Gatherv` calls. The bug corrupts data in the receiving buffer, which likely causes silent errors in MPI applications.

After applied to this case, FlowChecker detects the bug and reports that one out of 16,064 MP-intentions is not fulfilled by the corresponding message flows. In particular, the failed MP-intention is a group of `MPI_Gatherv` calls in 64 processes, which are equivalent to 64 MPI call-pairs. Among them, 24 call-pairs are not fulfilled due to failed transmission units. As shown in Figure 3.8, the transmission unit $B$ (buffer 2 → 8) fails due to incorrect message flow in the MPI library. These failed transmission units are caused precisely by the bug.

Furthermore, FlowChecker pinpoints the root cause of this bug. As shown in Figure 3.8, the failed transmission unit $B$ indicates that the message data in buffer
2 are not delivered to buffer 8. After tracing back the corresponding message flow, FlowChecker identifies that the first mismatch (i.e., the broken link) occurs between the data sources of buffer 2 and buffer 4. Additionally, FlowChecker reports the data movement operations and their functions that involve buffer 2 and buffer 4. In fact, the bug exactly resides in the reported function `MPID_Segment_vector_m2m`, where the developers mistakenly switched the source and destination buffers. With detailed diagnostic information from FlowChecker, developers can easily understand and fix this bug.

**Bug Case 3: Incorrect Sending Buffer Size in MVAPICH2**

This bug was found in MVAPICH2 version 0.9.8. It was triggered by executing a communication library for linear algebra, called BLACS (Basic Linear Algebra Communication Subprograms) [2]. The test program `xCbtest` reports “Invalid element” error after executing the BSBR (broadcast/send and broadcast/receive) test case. This bug causes data corruption, which happens silently and is caught at the last stage of result verification.

After being applied to this case, FlowChecker reports three unfulfilled MP-intentions which happen in the `MPI_Bcast` calls. As shown in Figure 3.9, the transmission units $A$ (buffer 1 → 7) and $B$ (buffer 2 → 8) fail. This is because incorrect sending buffer size makes the InfiniBand network sending another 8-byte buffer (buffer 9) to the remote process $P1$.

Furthermore, FlowChecker pinpoints the root cause of this bug. FlowChecker reports that the data in buffer 3 and buffer 4 are not sent out. This happens in the function `MPIDI_CH3_iSendv`, which is the exact function contains the bug. With such
Figure 3.9: Bug case 3: Incorrect Sending Buffer Size in MVAPICH2 (Bug ID: 03/2007)

diagnostic information, MPI library developers can quickly locate the bug and fix this bug.

3.5.3 Bug Report

The bug report is helpful for programmers to locate the bug. Figure 3.10 shows the bug report generated by FlowChecker for the bug case in Open MPI with Bug ID #689. It consists of four parts: the failed MPI call-pairs, the sources of the message data which are not delivered to the receiver, the broken point data and location information, and the most possible position where the data should reach if the program runs correctly. Note that some details are not shown in Figure 3.10.

Item 1 in Figure 3.10 shows the failed MPI call-pair. It can tell the programmers which MPI calls fail to deliver the data from source buffer to destination buffer. In this example, the MPI_Send fails to send the data in sending buffer to the receiving buffer specified by MPI_Recv. Item 2 shows the data source of the broken point data. It is the corresponding origin of the data which locates in the sending buffer. It can tell the
Bug report:

1. Failed MPI call-pair:
   MPI_Send and MPI_Recv
2. Data source of the broken point data:
   rank: 0
   buf: 0xcb66d3
   length: 1001
   location: MPI_Send
3. Broken point data and position:
   rank: 1
   buf: 0x1086fd6
   length: 1001
   location: memcpy
   Caller: ompi_unpack_homogeneous_contig
   Image: libmpi.so.0
4. The most possible next position if program runs correctly:
   Caller: mca_pml_ob1_recv_frag_match
   Image: mca_pml_ob1.so

Figure 3.10: Bug report for the bug case in Open MPI (Bug ID #689)

programmers which chunk of data are not delivered correctly in the sending buffer. Item 3 reports the information of the broken point data and their position. From this information, programmers know which data are not moved to the right buffer. Programmers can also know the relevant data movement routine which contains the data, the caller of the routine, and the binary image name that contains the caller. Item 4 shows the next position that the broken point data should possibly reach if the program runs correctly. This information can greatly help programmers narrow down the analysis scope for finding the root cause of the bug. With the help of the bug report, programmers can find and fix the bug quickly. If the library are compiled with debugging information (e.g., with “-g” option), FlowChecker can provide even more detailed diagnostic information such as buggy code file names and line numbers.

3.5.4 Runtime Overhead

Figure 3.11 (a), (b), and (c) show the execution time for the benchmarks with and without FlowChecker on Open MPI, MPICH2, and MVAPICH2, respectively.
The execution time for each benchmark is normalized to its baseline, which is the native execution without applying FlowChecker. The runtime overhead incurred by FlowChecker is very low, ranging from 0.9% to 5.6% with an average of 2.8% on Open MPI, from 0.9% to 8.1% with an average of 2.7% on MPICH2, and from 1.6% to 9.7% with and average of 4.8% on MVAPICH2. The runtime overhead is low because FlowChecker only logs a small number of function-level events that are related to MPI calls and data movement operations.
Currently, FlowChecker still performs the analysis step offline. Therefore, this section only discusses the runtime overhead incurred by the online profiler. In the future, we plan to extend the analysis components of FlowChecker to be performed online by utilizing stream processing algorithms on program traces.
Table 3.3: Disk space overhead of FlowChecker. T-Size means trace size, E-Time means execution time, and G-Rate means growth rate of the trace. Trace size reported here is per process.

3.5.5 Scalability Study

Figure 3.12 shows the scalability result of FlowChecker. Figure 3.12 (a) to (f) shows the runtime overhead of six NPB benchmarks on the different number of processes from 4 to 64. Figure 3.12 (g) to (i) shows the runtime overhead of HPL benchmarks with problem sizes 30000, 40000, and 50000 on different number of processes from 4 to 64. From these figures, we can see that FlowChecker is scalable, incurring modest overhead, i.e., less than 10%, for different numbers of processes. The main reason is that FlowChecker tracks each process independently.

3.5.6 Disk Space Overhead

Table 3.3 shows the sizes of the trace files, the execution time, and the per-process growth rates of the trace files for each benchmark on the three MPI libraries. The growth rate of trace files generated by FlowChecker is moderate, ranging from 0.65 to 10.78 MB/min with an average of 3.01 MB/min on Open MPI, from 0.12 to 5.56 MB/min with an average of 1.77 MB/min on MPICH2, and from 1.49 to 29.86
MB/min with an average of 10.08 MB/min on MVAPICH2. The space overhead is moderate because FlowChecker only profile a small number of function-level events that are related to MPI calls and data movement operations.

Table 3.3 also shows that the growth rate of trace file sizes varies for different MPI libraries. For example, the growth rate for BT is 1.14 MB/min, 1.07 MB/min, and 3.34 MB/min on Open MPI, MPICH2, and MVAPICH2, respectively. This is mainly because different MPI libraries may use different algorithms and/or different network protocols to implement the same MPI function calls. Additionally, the growth rate also depends on MPI applications. For example, the growth rates for the six NPB benchmarks on Open MPI vary from 0.83 to 10.78 MB/min. The main reason is that different MPI applications may have different communication patterns.

3.5.7 False Positive

In our experiments, FlowChecker reports zero false positives for all the evaluated MPI libraries with HPC benchmarks that cannot trigger bugs and the seven bug cases. In the case of HPL and NPB benchmarks, FlowChecker does not report any bugs since none has been triggered. In the case of the seven bug cases, FlowChecker only report the bugs which are real bugs in the MPI libraries. The reason why FlowChecker reports no false positives is because FlowChecker is a bug detection technique that tracks the exact message flows in the MPI libraries so that FlowChecker catches the exact message flow errors that are related to the communication bugs in the MPI libraries. No false positive is very important to programmers because the programmers can focus their limited resources on understanding the real bugs and fix them.
3.6 Summary

In this chapter, we have presented FlowChecker, a low-overhead method for detecting communication-related bugs in MPI libraries. Based on collected runtime traces, it extracts MP-intentions and checks whether the underlying message flows in MPI libraries fulfill the MP-intentions. If an MP-intention is not fulfilled, FlowChecker reports the bug and provides relevant diagnostic information.

We have built a prototype of FlowChecker. Our evaluation with five real-world and two injected bug cases in three popular MPI libraries, including Open MPI, MPICH2, and MVAPICH2, shows that FlowChecker detects all the evaluated bug cases effectively. Additionally, FlowChecker provides useful diagnostic information for narrowing down root causes of the bugs. In fact, FlowChecker pinpoints root causes for six out of seven evaluated bug cases. Furthermore, FlowChecker incurs low runtime overhead. Our experimental results with HPL and NPB show that FlowChecker incurs 0.9-5.6%, 0.9-8.1%, and 1.6-9.7% execution overhead on Open MPI, MPICH2, and MVAPICH2, respectively.
Chapter 4: Synchronization Error Detection

In this chapter, we present a new technique called SyncChecker to detect synchronization errors between MPI applications and underlying MPI libraries. Our main idea is to check, at runtime, whether the reuse of a message buffer in an MPI application is properly enforced to occur after the corresponding message data have been transferred by the MPI library. If not, SyncChecker reports this synchronization error with detailed diagnostic information to help developers understand the root cause and fix the bug.

Based on the above ideas, we have implemented a prototype of SyncChecker on Linux and evaluated it with seven bug cases including five introduced by the developers of the software and two injected by us. Our experiments have shown that SyncChecker effectively detects all the evaluated bugs and reports detailed diagnostic information. In summary, SyncChecker has the following advantages:

- SyncChecker accurately detects synchronization errors in MPI nonblocking communication. Our experimental results have shown that SyncChecker detects all of the seven evaluated bugs that reside in either nonblocking sends or nonblocking receives. Furthermore, SyncChecker provides detailed diagnostic information which helps programmers understand the root causes and fix the
bugs. SyncChecker’s effectiveness is due to the fact that it exploits semantic information in both MPI applications and the underlying MPI libraries.

- SyncChecker incurs moderate runtime overhead. Our experiments with seven NAS Parallel Benchmarks have shown that SyncChecker (with both online profiling and bug detection enabled) slows down program execution by 1.3-9.5 times with an average of 5.2 times. This indicates that SyncChecker is suitable for the testing phase. The reason of SyncChecker’s moderate runtime overhead is that it aggressively exploits three dynamic optimizations (see Section 4.3) for eliminating the number of profiled memory accesses that are irrelevant to nonblocking communication.

- SyncChecker is independent of system scales. Our experiments have shown that SyncChecker can detect the synchronization errors when running the programs with 8 processes as well as running with 64 processes. This is mainly because SyncChecker identifies these errors based on programming rules and semantics (i.e., the execution order of runtime events), instead of statistical invariants [37, 57]. Furthermore, SyncChecker’s performance is scalable since one detection process is running together with each MPI process on a local node.

- SyncChecker is easy to use. It requires no modification of source code and is independent of programming languages, e.g., supporting MPI applications written in C/C++ or Fortran. This is because SyncChecker instruments the binary code of MPI applications and MPI libraries using Pin [54], a lightweight dynamic binary instrumentation framework.
The rest of this chapter is organized as follows. Section 4.1 presents the main idea of SyncChecker, followed by the design and implementation of SyncChecker in Section 4.2. Then, Section 4.3 discusses three dynamic optimizations. Section 4.5 presents our evaluation methodology, followed by the experiment results in Section 4.6, and finally Section 4.7 summarizes this chapter.

## 4.1 Main Idea of SyncChecker

The main idea of SyncChecker is to examine whether the use of a message buffer in nonblocking communication is well synchronized between an MPI application and the underlying MPI library. In particular, for each nonblocking MPI call, SyncChecker monitors memory accesses to the message buffer in the MPI application and data movement operations in the MPI library, and checks whether their execution orders are enforced by an MPI completion check. The four types of runtime events monitored by SyncChecker are: (1) nonblocking MPI calls invoked by the MPI application ($NB_{App}$), (2) message buffer processing events, e.g., sending or receiving, at the library level ($SR_{Lib}$), (3) MPI completion checks invoked by the MPI application ($CK_{App}$), and (4) accesses to the message buffers by the MPI application ($ACC_{App}$). For each nonblocking MPI call $NB_{App}$, if the order of $SR_{Lib} \rightarrow ACC_{App}$ is enforced by a completion check $CK_{App}$, i.e., $SR_{Lib} \rightarrow CK_{App} \rightarrow ACC_{App}$, where $\rightarrow$ means “happens before” relation [46], SyncChecker considers it as the correct usage of nonblocking communication. Otherwise, SyncChecker reports a synchronization error since the MPI application may reuse the message buffer before the message data transmitted by the MPI library (i.e., $ACC_{App} \rightarrow SR_{Lib}$), leading to either sending out corrupted messages or reading undefined messages.
Figure 4.1 shows six scenarios with different execution orders of the relevant runtime events for nonblocking sends and receives. Specifically, Figure 4.1 (a) shows the correct execution order of these runtime events for a nonblocking send, i.e., $NB_{App} \rightarrow SR_{Lib} \rightarrow CK_{App} \rightarrow ACC_{App}$. In this scenario, the order of $SR_{Lib} \rightarrow ACC_{App}$ is enforced by a completion check $CK_{App}$. On the other hand, Figure 4.1 (b) shows an incorrect execution order, i.e., $ACC_{App} \rightarrow SR_{Lib}$, where the sending buffer is corrupted by the MPI application before the message is copied out or sent out by the underlying MPI library. Figure 4.1 (c) shows another incorrect execution order, where the synchronization error does not manifested itself as message corruption. In this scenario, however, the order of $SR_{Lib} \rightarrow ACC_{App}$ is not enforced by the completion check $CK_{App}$. As a result, the incorrect order of $ACC_{App} \rightarrow SR_{Lib}$ (as in scenario (b)) is still possible to occur in different program runs if the execution environment
changes, e.g., using another MPI library [77]. Similarly, Figure 4.1 (d), (e), and (f) show three corresponding scenarios for nonblocking receives.

SyncChecker detects the synchronization errors in scenarios (b) and (c) in the following way. For a nonblocking call \( NB_{App} \), if a memory access \( ACC_{App} \) is observed before either the message transfer event \( SR_{Lib} \) or the completion check \( CK_{App} \), SyncChecker reports the bug (scenario (b)). If the order of \( SR_{Lib} \rightarrow ACC_{App} \) is observed but no \( CK_{App} \) is performed between \( SR_{Lib} \) and \( ACC_{App} \), SyncChecker reports the potential bug (scenario (c)). In addition to reporting the synchronization error, SyncChecker provides diagnostic information, such as the problematic nonblocking communication call \( NB_{App} \), the observed order-violating memory accesses \( ACC_{App} \), and the relevant events \( SR_{Lib} \) and \( CK_{App} \) if available. The diagnostic information can help developers quickly understand and fix the bug.

It is worth noting that a completion check \( CK_{App} \) is often missing in real-world MPI programs, which is demonstrated by the bug cases in our experiments (see Section 4.6). The main reason might be that developers are often more familiar with MPI blocking communication than nonblocking counterparts. As discussed above, SyncChecker’s detection capability does not depend on the existence of the completion check event \( CK_{App} \) and therefore it can detect the synchronization error no matter the completion check is missing or not. In contrast, previous work Umpire [77] cannot handle the cases where the completion check is missing since it recalculates message checksum at the completion check.
4.2 Design and Implementation

4.2.1 Design Overview and Challenges

SyncChecker is composed of two main components, including Profiler and Analyzer, as shown in Figure 4.2. Profiler instruments the MPI applications and the underlying MPI library to profile the relevant runtime events during program execution and sends the information of the runtime events to Analyzer in a streaming fashion. By scanning these events, Analyzer then on-the-fly detects synchronization errors in MPI nonblocking communication, and provides diagnostic information to developers. The design is scalable since the Analyzer process only needs to analyze the runtime events that are local to each host.

There are two key challenges with the design of SyncChecker and we will address them in Section 4.2 and Section 4.3, respectively.
(1) How to effectively profile runtime events and detect synchronization errors? To accurately detect synchronization errors in nonblocking communication, SyncChecker must understand MPI’s rich semantics [4]. For example, MPI supports various data types ranging from simple ones such as \texttt{MPI_INT} to derived non-contiguous datatypes. Similarly, MPI nonblocking communication supports various completion checking semantics, including MPI blocking calls (e.g., \texttt{MPI_Wait}) and MPI nonblocking calls (e.g., \texttt{MPI_Test}).

(2) How to efficiently profile runtime events? MPI program execution can generate a large number of relevant runtime events, especially memory accesses at the MPI application level. With each memory access being profiled, Profiler can easily slow down MPI programs by 100 times [40, 61]. This creates a significant challenge for Profiler. Furthermore, the large number of profiled runtime events could overload Analyzer as well. Therefore, the key question is how to reduce the number of runtime events that are necessary for profiling as well as for analyzing.

4.2.2 Profiler: Collecting Runtime Information

To facilitate error detection in nonblocking communication, Profiler collects relevant runtime events in the MPI application and in the MPI library. Specifically, Profiler instruments two types of MPI calls at the application level. The first type is nonblocking communication and completion checking functions. Examples include \texttt{MPI_Isend}, \texttt{MPI_Irecv}, \texttt{MPI_Wait}, and \texttt{MPI_Test}. We need this type of runtime information for the locations of possible synchronization errors. The second type is MPI datatype manipulation functions (e.g., \texttt{MPI_Type_struct}). These functions create new datatypes from primitive types (e.g., \texttt{MPI_INT}) or existing user-defined datatypes.
This runtime information is required because the message data may be stored in a memory region defined as such derived datatype.

In addition to the relevant MPI calls, Profiler instruments every memory access at the MPI application level. This is because SyncChecker needs to detect order-violating memory accesses to the message buffer in a nonblocking call. Unsurprisingly, such fine-grained instrumentation for memory accesses can incur prohibitive runtime overhead, even for the testing phases. We will address this issue with three dynamic optimizations proposed in Section 4.3. Furthermore, memory management function calls in the MPI application need to be instrumented since these routines may access the message buffer too. For example, memcpy function call in the MPI application can overwrite the sending buffer in a nonblocking communication.

To detect synchronization errors, SyncChecker needs to know whether the message has been transmitted or not at the MPI library level. Instead of instrumenting each memory access in the MPI library, Profiler tracks data movement operations such as memory copy and network send/receive. This will not affect the detection capability of SyncChecker because the underlying MPI libraries often exploit such coarse-grained operations for transferring messages, i.e., copying out message to an intermediate memory location or directly sending message over the network [25, 37].

For each instrumented function call, Profiler records the function name and the arguments. For each instrumented memory access, Profiler records the access type (i.e., read or write), the memory address and the accessed memory size. Such information is sufficient for detecting synchronization errors and providing diagnostic information. During program execution, Profiler sends the collected information of these runtime events to Analyzer for on-the-fly error detection. Our current prototype
of SyncChecker uses UNIX domain sockets for fast communication between Profiler and Analyzer.

One legitimate concern is how to record the order of these events since Analyzer relies on the event order for error detection. Our current prototype of Profiler takes no special measures to handle the event order, since most existing MPI applications and the underlying MPI libraries are executed in a single-threaded process, and the event order is naturally preserved by the program execution. To handle future multi-threaded MPI library implementations and/or multi-threaded MPI programs, we plan to extend our Profiler by maintaining a global logical clock [46] and recording the clock value for each profiled runtime event.

We leverage a lightweight dynamic binary instrumentation tool Pin [54] to implement our current prototype of Profiler. In other words, Profiler performs the instrumentation on the binary code of the MPI application and the underlying MPI library. Therefore, Profiler is language-independent, e.g., working with MPI applications written in C/C++ or Fortran. Furthermore, Profiler does not require source code modification or recompilation of MPI applications and libraries.

4.2.3 Analyzer: Analyzing Runtime Events and Detecting Synchronization Errors

Analyzer receives the profiled runtime events and analyzes them for detecting synchronization errors in nonblocking communication. We next describe the error detection process, followed by the detailed discussion on processing each type of runtime events.
Detecting Synchronization Errors

To quickly detect the errors, Analyzer associates the message buffer in each MPI nonblocking call with a runtime state and performs the state transition based on the error detection state machine that implicitly contains the event ordering information. Figure 4.3 shows the state machine for detecting synchronization errors in MPI nonblocking communication. Take the nonblocking MPI send as an example. The state of a message buffer is initialized as Init when a nonblocking send (i.e., the event of \( NB_{App} \)) is invoked. After the message is sent over the network or copied out to a temporary system buffer at the MPI library level (i.e., the event of \( SR_{Lib} \)), Analyzer transits the buffer state from Init to LibDone. After this, if a completion check is performed at the MPI application level (i.e., the event of \( CK_{App} \)), the buffer state is transited to Safe, i.e., no synchronization errors were found. Once the state of a buffer becomes Safe, Analyzer stops processing future runtime events associated to the buffer. Otherwise, if the MPI application performs a memory write (i.e., the event of \( ACC_{App} \)) to the buffer when it is in the state of LibDone, Analyzer reports a potential synchronization error, i.e., the error that is not manifested during this particular program execution. If the MPI application performs a memory write to the message buffer with the state of Init, Analyzer reports a synchronization error since the message is overwritten by the MPI application before it is sent out by the underlying MPI library. Similar detection procedure is performed for an MPI nonblocking receive.

For a detected (potential) error, Analyzer provides detailed diagnostic information such as the MPI nonblocking methods, the message buffer information, the memory
Figure 4.3: The state machine for detecting synchronization errors

accesses that cause the synchronization errors or potential ones, relevant data movement operations in the MPI library, and the process rank. With the debugging information of an MPI program (compiled with “-g” option), Analyzer can map the above-mentioned diagnostic information to the line numbers, function names, and file names in the source code. This diagnostic information offers a significant help for developers to understand and fix the error.

**Processing Runtime Events**

Before error detection, Analyzer handles the following types of runtime events: datatype manipulation routines, nonblocking communication routines, memory access instructions and memory management routines, and data movement routines. The first three runtime events are from MPI applications and the last one is from the MPI library.

**Datatype manipulation routines.** MPI datatypes are complex, ranging from primitive types such as MPI_INT to non-contiguous memory regions that are created by datatype manipulation routines. To simplify processing datatypes, Analyzer uses a *data-map* data structure to represent each datatype. In particular, a data-map
consists of a series of segments, each segment represents the displacement and the length of a continuous memory chunk specified in a datatype. For example, the data-map of a datatype MPI\_INT is \{(0, 4)\}, where 0 is the displacement and 4 is the length of the datatype. Similarly, for a datatype that consists of two non-contiguous MPI\_INT’s with the gap of 4 bytes, the data-map is \{(0, 4), (8, 4)\}, where 0 and 8 are the displacements for the first and second MPI\_INT, respectively, and the two 4’s are the length for both MPI\_INT’s.

Analyzer uses a vector to store the data-maps for all datatypes in an MPI program. Initially, Analyzer creates a data-map for each primitive datatype, such as MPI\_INT, and stores them in the vector. After receiving an event of a datatype manipulation routine, Analyzer calculates the lower bound, upper bound and data-map of the new datatype based on the arguments in the routine and existing data-maps information in the vector. Then Analyzer stores the data-map of the new datatype into the vector for future use.

**Nonblocking communication routines.** MPI nonblocking communication includes the initialization routines such as MPI\_Isend and MPI\_Irecv, and the completion checking routines such as MPI\_Wait and MPI\_Test. To handle an initialization routine, Analyzer creates a new record containing the starting address of the message buffer, the number of elements, the datatype of each element in the message buffer, the request handle value, and the initial state Init for the buffer. Then Analyzer stores the newly-created record for the message buffer in the send or receive list.
It is a little complicated to handle the completion checking routines since they may have blocking or nonblocking semantics. For example, `MPI_Wait` uses blocking semantics, i.e., the function will not return until the specified nonblocking send/receive completes. Differently, `MPI_Test` uses nonblocking semantics, i.e., the function will return immediately no matter the specified nonblocking send/receive completes or not. To handle blocking completion checking routines such as `MPI_Wait`, Analyzer identifies the buffer record in the send or receive list based on the request handle, then performs buffer state transition based on the error detection state machine in Figure 4.3. If the state transits to Safe, Analyzer removes the buffer record from the corresponding list. To handle nonblocking completion checking routines such as `MPI_Test`, Analyzer checks the status flags that are returned from the function call to see whether the specified nonblocking send/receive function completes or not. If the status flags indicate the completion of the specified nonblocking send/receive, Analyzer performs the same steps as the ones above-mentioned for handling `MPI_Wait`. Otherwise, Analyzer does nothing since the nonblocking send/receive has not completed yet.

**Memory access instructions and memory management routines.** To handle memory accesses issued in the MPI application, Analyzer identifies the accessed message buffer in the send or receive list. Specifically, for each memory access instruction and memory management routine such as `memcpy` or `free`, Analyzer calculates the intersection between the memory address range in the access event and the ranges of the message buffers in the send or receive list, depending on the access type. If no intersection is found for all the message buffers, Analyzer simply discards the runtime event since the memory access is irrelevant to the nonblocking communication. Otherwise, Analyzer performs the state transition for the identified message buffer
based on the error detection state machine. If an error or a potential error is detected, Analyzer provides diagnostic information. Note that Analyzer only need to check the send list for write accesses and the receive list for read accesses.

**Data movement routines from the MPI library.** The underlying MPI library performs data movement operations that move data from one memory location to another location or send data from memory to network cards. Some data movement operations are related to nonblocking communication while others are not. To identify relevant data movement operations, Analyzer calculates the intersection between the buffer in the data movement operations and the buffers in the send or receive list. If no intersection is found, Analyzer simply discards the events of data movements since they are irrelevant to nonblocking communication. Similar technique has been used in our prior work [25, 37]. Otherwise, Analyzer performs the state transition for the identified message buffer based on the error detection state machine in Figure 4.3.

### 4.3 Profiler Optimizations

The basic design of Profiler presented in Section 4.2 instruments every memory access at the MPI application level for error detection. Such fine-grained profiling can easily slow down program execution by hundreds times as shown in our experiments (see Section 4.6), making our tool inapplicable in testing environment. To reduce overhead, Profiler employs three dynamic optimizations: (1) Execution Region (ER) optimization that eliminates profiling efforts when there is no nonblocking communication; (2) Access Type (AT) optimization that treats nonblocking send and receive differently; and (3) Memory Region (MR) optimization which eliminates profiling efforts for memory accesses that are out of the buffer range.
4.3.1 Execution Region (ER) Optimization

Profiling all the memory accesses at runtime is unnecessary for error detection. We observe that synchronization errors in nonblocking communication can only occur between the nonblocking send/receive calls and the corresponding completion checking calls (if available). In other words, if there are no pending nonblocking communication calls, Profiler does not need to collect memory access information during program execution. This is the basic idea for our Execution Region (ER) optimization.

To perform ER optimization, Profiler records each nonblocking send or receive call in the send or receive list, respectively. When a completion check has finished (checking the status flags for nonblocking completion check routines such as MPI_Test), Profiler removes the nonblocking send/receive call from the send/receive list. For each instrumented memory access, Profiler checks whether both lists are empty. If yes, Profiler simply ignores the memory access without sending it to Analyzer. Otherwise, Profiler performs the basic profiling function, i.e., collecting memory access information and sending it to Analyzer.

For correct MPI programs, ER optimization significantly reduces runtime overhead and avoids sending unnecessary runtime events to Analyzer since most of the memory accesses are not within the execution regions of nonblocking communication. On the other hand, for buggy MPI program, ER optimization is conservative – keep profiling memory accesses when a completion checking routine is missing.

4.3.2 Access Type (AT) Optimization

After ER optimization, profiling information for all the memory accesses within the execution region may still be unnecessary. In particular, for nonblocking sends,
Analyzer only needs to check memory write accesses to see whether the message in the sending buffer is overwritten or not. Similarly, for nonblocking receives, Analyzer only needs to check memory read accesses to see whether the receiving buffer has been read before the message is ready. This observation leads to the second optimization, Access Type (AT) optimization, which profiles write accesses for nonblocking sends and read accesses for nonblocking receives. Note that memory writes in applications may corrupt message buffers for nonblocking receives. To detect such error, AT optimization can be relaxed by also checking writes for nonblocking receives. However, such error is out of the scope of this chapter since it exists even with correct application-library synchronization.

AT optimization works as follows. For each instrumented memory write access, Profiler only checks whether the send list is empty or not. If yes, Profiler returns immediately. Otherwise, Profiler performs the basic profiling function. Profiler performs similar check for memory read accesses, i.e., only checking the receive list. AT optimization reduces overhead by saving half of the send/receive list checking effort and eliminating the profiling of memory accesses whose types do not match the nonblocking communication types.

### 4.3.3 Memory Region (MR) Optimization

Even after applying ER and AT optimizations, not all memory accesses are relevant to the message buffers in nonblocking sends or receives. Instead, there are accesses to other memory regions. Therefore, Profiler applies the third optimization, Memory Region (MR) optimization. The main idea is to only profile memory accesses that are within the range of message buffers in nonblocking communication. Like AT
optimization, MR optimization handles write accesses for nonblocking send and read accesses for nonblocking receive.

One straightforward way to implement MR optimization in Profiler is to maintain message buffer and datatype information for each nonblocking communication call, and then search all the message buffers given a memory access event (i.e., similar to what Analyzer does). However, it is very time consuming to perform such fine-grained search for each memory access. Instead, we implement a more efficient MR optimization, which only performs coarse-grained search. The main idea of our MR optimization is to maintain one memory range (i.e., lower and upper bounds) of all the message buffers in the send list and one memory range for the receive list, and then check the memory range in the send list or receive list for each instrumented write or read access, respectively. Whenever a new nonblocking send or receive is performed, Profiler updates the memory range in the send list or the receive list with the new buffer bounds. Specifically, for the memory range in the send list, the lower bound is the minimum one among the lower bounds of all the sending buffers in the list. The upper bound is the maximum value among the lower bounds of all the sending buffers in the send list, plus the corresponding buffer length. The buffer length should be calculated from the extent of the corresponding datatypes. To avoid processing complex datatype information, Profiler conservatively uses a large threshold value for the buffer length. For different MPI programs, developers can specify the threshold with different values. The above mentioned steps for memory range calculation is also applicable to the receive list.
Algorithm 2 Profiler Optimizations

1: for each memory write access $addr$ do
2:   if $sendlist$ is non-empty then
3:     if $addr \geq sendlist\text{.minaddr}$ and $addr \leq sendlist\text{.maxaddr}$ then
4:       Profiling the memory write access
5:     end if
6:   end if
7: end for
8: for each memory read access $addr$ do
9:   if $recvlist$ is non-empty then
10:      if $addr \geq recvlist\text{.minaddr}$ and $addr \leq recvlist\text{.maxaddr}$ then
11:        Profiling the memory read access
12:      end if
13:   end if
14: end for

4.3.4 Summary of the Optimizations

Algorithm 2 shows the summarized algorithm for all the three optimizations. More specifically, with AT optimization, Profiler instruments memory writes at lines 1-7 and instruments memory reads at lines 8-14. Lines 2 and 9 check whether the send list and the receive list are empty or not, respectively i.e., ER optimization. Lines 3 and 10 check whether the memory access is within the boundary of the memory ranges of the send list or the receive list, respectively, i.e., MR optimization. Lines 4 and 11 collect the information of the memory access and sending it to Analyzer, as what the basic profiling does. For clarity, we skip the code for maintaining $sendlist$ and $recvlist$ in this algorithm.

Profiler performs the optimizations dynamically by instrumenting each memory access in the binary code of the MPI application. As shown in Algorithm 2, each optimization reduces the number of profiled memory accesses, which are expensive
due to I/O operations, at the cost of performing an additional check at each memory access. Note that none of the three optimizations sacrifices SyncChecker’s capability of detecting synchronization errors in nonblocking communication. This is because the optimizations are all conservative – only eliminating profiling memory accesses that are irrelevant to the targeted errors.

4.4 Issues and Discussion

Data movement via hand-coded routines: Although most MPI libraries use general data movement routines such as memcpy, some may use their own hand-coded routines. To address this issue, we can rely on programmers to pass the routine interfaces such as routine names and parameters to SyncChecker so that SyncChecker can intercept and analyze them similarly as general data movement routines.

Completion guaranteed by other mechanisms: The completion of the nonblocking sends/receives are usually guaranteed by completion checking routines. In some scenarios, however, the completion of nonblocking communication are enforced by other mechanisms, e.g., succeeding blocking MPI calls. To handle this case, we need to extend our proposed mechanism to track the happens-before relations among such runtime events. Once the order between buffer reuse in the MPI application and the corresponding sending/receiving events in the MPI libraries is correctly enforced by happens-before relations, we will consider it as correct nonblocking communication.

4.5 Evaluation Methodology

Our experiments are conducted on two partitions of the Glenn cluster at Ohio Supercomputer Center [65]. One partition contains 877 computer nodes. Each node
<table>
<thead>
<tr>
<th>MPI Apps</th>
<th>#LOC</th>
<th>Bug IDs</th>
<th>Bug Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athena-r1086</td>
<td>89,549</td>
<td>#1093</td>
<td>nonblocking send</td>
</tr>
<tr>
<td>Athena-r1090</td>
<td>89,749</td>
<td>#1095</td>
<td>nonblocking send</td>
</tr>
<tr>
<td>octopus-r1278</td>
<td>37,772</td>
<td>#1284</td>
<td>nonblocking send</td>
</tr>
<tr>
<td>Boost-app</td>
<td>133</td>
<td>10/2010</td>
<td>nonblocking send</td>
</tr>
<tr>
<td>Sort</td>
<td>156</td>
<td>07/2007</td>
<td>nonblocking receive</td>
</tr>
<tr>
<td>Athena-r1086-cc</td>
<td>89,549</td>
<td>#1093-cc</td>
<td>nonblocking send</td>
</tr>
<tr>
<td>octopus-r1278-cc</td>
<td>37,772</td>
<td>#1284-cc</td>
<td>nonblocking send</td>
</tr>
</tbody>
</table>

Table 4.1: Evaluated applications and synchronization errors. Note that Athena-r1086 means the MPI application Athena with the revision number 1086. "r" has the same meaning in the names of other applications.

is a dual-core machine with 2.3 GHz AMD opteron CPU, 8 GB RAM and 48 GB local disk space. The other partition contains 650 computer nodes. Each node is a quad-core machine with 2.5 GHz AMD opteron, 24 GB RAM and 393 GB local disk space. The operating system running on the cluster is Linux 2.6.18. Note that we perform our experiments for each application with different configurations on one system-assigned partition so that the performance results can be normalized to the native runs. We implement Profiler of SyncChecker using Pin [54], a lightweight dynamic binary translation framework. Additionally, we implement Analyzer of SyncChecker using two threads, i.e., one for receiving runtime events from Profiler and the other for processing the events and detecting errors. In our experiments, Analyzer is running together with Profiler for each MPI process on a local node.

We evaluate the effectiveness of SyncChecker using four different real-world applications as shown in Table 4.1, including (1) Athena [14], a grid-based application for astrophysical magnetohydrodynamics; (2) octopus [64], a simulator for electron-ion
dynamics; (3) Boost-app, an application using Boost.MPI, which is a C++-friendly interface to the standard MPI [39]; and (4) Sort, an integer sorting algorithm using MPI. These four applications consist of various lines of code and contain seven different synchronization errors residing in a nonblocking send and/or a nonblocking receive. Five of the synchronization errors have no completion checks and were introduced by the original application developers. We have not yet located MPI applications that contain synchronization errors with mislocated completion checks, i.e., the potential errors shown in Figure 4.1 (c) and (f). To evaluate SyncChecker’s functionality of detecting such errors, we injected completion checks after memory accesses of sending buffers in Athena-r1086 and octopus-r1278, renaming them as Athena-r1086-cc and octopus-r1278-cc, respectively.

To evaluate the efficiency of SyncChecker, we run seven NAS Parallel Benchmarks (NPB) [16] with class C inputs on 64 processors. All of the seven NPB benchmarks contain various numbers of nonblocking communication function invocations (none in EP). We evaluate the impact of Analyzer on the overall runtime overhead of SyncChecker as well as the performance benefits brought by the three optimizations for Profiler. Specifically, we measure the program execution time with the following six configurations. In the first five configurations we only enable Profiler and redirect the runtime events to /dev/null. In the six configuration, we enable both Profiler and Analyzer.

- Native: Executing the benchmarks without applying SyncChecker.
- Profiler-basic: Executing the benchmarks with non-optimized Profiler

71
Table 4.2: Overall effectiveness of SyncChecker for bug detection

- Profiler-ER: Executing the benchmarks with Profiler and ER optimization enabled.
- Profiler-ER-AT: Executing the benchmarks with Profiler and ER and AT optimizations enabled.
- SyncChecker-P: Executing the benchmarks with Profiler and all the three (i.e., ER, AT and MR) optimizations enabled.
- SyncChecker-PA: Executing the benchmarks with optimized Profiler and Analyzer (i.e., the entire tool SyncChecker).

### 4.6 Experimental Results

#### 4.6.1 Overall Effectiveness

Table 4.2 shows the overall effectiveness of SyncChecker. For each bug case, we measure whether SyncChecker can detect it. For comparison purposes, we analyze each case to see whether it can be detected by previous work Umpire [77]. Additionally, we report more detailed results for SyncChecker’s detection capability, including
(1) whether the error code region misses a completion check; (2) where (i.e., non-blocking send or receive) each error locates; (3) failure symptoms once the error is triggered; and (4) the number of processes running for triggering each error.

SyncChecker is effective in detecting the synchronization errors for MPI programs with nonblocking communication. As shown in Table 4.2, SyncChecker effectively detects all of the five real bug cases that miss completion checks and two injected ones that invoke completion checks. Additionally, SyncChecker’s effectiveness is not affected by the error locations or failure symptoms. For example, SyncChecker detects the errors in Athena-r1086’s nonblocking send and Sort’s nonblocking receive, where the errors cause the program to crash. Similarly, SyncChecker locates the errors in octopus-r1278’s and Boost-app’s nonblocking sends, where the errors cause the program to hang and generate incorrect results, respectively. SyncChecker’s effectiveness in detecting synchronization errors is because it accurately captures the essential runtime events, i.e., relevant memory accesses in MPI applications and data movement operations in MPI library, and their relative execution orders.

In contrast, Umpire can only detect the last two injected synchronization errors. This is because Umpire relies on the completion checking call at nonblocking sends for re-calculating checksum and cannot detect synchronization errors for nonblocking receives. However, all the five real bug cases at the nonblocking sends miss invoking the completion checking routines. This also indicates that programmers tend to forget the completion check routines due to unfamiliarity with nonblocking communication. Note that Umpire can detect many other types of MPI bugs, while SyncChecker focuses on synchronization errors in MPI nonblocking communication.
Table 4.2 also shows that SyncChecker’s detection capability does not depend on the scale of running processes. For example, SyncChecker detects the error when running the Athena-r1086 with 64 processes as well as catches the error when running Athena-r1090 on 8 processes. The reason why SyncChecker’s detection capability is oblivious to system scales is because SyncChecker utilizes programming rules and semantics instead of statistics-based program invariants [49,84]. In contrast, previous statistics-based approaches [37, 57] cannot handle bug cases that only manifested themselves in a small system scale since these approaches require collecting a large number of statistical data for error detection.

As shown in Table 4.2, SyncChecker is able to handle the MPI applications written in different languages, including C, C++ and Fortran. In contrast, some previous MPI bug detection tools (e.g., [52]) can only deal with applications written in one programming language. The reason for SyncChecker’s language independence is that it utilizes a binary instrumentation framework, Pin [54], to directly work on the binary code of the MPI applications and the MPI library for profiling and analyzing runtime events.

4.6.2 Case Studies

This subsection presents two representative cases of synchronization errors, one from Athena written in C and the other from octopus written in Fortran.

Case 1: Sending Buffer Overwritten in Athena

This case was found in Athena with the revision number 1086. The error causes the program to crash once it is triggered. Figure 4.4 shows the buggy code extracted from the source files. First, a nonblocking send `MPI_Isend` is invoked at line 3 in
function `bvals_mhd`, which intends to send the message in the buffer `send_buf[0]` to another process. Without performing a completion check, the buffer `send_buf[0]` is overwritten at line 12 in a different function `pack_ix1` being invoked by function `bvals_mhd`. Moreover, the buffer is overwritten via an aliased pointer `pSnd` instead of the original buffer pointer `send_buf[0]`.

This error can be triggered by running Athena in 64 processes with certain error triggering inputs. After being applied to this error case, SyncChecker reports that, out of the total 71.6 billions memory accesses, 1.25 millions write accesses overwrite the nonblocking send buffers before the data are sent out. Furthermore, SyncChecker pinpoints the root cause of this synchronization error by successfully locating the functions `bvals_mhd` and `pack_ix1`, the relevant MPI nonblocking send at line 3, and the buggy memory access statement at line 12 as shown in Figure 4.4. Additionally, SyncChecker reports the incorrect execution order between the memory accesses at the MPI application and the message sent out event at the MPI library (skipped in
Figure 4.5: Bug case 2: Sending buffer overwritten in octopus, which is written in Fortran.

Figure 4.4 for simplicity). With such detailed diagnostic information, programmers can quickly understand and fix the error.

It is worth noting that the violating memory accesses are convoluted since they are in a different function via an aliased pointer. This creates significant challenges for static bug detection methods. Furthermore, this error can not be detected by Umpire because the completion checking function is missed by programmers, which is a common reason causing synchronization errors in real-world MPI programs with nonblocking communication.

Case 2: Sending Buffer Overwritten in Octopus

This synchronization error was found in octopus with the revision number 1278. The error causes the program to run very slowly and then hang at certain program location. Figure 4.5 shows the buggy code extracted from the source files. The code contains a do loop between line 3 and line 10. After assigning the data to send_buffer at line 5, it intends to send the data to another process at line 7. However, line 7
Figure 4.6: Runtime overhead of SyncChecker. “Native” means execution without applying SyncChecker, “SyncChecker-P” means execution with Profiler only, “SyncChecker-PA” means execution with both Profiler and Analyzer enabled.

is a nonblocking send and there is no completion checking function afterward. After executing line 7, the program may re-execute line 5 in other iterations of the do loop. The re-execution of line 5 may corrupt last message in the buffer send_buffer before the message is sent out by the underlying MPI library.

This error can be triggered in 64 processes with certain error triggering inputs. After being applied to this case, SyncChecker reports that there are 22.3 thousands error-relevant memory write operations out of 43.6 millions memory writes in total. Furthermore, SyncChecker locates the exact subroutine X(oep_x), the relevant nonblocking send at line 7, the buggy memory access statement at line 5 as shown in Figure 4.5, as well as the incorrect execution order of memory accesses and the message sent event. Such detailed diagnostic information can help programmers understand and fix the synchronization error.

Note that octopus is written in Fortran. The error case demonstrates that SyncChecker is a language-independent tool, i.e., applicable to MPI applications written in C/C++ or Fortran. This is because SyncChecker’s instrumentation is at the binary level instead of at the source code level.
4.6.3 Runtime Overhead

Figure 4.6 shows the execution time of seven NAS Parallel Benchmarks without SyncChecker, with SyncChecker’s Profiler only, and with both Profiler and Analyzer. The execution time for each benchmark is normalized to the native execution without SyncChecker. As shown in Figure 4.6, the runtime overhead incurred by SyncChecker’s Profiler is moderate, ranging from 96% to 6.73 times with an average of 3.85 times. The reason for the modest overhead is that SyncChecker aggressively exploits three dynamic optimizations, which substantially reduces the number of profiled memory accesses.

Additionally, Figure 4.6 shows that SyncChecker’s on-the-fly Analyzer is very efficient. For example, executing Profiler together with Analyzer only adds a little additional overhead, ranging from 32% to 2.8 times with an average of 1.4 times, compared to executing the programs with Profiler only. Otherwise, if we save the runtime event logs to disk and process them offline, it takes much longer time to analyze due to expensive I/O operations, and costs extra disk space.

Overall, SyncChecker incurs runtime overhead ranging from 1.3-9.5 times with an average of 5.2 times, which is acceptable for software testing. Our current prototype of SyncChecker only performs dynamic optimizations to reduce overhead. We leave applying static analysis to further reduce runtime overhead as future work. For example, we can statically identify the functions where memory accesses are unnecessary for profiling due to irrelevance to nonblocking communication.
4.6.4 Effects of the Optimizations

Figure 4.7 shows the runtime overhead incurred by Profiler with different optimizations applied. The result clearly indicates that the three optimizations are very effective in reducing runtime overhead incurred by Profiler. For example, the basic Profiler without any optimizations slows down all the benchmarks by more than 100 times. With all three optimizations applied, the runtime overhead incurred by Profiler is reduced to an average of 3.85 times for the seven benchmarks. The main reason is that the three optimizations significantly reduce the number of profiled memory accesses as shown in Figure 4.8. Next we will discuss benefits brought by each optimization.

We first apply ER optimization since we observe that many memory accesses are outside of the execution region of nonblocking communication and thereby can be eliminated by ER optimization. As shown in Figure 4.7, after applying ER optimization, the overhead is reduced from more than 200 times to less than 10 times for most of the evaluated benchmarks. This is echoed by substantial reduction of profiled memory accesses as shown in Figure 4.8. For example, after applying ER, the number of profiled memory accesses per process for BT is reduced from 56.3 billions to 221 thousands, which leads to the overhead being reduced from more than 200 times to 6.47 times.

AT optimization further reduces the overhead incurred by Profiler with ER enabled. As shown in Figure 4.7, although the overhead reduction is not obvious for most of the benchmarks, it is still worth to perform AT optimization for some benchmarks such as MG, where the overhead is reduced from 20.1 times to 8.48 times. This
Figure 4.7: Optimization effects of runtime overhead with Profiler enabled only. The runtime overhead of “> 200” incurred by Profiler-basic is because we terminate the program execution after that long time. “Native” means execution without applying SyncChecker, “Profiler-basic” means execution without any optimization, “Profiler-ER” means execution with ER optimization enabled, “Profiler-ER-AT” means execution with ER and AT optimizations enabled, “SyncChecker-P” means execution with ER, AT and MR optimizations enabled.

is because AT optimization further reduces the number of profiled memory accesses for MG from 80.3 millions to 13.7 millions, i.e., by almost 5 times.

After enabling ER and AT optimizations, Profiler can further reduce the runtime overhead by applying MR optimization. While most of the benchmarks have reached their minimum overhead after perform ER and AT optimizations, the runtime overhead for benchmark SP is still high, i.e., 55 times. After performing MR optimization, the runtime overhead for benchmark SP is reduced to 1.78 times, as shown in Figure 4.7. This is because the number of profiled memory accesses for SP is reduced from 4.28 billions to zero, as shown in Figure 4.8. Moreover, MR optimization is especially useful for MPI applications which miss the completion checks. This is because missing completion checks disables ER and AT optimizations to eliminate many memory accesses due to no ending boundaries of the nonblocking execution regions.
Figure 4.8: The numbers of profiled memory accesses per process with different level of optimizations. For clarity, we put a tiny bar with “0” on the top indicating no memory access. Also note that y-axis is in a logarithmic scale. “Profiler-basic”, “Profiler-ER”, “Profiler-ER-AT”, and “SyncChecker-P” have the same meanings as those in Figure 4.7.

While the three optimizations greatly reduce the runtime overhead, they will introduce some additional overhead because the optimizations are performed dynamically and the introduced conditional checks consume system resources. For example, benchmark EP does not contain any nonblocking communication, as indicated by the profiled memory accesses number after ER optimization. Due to additional checks, Profiler slows down EP by 1.04 times after ER optimization.

4.7 Summary

In summary, this chapter presents SyncChecker, a new approach to detect synchronization errors when MPI programs use nonblocking communication. Based on the profiled runtime events, SyncChecker checks whether the use of the message buffers in nonblocking communication is well synchronized between MPI programs and the underlying MPI library. If not, SyncChecker reports the error with detailed information to assist bug diagnosis.
We have built a prototype of SyncChecker on Linux. Our evaluation with five real-world and two injected bug cases in four different MPI applications shows that SyncChecker is effective in detecting the synchronization errors in nonblocking communication. Additionally, SyncChecker provides useful diagnostic information to pinpoint the root causes and help programmers to understand the bugs. Furthermore, our experiments with seven NAS Parallel Benchmarks show that SyncChecker incurs moderate overhead, ranging from 1.3 times to 9.5 times with an average of 5.2 times. This indicates that SyncChecker is suitable for programmers to detect synchronization errors in the testing phase.
Chapter 5: Bug Detection in MPI One-sided Application

In this chapter, we present a technique called MC-Checker to detect memory consistency errors in MPI one-sided applications. Memory consistency error occurs when two memory operations, which can be MPI one-sided calls such as MPI\texttt{Put} or local load/store from one or two processes, conflict with each other, leading the programs to an undefined state. To detect memory consistency errors in MPI one-sided applications, the main idea is to efficiently identify the execution epochs and concurrent execution regions, and to check the one-sided operations and/or local memory accesses within an epoch or within a concurrent region across processes. The checking is performed against the compatibility tables, which are constructed based on MPI specification [5]. If a pair of operations conflict, MC-Checker will report the error and provide diagnostic information to help programmers further locate and fix the bug.

Based on the above idea, we have implemented a prototype of MC-Checker in Linux and evaluated it with three real-world and two injected bugs in five MPI one-sided applications. Our experiments show that MC-Checker can effectively detect memory consistency errors in the evaluated MPI applications. In summary, MC-Checker has the following advantages:
• To the best of our knowledge, MC-Checker is the first comprehensive approach to address memory consistency errors in MPI one-sided communication. Our experimental results show that MC-Checker can effectively detect different types of memory consistency errors in various MPI one-sided applications. Furthermore, MC-Checker can provide useful diagnostic information to help programmers locate and fix the bugs.

• MC-Checker incurs low runtime overhead. Our experiments show that the average profiling runtime overhead incurred by MC-Checker is 11.1%. This indicates that MC-Checker is suitable for production runs. The reason of low runtime overhead is that MC-Checker exploits static analysis for instrumenting only relevant memory load/store accesses. This can significantly reduce runtime overhead.

• MC-Checker is easy to use and extensible. It leverages LLVM [48] techniques for automatic static analysis and instrumentation. No manual efforts are required. Although our current implementation focuses on the MPI applications written in C, we find no difficulties in extending MC-Checker to MPI applications written in other languages such as Fortran.

• The idea of MC-Checker can also be applied to other programming models with one-sided features such as ARMCI [62]. The memory models of most programming models with one-sided features are similar. Our approach can be easily applied to those programming models with little modification for memory consistency error detection.
Figure 5.1: Examples of memory consistency errors. The bold operations are conflicting operations without happens-before or consistency order.

### 5.1 Main Idea of MC-Checker

#### 5.1.1 Memory Consistency Errors

A memory consistency error in MPI is an erroneous program execution state with an undefined memory value based on MPI semantics [5]. It occurs when there are at least two conflicting operations that are potentially concurrent during program execution.

**Concurrency.** A pair of memory operations \( a \) and \( b \) (i.e., remote memory accesses or local memory accesses) in MPI programs are concurrent if there are no happens-before relation [46] and consistency ordering [5] between them. A happens-before relation between two operations \( a \) and \( b \) (i.e., \( a^h \rightarrow b \)) can be either the program order within one process, i.e., the previous instruction is executed before the later
instruction, or the synchronization order between different processes, e.g., `MPI_Ssend` at the source process completes before `MPI_Recv` at the corresponding destination process. A consistency order between two operations `a` and `b` (i.e., `a \rightarrow b`) guarantees that the memory effects of `a` are visible before `b`. This order is necessary because some synchronization actions (e.g., flush) order memory accesses without synchronizing processes. Another example is that if `a` is a non-blocking operation, `b` is the operation immediately following `a`, and both `a` and `b` access overlapping buffers, there is no consistency order between `a` and `b` due to `a`'s non-blocking nature.

**Conflicting Operations.** There are two types of conflicting operations. The first type is that all conflicting operations are within an epoch in a single process and they access local memory. An epoch is a program execution region that starts with a one-sided synchronization operation (e.g., `MPI_Win_fence` or `MPI_Win_lock`), and ends with a matching one-sided synchronization operation (e.g., `MPI_Win_fence` or `MPI_Win_unlock`). Epochs have a total order per process. Two memory operations `a` and `b` within an epoch are called conflicting if they are operating overlapping local memory locations and either: (1) the first one is a get operation, or (2) the first one is a put or accumulate operation and the second one is a get or local write operation.

The second type is that at least one of them is a remote operation that accesses a remote target memory location, i.e., conflicting operations across processes. For this type of conflicting operations, we need to consider two types of memory models in MPI one-sided communication: unified and separate memory models. In the unified memory model, two memory operations `a` and `b` are called conflicting if they are directly towards an overlapping memory locations in the same target process and either: (1) one of them is a put operation, (2) exactly one of them is an accumulate
operation, or (3) one is a get operation and the second one is a local write. In the separate memory model, in addition to the above three conflicting scenarios, any remote put and accumulate conflict with local write operations in the target process regardless of the access location. MPI 2.0 uses the separate memory model.

**Examples of Memory Consistency Errors.** Figure 5.1 shows the memory consistency errors caused by the above two types of conflicting operations. Figure 5.1 (a) shows an example of memory consistency errors within an epoch. **MPI_Put** sends data in *buf* from P0 to P1. After **MPI_Put**, the data in *buf* may or may not be sent out because the **MPI_Put** is nonblocking. However, the data may be corrupted by the store operation right after **MPI_Put**. Figure 5.1 (b) shows an example of memory consistency errors across processes for active communication mode. The **MPI_Put** operations in P0 and P2 are conflicting because they may access the window location in P1 concurrently which may lead to data corruption or undefined results during program execution. Similarly, Figure 5.1 (c) shows an example of memory consistency errors across processes for passive communication mode. Both Figure 5.1 (b) and (c) show memory consistency errors between the origin processes. Figure 5.1 (d) shows another example of consistency memory errors between the origin and target processes. In this example, the **MPI_Put** in the origin process conflicts with the store operation in the target process because they will write to the same buffer concurrently and may cause data corruption.
5.1.2 Bug Detection Process

The main idea of MC-Checker is to check whether there are conflicting operations that are not ordered by $h_b \rightarrow$ and $c_0 \rightarrow$. In particular, MC-Checker handles the two types of conflicting operations, i.e., within an epoch and across processes, as follows.

To detect conflicting operation within an epoch, MC-Checker first identifies the region of each epoch based on the matching synchronization calls within one process. Then MC-Checker checks operations within each epoch against the compatibility table. For the example in Figure 5.1 (a), MC-Checker identifies the epoch region as the instructions between two MPI Win_fence calls. Then MC-Checker checks all the statements inside the epoch and find conflicting operations.

It is more complicated to detect conflicting operations across processes. MC-Checker first matches all synchronization points across different processes. The synchronization points include all MPI calls that can synchronize processes, such as MPI collective calls and blocking sends/receives. Then MC-Checker identifies concurrent execution regions among processes via a graph-based approach. After that, MC-Checker will check operations against the compatibility table for each pair of concurrent execution regions. If a pair of conflicting operations is found, MC-Checker will report the memory consistency error and provide diagnostic information. For example, in Figure 5.1 (b), MC-Checker will identify the concurrent regions between P0 and P2. For this pair of concurrent regions, MC-Checker will detect the conflicting MPI_Put operations in P0 and P2 and report this error. Similarly, in Figure 5.1 (c), MC-Checker will identify the concurrent regions and detect the conflicting operations MPI_Put in P0 and MPI_Get in P1. It is worth noting that the passive mode of MPI one-sided communication requires other MPI calls such as MPI_Barrier to perform
synchronization. The conflicting operations in both 5.1 (b) and (c) happen between origin processes. It is also possible for conflicting operations to happen between the origin and target processes, such as the example in Figure 5.1 (d). In this example, MC-Checker will detect the conflicting operations of the \texttt{MPI\_Put} in P0 and the store operation in P1. After detecting the conflicting operations, MC-Checker will also provide diagnostic information such as pairs of conflicting operations and operation locations including file names, routine names and line numbers to help programmers locate and fix the bugs.

It is easy for developers to make mistakes and introduce the above memory consistency errors due to the complexity of MPI one-sided communication. For example, programmers may forget the nonblocking nature of an MPI communication call and overwrite the data before they are sent out. There could also be multiple \texttt{MPI\_Put}s from different remote processes that write to the same target window buffer. All of these mistakes will cause memory consistency errors that can lead to incorrect results, crashes, or hangs. MC-Checker detects these errors and provides useful diagnostic information to help locate and fix the bugs.

\section{5.2 Design and Implementation}

\subsection{5.2.1 Design Overview and Challenges}

MC-Checker consists of three components: ST-Analyzer, Profiler, and DN-Analyzer. As shown in Figure 5.2, ST-Analyzer performs static analysis of MPI applications at the source code level and generates a report that records relevant variable names that need to be monitored for their load/store instructions. Based on the result of ST-Analyzer, Profiler then instruments MPI applications before program execution
and logs relevant run time events to the trace files during program execution. DN-Analyzer analyzes the runtime traces and detect memory consistency errors. If it identifies a pair of conflicting operations that is not ordered, DN-Analyzer will report the error and provide important diagnostic information to help locate and fix the bug. Among the above three components, Profiler is an online component since the instrumented code is executed together with the MPI application while the other two are offline components.

There are three design challenges for the MC-Checker. We address them in Section 5.2.2, 5.2.3, and 5.2.4.

(1) How to profile runtime events efficiently? On one hand, the overhead incurred by Profiler should be as low as possible. If the overhead is too high, the tool can not be used even in the testing phase. On the other hand, Profiler should log as much information as possible to facilitate program correctness checking. Profiler should log all MPI calls, which are numerous due to MPI’s rich semantics. Furthermore, Profiler needs to log the load/store instructions because load/store instructions and MPI
one-sided communication function calls may conflict. But profiling every load/store instructions is prohibitively expensive for production runs since it can slow down the program execution by hundreds of times. So we need to devise some mechanisms to reduce Profiler’s runtime overhead without losing the detection capability of our tool.

(2) How to check memory consistency errors within an epoch? To check memory consistency errors with an epoch, MC-Checker needs to first identify the range of an epoch. There are different types of epochs in MPI one-sided communication, including active and passive communication modes. Second, there are many types of operations within an epoch such as MPI one-sided calls and load/store functions. MC-Checker needs to identify them and checks for potential conflicts among them. Third, MPI one-sided calls may involve different datatypes ranging from pre-defined contiguous datatypes to user-defined non-contiguous datatypes. MC-Checker should handle all of these complexities in order to detect memory consistency errors within an epoch.

(3) How to check memory consistency errors across processes? To check memory consistency errors across processes, MC-Checker first needs to identify concurrent regions of code among processes. There are many types of synchronization points that can form concurrent regions. Examples include collective MPI calls and blocking send/receive calls. It is difficult to match these synchronization points across processes. Second, it is challenging to represent these synchronization points. MC-Checker should find a good way to represent them in order to facilitate the detection of memory consistency errors. Third, MC-Checker should handle different types of operations in concurrent regions between processes for correctness checking.
5.2.2 ST-Analyzer: Identifying Relevant Memory Accesses

To perform thorough memory consistency checking, Profiler needs to instrument each memory access, leading to large runtime overhead. To address this issue, ST-Analyzer performs static analysis to identify relevant memory access instructions that need to be instrumented and logs them in a report file. As a result, Profiler only instruments these relevant memory access instructions to reduce the runtime overhead.

In order to identify relevant memory access instructions for detecting memory consistency errors, one straightforward approach is to analyze each load/store instruction, branch, loop, the scope of each variable, and so on. Such complete and sound analysis is expensive and slow since it requires context-sensitive and parameter-sensitive inter-procedure analysis and sound pointer alias analysis.

ST-Analyzer simplifies the analysis without losing the load/store instructions in which we are interested. First, ST-Analyzer identifies all variables that belong to the window buffers or the buffers being accessed by one-sided communication calls. It labels these variables as “relevant”. Then ST-Analyzer propagates such labels by following pointer assignments or function calls involving pointers. After that, ST-Analyzer records all labeled variables in the report. These variables are the ones we would like to instrument in the load/store instructions. Our design of ST-Analyzer is conservative since it is insensitive to branch and loop. In other words, ST-Analyzer may mark some variables that do not need to be instrumented in reality, but it will not fail to mark those that need to be instrumented. The current prototype of ST-Analyzer is implemented using the Clang front-end as a library in the LLVM [48] compiler framework.
5.2.3 Profiler: Collecting Runtime Information

To facilitate error checking, Profiler collects relevant runtime events and logs them in trace files. In particular, there are four types of MPI calls we need to collect during program execution. The first type is MPI one-sided calls, including MPI one-sided initialization calls, communication calls, and synchronization calls. Examples of this type include `MPI.Win_create`, `MPI.Put`, and `MPI.Win_fence`. These MPI calls are required because they are directly related to one-sided communication. The second type is MPI datatype manipulation routines such as `MPI_Type_struct`. These routines create new datatypes from existing primitive datatypes (e.g. `MPI.INT`) or other user-defined datatypes. They are needed because message data during one-sided communication may reside in a memory location specified by such derived datatypes. The third type is general synchronization calls. Examples include `MPI.Barrier`, `MPI.Bcast`, `MPI.Send`, and `MPI.Recv`. These synchronization calls should be instrumented because they may affect data availability for MPI communication and the derivation of concurrency among operations. The last type is MPI support routines such as `MPI.Comm_rank` and `MPI.Group_incl`. We need them to retrieve basic information for error detection.

In addition to instrumenting the above four types of MPI calls, Profiler needs to collect memory load/store accesses because they may conflict with one-sided communication operations. Profiler takes the report of ST-Analyzer as its input. The report contains the names of the variables and pointers that need to be instrumented for their memory accesses. With the identified variables, Profiler only logs their memory access information to the trace files.
For each instrumented function call, Profiler logs the function name and the arguments to the trace files. For each memory access, Profiler logs the access type (e.g. read or write), the address of the accessed memory, and the size of the accessed memory. Such information is sufficient to detect memory consistency errors for MPI one-sided communication.

We leverage techniques from the LLVM [48] compiler framework to implement the current prototype of Profiler. Specifically, the LLVM compiler front-end first transforms the MPI application to LLVM intermediate representation (IR). Then the Profiler pass is applied and transforms the original IR into the instrumented IR. After that, the code generation part of LLVM converts the instrumented IR into binary code.

5.2.4 DN-Analyzer: Analyzing Traces and Detecting Memory Consistency Errors

Pre-process trace files and extract information

Before checking memory consistency errors, DN-Analyzer pre-processes the trace files to retrieve some basic information. There are four types of such information: communicator, group, window buffer, and datatype.

Processing communicators and groups. An MPI program has a basic communicator MPI_COMM_WORLD. Additionally, an MPI program can use communicator/group manipulating routines to create user-defined communicators/groups based on previously-defined or basic communicators/groups. For example, the MPI call MPI_Group_incl creates a new group including part of the processes from an old group. The rank information in this function call is relative to the old group, not the basic group.
associated with \texttt{MPI\_COMM\_WORLD}. For convenience, DN-Analyzer transfers all relative rank information in the newly created communicators/groups to absolute rank information in the basic communicator. After that, DN-Analyzer stores all of the communicator/group information in a hash map so that it can be easily retrieved for future analysis.

\textbf{Processing window buffers.} A window buffer is created by the MPI call \texttt{MPI\_Win\_create}. After finishing the call, an MPI program generates a handle to represent this chunk of window buffer for one-sided communication. DN-Analyzer stores the handle of the window buffer in the hash map. As a result, when one-sided communication involves the window buffer, DN-Analyzer can retrieve its detailed information for error detection.

\textbf{Processing datatypes.} MPI datatypes are complex. They range from simple primitive contiguous datatypes such as \texttt{MPI\_INT} to complex user-defined non-contiguous datatypes. DN-Analyzer uses a \textit{data-map} structure to represent a datatype. A data-map consists of a series of segments. Each segment contains the displacement and the length of a contiguous chunk of the buffer. For example, \texttt{MPI\_INT} is represented as a data-map \{(0, 4)\}, where 0 is displacement and 4 is length. Similarly, a derived datatype containing two \texttt{MPI\_INT}s separated by an 8-byte gap is represented as \{(0, 4), (12, 4)\}. DN-Analyzer processes all datatype manipulating routines and stores all datatype information including primitive datatypes and derived datatypes in the hash map. To facilitate error detection, DN-Analyzer sorts the data-map segments based on their displacements in ascending order.
Table 5.1: The operation compatibility table within an epoch. The operations in the first row will be executed before the operations in the first column. BOTH means both overlapping and non-overlapping memory operations are permitted. NOVL means only non-overlapping operations are permitted, i.e., operations that access overlapping buffers conflict with each other.

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put/Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>BOTH</td>
</tr>
<tr>
<td>Store</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Get</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put/Acc</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>BOTH</td>
</tr>
</tbody>
</table>

**Detecting memory consistency errors within an epoch**

As Figure 5.1 (a) shows, MPI applications with one-sided communication can have conflicting operations within an epoch. This is because these one-sided operations are non-blocking. The data may not be moved in/out until the epoch ends. Based on the MPI specification [5], we derived the operation compatibility table for operations within an epoch, as shown in Table 5.1. A pair of conflicting operations means if such two operations within an epoch have overlapping buffers, they will lead to memory consistency errors.

**Identifying epoch regions.** To check this type of memory consistency errors, DN-Analyzer first identifies the epoch regions. An epoch region is delimited by a pair of one-sided synchronization calls, such as a pair of `MPI_Win_start` and `MPI_Win_complete` calls. The first routine is called the epoch start routine and the second is called the epoch end routine. DN-Analyzer performs the following analysis to identify an epoch region. For each epoch end routine, DN-Analyzer identifies the closest available
matching epoch start routine and then stores the matching information in both routine entries. DN-Analyzer performs the above analysis for all the epochs and stores the epoch information in a vector for each process.

**Checking conflicting operations.** After getting the epoch information, DN-Analyzer detects conflicting operations for the operations within each epoch. A simple and straightforward approach is to check in the following way. For each operation within an epoch, it scans through all other operations in the epoch and checks whether the second operation conflicts with the first chosen operation based on the compatibility table. However, this algorithm is inefficient since its time complexity is $O(n^2)$, where $n$ is the number of operations within an epoch.

Instead, DN-Analyzer performs the error detection in a more efficient way. It uses a vector to store the buffer information for each type of operations. One exception is that DN-Analyzer stores the buffer information for both MPI_Put and MPI_Accumulate operations in one vector because they access the local buffers in a similar way. For each operation within an epoch, DN-Analyzer only checks whether the vectors that store conflicting types of operations are empty or not. If yes, there are no previous operations conflicting with the operation currently under examination. Otherwise, all operations in the vectors potentially conflict with the current operation. After the step of vector checking, DN-Analyzer stores the buffer information of the current operation into the corresponding vector so that future operations can use such information. Let us use MPI_Put as an example. As shown in the last row of Table 5.1, it conflicts with MPI_Get operations. Then DN-Analyzer only checks its conflict with buffer information in the vector of MPI_Get. After that, it stores the local buffer information for MPI_Put in its corresponding vector. With the four vectors of buffer information
for four types of operations, DN-Analyzer scans the operations within an epoch only once and for each operation so that the checking time is constant (i.e., check whether the vectors are empty). Therefore, the time complexity of the algorithm deployed by DN-Analyzer is $O(n)$.

For each pair of potentially-conflicting operations reported by the above step, DN-Analyzer further identifies whether their buffers are overlapping. Buffers in one-sided communication can be specified by non-contiguous datatypes and contain multiple segments. If the displacements of the segments are unordered, the buffer overlap checking will be inefficient. During the pre-processing of the datatype, DN-Analyzer sorts the displacements in the data-map in ascending order. After that, DN-Analyzer performs an efficient interval intersection checking algorithm (i.e., with a constant time complexity) to see whether the pair of potentially-conflicting operations are true conflict operations. If yes, DN-Analyzer will report the memory consistency error with useful diagnostic information.

**Detecting memory consistency errors across processes**

As shown in Figure 5.1 (b), (c), and (d), MPI one-sided communication calls may cause memory consistency errors with other one-sided communication calls or load/store operations in another process. However, not every pair of operations will have such errors. The reason is that MPI applications have synchronization calls to enforce happens-before [46] and/or consistency ordering relations between two operations. For example, in Figure 5.3, operations $b$ (i.e., `MPI_Put` in process P2) and $c$ (i.e., `MPI_Get` in process P1) will not cause memory consistency errors since the barriers in P0, P1, and P2 make $b$ always happen before $c$ (i.e. $b \xrightarrow{hb} c$) and $b$ is consistent before $c$ (i.e., $b \xrightarrow{co} c$). Only two operations that fall into a concurrent
Figure 5.3: Memory consistency errors across processes. We abbreviate MPI_Barrier, MPI_Win_lock, MPI_Win_unlock, MPI_Put, and MPI_Get as Barrier, lock, unlock, Put, and Get, respectively.

execution region may cause memory consistency errors, such as $a$ and $b$, or $c$ and $d$ in this example. In this chapter, a concurrent program region is defined as a group of program regions across multiple processes that can be executed concurrently, without happens-before and consistency ordering relations. Since the concurrent regions are formed by MPI synchronization calls, DN-Analyzer only needs to identify matching MPI synchronization calls across different processes and detect conflicting operations in each concurrent region.

**Identifying concurrent regions.** To identify the concurrent regions, DN-Analyzer first locates matching MPI synchronization calls across multiple processes, which form happens-before and consistency ordering relations among processes. A concurrent
region is formed by the set of program regions in all processes that are not ordered by happens-before and consistency ordering relations.

MPI has different types of synchronization calls, including all collective calls and blocking send/receive. Matching all of these synchronization calls is a challenging task. A straightforward way works as follows. For each synchronization call, we scan through all of the traces in the corresponding processes and locate its matching synchronization calls. This algorithm is time consuming and error-prone, especially for large trace files. This is because MPI applications may use the same synchronization calls with same arguments for many times during program execution. It is not easy to determine which synchronization calls to match in other processes for a specific synchronization call in the current process.

DN-Analyzer uses a more efficient approach for matching synchronization calls. This approach tries to simulate the progress of real MPI processes. More specifically, DN-Analyzer maintains a vector of “progresses” to track the matching progress for each process. The progress for a process is the ratio of the number of matched entries over the number of all entries in the process. Each entry is a runtime event logged in the trace file for a process. At each step, DN-Analyzer selects a process with the minimum progress and starts the matching process for its first unmatched entry. If the entry is not a synchronization call, DN-Analyzer will skip it and update the progress of the current process. If the entry is a synchronization call, DN-Analyzer then retrieves the argument values of the synchronization call. Based on argument information, DN-Analyzer figures out other processes that have matching synchronization calls. For collective calls, DN-Analyzer can retrieve this information from the communicators.
Algorithm 3 Match synchronization calls

1: \( \text{minProgRank} \leftarrow 0 \)
2: \( \textbf{while} \ progress[\text{minProgRank}] < 1 \ \textbf{do} \)
3: \( \quad \text{entry} \leftarrow \text{getEntry}(progress[\text{minProgRank}]) \)
4: \( \quad \textbf{if} \ \text{entry} \ \text{is not synchronization call} \ \textbf{then} \)
5: \( \quad \quad \text{continue} \)
6: \( \quad \textbf{else} \)
7: \( \qquad \quad \text{match synchronization calls with other processes and store matching information} \)
8: \( \quad \textbf{end if} \)
9: \( \quad \text{update} \ progress[\text{minProgRank}] \)
10: \( \quad \text{minProgRank} \leftarrow \text{getMinProgRank}(progress) \)
11: \( \textbf{end while} \)

For blocking send/receive, the information can be fetched from send/receive arguments that specify the corresponding receive/send rank. After identifying the target processes of the matching synchronization calls, DN-Analyz er processes one process at a time. For each matching process, DN-Analyz er does not search from the beginning of the trace file since it is inefficient and unnecessary. Instead, DN-Analyz er locates the first unmatched entry from the recorded progress information and searches for the nearest matching synchronization calls from there. After the matching call is found, DN-Analyz er stores the matching information for both synchronization calls.

Algorithm 3 shows how DN-Analyz er identifies matching synchronization calls. Line 1 initializes the \( \text{minProgRank} \) to 0. The \textbf{while} loop in line 2 checks if the matching completes for a particular process. Inside the \textbf{while} loop, DN-Analyz er retrieves the first unmatched entry for the process in line 3. If it is not a synchronization call, it simply skips the entry in line 5. Otherwise, DN-Analyz er finds matching calls in other processes and stores the matching information in line 7. Then it updates the
progress for the current rank in line 9 and finds the next minimum progress rank in line 10.

DN-Analyzer needs to efficiently store the matching synchronization calls so that memory consistency checking can be performed easily. Specifically, DN-Analyzer leverages graph to represent the matching information. As shown in Figure 5.3, each process has a timeline. Each entry in the process is a node in the timeline. If two MPI calls match with each other, DN-Analyzer adds an edge between these two MPI calls. If they are collective calls, the edge is bidirectional. It means they will happen at the same time. If the calls are blocking send/receive, the edge will be unidirectional, pointing from the blocking send to the blocking receive. As in Figure 5.3, each shaded area actually has three bidirectional edges between P0 and P1, between P0 and P2, and between P1 and P2. We use shaded area for the purpose of clearer presentation. The edge information is stored with each synchronization call after the matching. For each non-synchronization call such as MPI\_Put, DN-Analyzer associates it with the nearest previous and next synchronization call position information to facilitate the detection of memory consistency errors.

**Detecting memory consistency errors.** After identifying the concurrent regions, DN-Analyzer detects possible memory consistency errors for each concurrent region. As discussed before, there are no memory consistency errors across two different concurrent regions since they are strictly ordered by both happens-before and consistency ordering relations. A straightforward method is that DN-Analyzer examines each pair of operations in a concurrent region against the compatibility table. The time complexity of this checking is combinatorial with respect to the total number of operations within one concurrent region. Can we do better than this?
Table 5.2: Operation compatibility table across processes. BOTH means both overlapping and non-overlapping operations are permitted. NOVL means only non-overlapping operations are permitted. X means combining these operations is erroneous.

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>BOTH</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Store</td>
<td>BOTH</td>
<td>BOTH</td>
<td>NOVL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Get</td>
<td>BOTH</td>
<td>NOVL</td>
<td>BOTH</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Acc</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>BOTH</td>
</tr>
</tbody>
</table>

By further understanding and analyzing MPI specification and programs, we observe that memory consistency errors across processes can only occur in the window buffers at target processes. There are two types of memory consistency errors across processes. The first type involves two operations, both of which are MPI one-sided communication calls, such as the example in Figure 5.1 (b) and (c). One condition for this type of errors is that they perform remote memory operations on the same target process. The second type involves one MPI one-sided communication call and one local load/store, such as the example in Figure 5.1 (d). The local load/store is performed in the target process of the one-sided communication calls.

Based on the above observation, we devise a more efficient error detection approach whose time complexity is linear in the total number of operations. In particular, we only need to examine the memory operations on the window buffer in the target process. This approach has two steps. DN-Analyzer first checks the memory consistency errors for all one-sided MPI calls that operate on the window buffers in the target processes. Additionally, it stores all one-sided MPI operations on the window buffers.
in the target processes for the next step. Second, DN-Analyzer checks whether the local memory operations conflict with the stored remote one-sided operations for each window buffer. The local operations include local load/store and the MPI calls that access a local buffer. We will discuss more technical details of this approach as follows.

For each concurrent region, DN-Analyzer uses a vector to store the information of the window buffers, one buffer per vector entry. Each entry stores the rank of the target process, the window location, and all previous one-sided operations on the window buffer. In the first step, DN-Analyzer scans the remote one-sided operations for each process. For each one-sided operation, DN-Analyzer retrieves its window buffer information from the argument. Based on the window buffer, DN-Analyzer checks the vector. If the vector has no entry for this window buffer, DN-Analyzer creates a new entry and stores relevant information (i.e., the rank of the target process, the window location, and all previous one-sided operations on this window buffer) about the window buffer into the vector. Otherwise, DN-Analyzer fetches all previous one-sided operations on the window buffer from the vector and checks them against the current one-sided operation via the compatibility table shown in Table 5.2. Once any conflict is found, DN-Analyzer reports the error and provide relevant diagnostic information. Then DN-Analyzer stores the current one-sided operation into the vector entry of the window buffer.

One optimization for this step is that DN-Analyzer organizes the one-sided operations on a window buffer by the type of the operations in the vector. More specifically, for each one-sided operation under examination, DN-Analyzer only stores the relevant information about this operation (e.g., source file name and line number) into the corresponding operation type of the window buffer in the vector. As a result, for
each one-sided operation in a process, DN-Analyzer only checks at most three times (one for each operation type) against the compatibility table shown in Table 5.2.

After checking the memory consistency errors for all one-sided calls, DN-Analyzer examines all local operations for each process to see whether they conflict with any previous remote one-sided calls to the overlapping window buffers stored in the vector. The local operations include the local load/store and all MPI calls performed to a local buffer. Since MPI\_Put and MPI\_Get access a local buffer, they can be treated as local load and store, respectively. In the second step, DN-Analyzer examines all local memory operations for each process. For each local memory operation, DN-Analyzer identifies the window buffers in this process that overlap with the local buffer accessed by this local memory operation. Buffer interleaving checking is simple since window buffers are sorted in ascending order in the vector. If no overlapping window buffers are found, DN-Analyzer continues processing the next local memory operation in this process. Otherwise, for each overlapping window buffer, DN-Analyzer checks whether the current local operation conflicts with any previously-performed one-sided operations. If yes, DN-Analyzer reports the error and provides diagnostic information.

It is worth noting that if a local memory operation under examination is a local store, we need to treat it differently. This is because a local store can not be combined with any MPI\_Put or MPI\_Accumulate even when they do not have any buffer overlap. This is required by the MPI specification 2.0. After that, DN-Analyzer checks whether there are buffer overlaps between the local buffer currently under examination and the window buffers for the process.
Table 5.3: Evaluated applications. Note that some bug IDs use the date of the applications and some use svn revision or version number.

<table>
<thead>
<tr>
<th>MPI Apps</th>
<th>Bug IDs</th>
<th>Bug Locations</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>emulate</td>
<td>04/2011</td>
<td>within an epoch</td>
<td>passive</td>
</tr>
<tr>
<td>BT-broadcast</td>
<td>06/2004</td>
<td>within an epoch</td>
<td>active</td>
</tr>
<tr>
<td>lockopts</td>
<td>r10308</td>
<td>across processes</td>
<td>passive</td>
</tr>
<tr>
<td>pingpong-inj</td>
<td>3.0.3</td>
<td>across processes</td>
<td>passive</td>
</tr>
<tr>
<td>jacobi-inj</td>
<td>09/2008</td>
<td>across processes</td>
<td>active</td>
</tr>
</tbody>
</table>

5.3 Evaluation Methodology

We perform our experiments on the Glenn cluster at the Ohio Supercomputer Center [65]. The cluster contains 658 computer nodes. Each node is a quad-core machine with 2.5 GHz Opteron CPU, 24 GB RAM, and 393 GB local disk space. The operating system running on the cluster is Linux 2.6.18. We use the cluster for online profiling. For analyzing the trace files, we use a computer with 2.67 GHz Intel Core i5 CPU, 4 GB RAM, and 1 TB hard drive. The online profiler is implemented using the LLVM compiler framework [48]. The offline analyzer is implemented as a single-threaded application using C++. It can be further improved using multi-threaded programming in our future work.

We evaluate the effectiveness of MC-Checker by using five real-world MPI one-sided applications as shown in Table 5.3, including (1) emulate, a program emulating distributed shared memory; (2) BT-broadcast [53], a binary tree broadcast algorithm using one-sided MPI discussed in a paper’s appendix; (3) lockopts, an RMA test case in the MPICH [6] library package; (4) ping-pong, a benchmark using ARMCI-MPI [30] in the MPICH library package; and (5) jacobi, an MPI one-sided implementation
Table 5.4: Overall effectiveness of MC-Checker

of the Jacobi method. The first three applications contain real-world bug cases of memory consistency errors within an epoch or across processes. To evaluate whether our tool can detect most of the buggy scenarios, we inject another two memory consistency errors in the last two applications, whose names end with “inj”.

We evaluate the runtime overhead of MC-Checker by using five benchmarks in the ARMCI-MPI package: bench groups, contiguous-bench, ping-pong, ring-flood, and strided-bench. These benchmarks are implemented by using ARMCI APIs, which are provided by ARMCI-MPI library. The ARMCI-MPI library is implemented by using MPI one-sided communication. To evaluate the runtime overhead, we run the benchmarks with two configurations, one with MC-Checker’s Profiler and one without MC-Checker’s Profiler. Each configuration is run for five times, in 64 processes. We also evaluate the scalability of MC-Checker’s Profiler on ping-pong benchmark in various number of processes ranging from 8 to 512.

5.4 Experimental Results

5.4.1 Overall Effectiveness

Table 5.4 shows the overall effectiveness of MC-Checker for detecting memory consistency errors in MPI one-sided applications. We measure its effectiveness by
examining whether MC-Checker can detect the bugs and whether MC-Checker can locate their root causes. In the table, we also list the bug information including error location, root cause and failure symptom. We record the number of processes involved in our experiments for each bug case.

MC-Checker is effective in detecting memory consistency errors in MPI one-sided applications. As shown in Table 5.4, MC-Checker is able to detect all of the evaluated three real-world and two injected bugs. Furthermore, MC-Checker pinpoints the root causes for all of these five bugs. Additionally, MC-Checker can detect the bugs residing in different error locations with root causes of different conflicting operations and failure symptoms. For example, MC-Checker detects the errors in emulate where the bug resides within an epoch. The bug is caused by conflicting \texttt{MPI\_Get} and local load/store operations. As another example, MC-Checker locates the root cause of the bug in lockopts where the bug occurs across two processes due to conflicting local load/store and remote \texttt{MPI\_Put/Get} operations. MC-Checker is effective in detecting memory consistency errors because it can accurately capture the timing of the MPI one-sided calls and the local load/store operations, then it identifies the conflicting operations.

Table 5.4 also shows that MC-Checker’s detection capability is not affected by the scale of the system. MC-Checker can detect the memory consistency errors occurring in a small scale such as emulate in two processes, as well as the errors occurring in a larger scale such as lockopts in 64 processes. The reason is that MC-Checker is a rule-based approach. It can accurately capture program semantics at run time and detect violation of specific program rules. In contrast, previous statistics-based
approaches [37, 57] can only detect bugs occurring in large scale because they need to collect a large amount of statistical data at run time.

The current prototype of MC-Checker is based on the rules of the one-sided model in MPI 2.0. However, we do not find any particular difficulties to support the rules of the one-sided model in MPI 3.0 or other one-sided models such as ARMCI [62]. After supporting additional programming rules, MC-Checker will also be able to detect memory consistency errors in these one-sided models.

To the best of our knowledge, MC-Checker is the first comprehensive approach to detect memory consistency errors in MPI one-sided applications. While Marmot [44] can detect some one-sided errors in MPI applications, it is limited to deadlock and parameter errors. A model checking approach [69] was proposed to detect deadlocks in MPI one-sided applications. However, it cannot handle memory consistency errors. A recently-proposed work, SyncChecker [26], can detect the errors occurring within an epoch. However, it is not able to detect memory consistency errors across processes.

5.4.2 Case Studies

This section presents two representative real-world bug cases evaluated in our experiments. One of them is from the algorithm in the appendix of a paper [53], while the other one is from the RMA test case in the MPICH package.

Case 1: Conflicting MPI_Get and load operations in BT-broadcast

This memory consistency error was found in a binary tree broadcast algorithm [53]. The error causes the program to execute a while loop forever. In our evaluation, we implement the algorithm in C code and run it in the cluster. Figure 5.4 shows the buggy code extracted from the algorithm. Line 1 and line 8 form a one-sided
1: MPI_Win_fence(0, win);
2: ...
3: check = 0;
4: while(check == 0){ // buggy load access
5:   MPI_Get(&check, 1, MPI_DOUBLE, ...);
6: }
7: ...
8: MPI_Win_fence(0, win);

Figure 5.4: Bug case 1: Conflicting MPI\_Get and load operations in BT-broadcast. Note that the load operation of check is conflicting starting from the second iteration of while loop.

communication epoch. Within the epoch, line 3 performs a store operation to initialize the local variable check, which is followed by a while loop between line 4 and line 6. The MPI\_Get will update the value of check from a remote process. However, it may not be completed until the end of the epoch at line 8 due to the nonblocking nature of one-sided communication. As a result, the program will execute the while loop forever as the value of variable check is always 0.

This error can be triggered by running BT-broadcast in two processes. After being applied to this program, MC-Checker reports that there is a local load operation conflicting with MPI\_Get. In addition, MC-Checker identifies the locations of these two conflicting operations, which are line 4 and line 5 as shown in Figure 5.4. With such detailed diagnostic information, programmers can easily locate the bug and fix it.
Case 2: Conflicting MPI_Put/Get and load/store operations in lockopts

This memory consistency error was found in the RMA test case in the MPICH package with svn revision number 10308. It can cause the program to yield non-deterministic results. Figure 5.5 illustrates the buggy scenario extracted from the source code. We only show the code snippet of part of the buggy areas due to the space limit. The double directional arrows show the synchronization points with happens-before and consistency ordering relations. We can see that the load and store operations in section A are within the concurrent region of the MPI_Put (abbreviated as Put in the figure) in section D. By checking with the compatibility table, the
load/store operation is conflicting with \texttt{MPI\_Put}. Thus the program will generate incorrect results. However, this bug is a little different from usual memory consistency errors. In particular, the lock in P0 is an exclusive lock. MC-Checker only reports a warning for this type of errors as there can be benign concurrent conflicting operations with exclusive locks. We need to rely on programmers to identify whether it is a real bug for this case.

This bug can be triggered by running lockopts in 64 processes. After being applied to this bug case, MC-Checker reports the conflicting put and load/store operations. As the lock in P0 is an exclusive lock, MC-Checker will only label this bug as a warning. Additionally, MC-Checker will pinpoint the location of the conflicting operations so that programmers can quickly identify whether it is a real bug.

From this bug case, we can also see that it is very easy to make mistakes when writing MPI one-sided applications. The RMA test case of MPICH should be written by MPI experts. They made this mistake in MPI one-sided code, not to mention people who are new to MPI one-sided communication.

### 5.4.3 Runtime Overhead

Figure 5.6 shows the execution time of five benchmarks without and with MC-Checker’s Profiler. The execution time for each benchmark is normalized to the native execution, i.e., the original program without MC-Checker. As shown in the figure, the runtime overhead incurred by MC-Checker’s Profiler is very low, ranging from 3.8 to 22.3% with an average of 11.1%.

The reason for the low runtime overhead is that MC-Checker only logs the necessary load/store and MPI function-level events. This is the benefit from static analysis.
Without static analysis, MC-Checker may cause hundreds of times overhead since it needs to instrument all memory load/store accesses, as demonstrated in previous studies such as SyncChecker [26] and Purify [40]. Although SyncChecker’s Profiler performs runtime optimizations, the average overhead is still 3.85 times. Thus static analysis greatly reduces runtime overhead by enabling Profiler to instrument relevant memory accesses and MPI function calls.

Figure 5.7 shows the scalability study of MC-Checker’s Profiler on pingpong benchmark. As shown in the figure, the runtime overhead decreases from 35.8% to 23.1% when the number of processes increases from 8 to 32. After that, the runtime overhead keeps between 20% and 25%. So the runtime overhead will not increase while the number of processes increases. The reason is that Profiler logs the runtime events into the local disk independently for each process.
Currently, MC-Checker performs analysis of traces offline. This section only discusses the runtime overhead incurred by online profiler. In the future, we can extend MC-Checker to perform online analysis by leveraging stream processing algorithms.

5.5 Summary

This chapter presents MC-Checker, a new method to detect memory consistency errors in MPI one-sided applications. Based on collected runtime events, MC-Checker checks the MPI one-sided calls and local load/store operations against the compatibility tables and see whether they have memory consistency errors. If any error is found, MC-Checker will report the bug with detailed diagnostic information.

We have built a prototype of MC-Checker on Linux. Our evaluation with three real-world and two injected bugs in five different MPI one-sided applications shows
that MC-Checker is effective to detect memory consistency errors. In addition, MC-Checker provides useful diagnostic information to help locate and fix the bugs. Furthermore, our experiments with five benchmarks shows that MC-Checker incurs low runtime overhead, ranging from 3.8 to 22.3% with an average of 11.1%. It indicates that MC-Checker can be applied to production runs without degrading performance substantially.
Chapter 6: Final Remarks

This dissertation proposes runtime support to improve the reliability of MPI applications and libraries. Our approach is to build software tools using system methods including instrumentation-based profiling and runtime analysis to help detect software bugs in MPI applications and libraries.

First, this dissertation proposes a method called FlowChecker to detect communication errors in MPI libraries. The main idea of FlowChecker is to extract program intentions of message passing and to check whether these intentions are fulfilled by the underlying MPI libraries, i.e., whether messages are delivered from the specified sources to the specified destinations.

Second, this dissertation proposes another approach called SyncChecker to detect synchronization errors between MPI applications and libraries. SyncChecker tracks relevant memory accesses in the MPI applications and corresponding message send/receive operations in the MPI libraries. Then it checks whether the correct execution order between the MPI application and the MPI library is enforced by the MPI completion check routines.

Finally, this dissertation proposes an approach called MC-Checker to detect memory consistency errors in MPI applications with one-sided communication. MC-Checker first performs online instrumentation and logs relevant runtime events such
as MPI one-sided calls and local load/store operations to the trace files. Then MC-Checker performs offline analysis and checks the operations against compatibility tables to see whether there are memory consistency errors.

We have designed and implemented software prototypes for the proposed methods and evaluated them with real-world and injected bugs on popular MPI libraries, including MPICH2, MVAPICH2, and Open MPI, and different MPI applications such as Athena and octopus. The experimental results show that the approaches proposed in this dissertation can effectively detect and locate bugs in MPI applications and libraries. They also provide useful diagnostic information to help programmers improve reliability of MPI applications and libraries. Furthermore, our tools incur low or moderate overhead which indicates that our tools are applicable for production runs or software testing phases.

In conclusion, system tools can greatly improve the reliability of MPI applications and libraries. Furthermore, they can enhance the availability and productivity of MPI programs.
Bibliography


