SyncChecker: Detecting Synchronization Errors between MPI Applications and Libraries

Zhezhe Chen† Xinyu Li† Jau-Yuan Chen† Hua Zhong* Feng Qin†
†Dept. of Computer Science and Engineering
The Ohio State University
{*chenzhe, lixiny, chenja, qin*}*@cse.ohio-state.edu*
*Technology Center of Software Engineering
Institute of Software, Chinese Academy of Sciences
zhongh@otcaix.iscas.ac.cn

***Abstract*—While improving the performance, nonblocking communication is prone to synchronization errors between MPI applications and the underlying MPI libraries. Such synchronization error occurs in the following way. After initiating nonblocking communication and performing overlapped computation, the MPI application reuses the message buffer *before* the MPI library completes the use of the same buffer, which may lead to sending out corrupted message data or reading undefined message data.**

This paper presents a new method called SyncChecker to detect synchronization errors in MPI nonblocking communication. To examine whether the use of message buffers is well synchronized between the MPI application and the MPI library, SyncChecker first tracks relevant memory accesses in the MPI application and corresponding message send/receive operations in the MPI library. Then it checks whether the correct execution order between the MPI application and the MPI library is enforced by the MPI completion check routines. If not, SyncChecker reports the error with diagnostic information. To reduce runtime overhead, we propose three dynamic optimizations.

We have implemented a prototype of SyncChecker on Linux and evaluated it with seven bug cases, i.e., five introduced by the original developers and two injected, in four different MPI applications. Our experiments show that SyncChecker detects all the evaluated synchronization errors and provides helpful diagnostic information. Moreover, our experiments with seven NAS Parallel Benchmarks demonstrate that SyncChecker incurs moderate runtime overhead, 1.3-9.5 times with an average of 5.2 times, making it suitable for software testing.

I. Introduction

A. Motivation

In the past decade the Message Passing Interface (MPI) [1] has been witnessed a great success. With the support of MPI libraries, programmers have developed many parallel applications across a variety of domains such as financial forecasting, bioinformatics, and astronomy [2]. The continuing trend of clusters as a major component in supercomputer environments [3] makes MPI the popular choice for writing high performance parallel programs.

One popular method for optimizing MPI applications is to overlap the computation of MPI applications and the communication of underlying MPI libraries since it can hide costly communication latency [4], [5], [6], [7]. Such overlapping can be achieved by invoking nonblocking MPI functions or RDMA (Remote Direct Memory Access) functions. For example, after issuing a nonblocking send `MPI_Isend`, the sender process will immediately return from the function call without waiting for the completion of the message sending. As a result, the sender process continues its own computation, while the communication can be concurrently performed by the MPI library via independent communication controllers, which are often equipped in modern high performance network cards [8], [9]. Similar computation-communication overlapping occurs when invoking MPI RDMA functions. Previous studies [10] have shown that nearly half of the MPI functions invoked in NAS Parallel Benchmarks [11] are nonblocking ones.

While improving performance, overlapping computation with communication is prone to synchronization errors between MPI applications (i.e., the computation part) and the underlying MPI library (i.e., the communication part). After overlapped computation, an MPI application often reuses the message buffer (e.g., for performance reasons) that has been passed to the MPI library for message transfer. Without proper synchronization, the MPI application and the MPI library may simultaneously access the message buffer and therefore could corrupt message data or receive undefined data, leading to severe program failures such as crashes, hangs, or incorrect results. According to a recent survey on the importance and severity of MPI errors [12], programmers have ranked such synchronization errors as No. 6 out of 21 different error types. To ensure correct synchronization, the MPI specification requires an MPI application to perform a completion check on the status of a nonblocking MPI call or a RDMA call before accessing the message buffer. Nevertheless, programmers, especially those who are used to the semantics of blocking communication, tend to forget the completion check or to access the message buffers before the completion check.

One particular challenge for detecting such synchronization errors is their non-determinism on manifestation. In other words, these synchronization errors may manifest as failures for some program runs but not for others, depending on whether the buffer accesses incorrectly issued by the MPI application occur *before* or *after* the message data being transferred by the MPI library. Such ordering is non-deterministic since message data transfer in MPI libraries [13], [14], [15], [16] can occur at arbitrary time depending on the runtime system status (e.g., the scheduling of network transmission and the input data of MPI calls) and/or different MPI implementations. Furthermore, for multi-threaded MPI libraries [17], [18], the degree of such non-determinism can significantly increase.

An existing tool, Umpire [19], cleverly exploits checksum of message data to detect synchronization errors in nonblock-

ing MPI communication. Specifically, Umpire intercepts each nonblocking MPI call and the corresponding completion check call, and then calculates the checksums of the data in the sending buffer at both MPI call interceptions, respectively. If the checksums mismatch, Umpire reports an error. Umpire can effectively detect the synchronization errors if the message data are corrupted. However, it relies on the invocation of the completion check to re-calculate the checksum. Nevertheless, based on our observation, such synchronization errors are often associated with omitted completion checks (See the real-world bug cases in Section VIII). Furthermore, Umpire's detection technique is inapplicable for nonblocking MPI receives. The reason is that order-violating read accesses in nonblocking receives will not corrupt the message data and therefore the checksums are intact. Note that Umpire is a general tool that can detect various types of MPI bugs besides the synchronization errors.

B. Our Approach

In this paper, 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. The specific implementation of SyncChecker we report in this paper focuses on nonblocking MPI calls such as `MPI_Isend` and `MPI_Irecv`. We do not see any particular difficulty in applying the core idea to handle RDMA calls since they also follow the semantics of nonblocking communication.

To detect such synchronization errors, SyncChecker performs online profiling on the relevant runtime events and then conducts on-the-fly analysis 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 invocation 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 the line numbers, the function names, and the source file names of the 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 types 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 the nonblocking calls while others are not. Second, memory access profiling can easily induce prohibitive runtime overhead as indicated by previous software instrumentation tools [20]. We address the two challenges in Sections IV and V, respectively.

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 [21], a grid-based application for astrophysical magnetohydrodynamics, (2) octopus [22], a simulator for electron-ion dynamics, (3) Boost-app, an application based on the Boost.MPI library [23], 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. 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 V) 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 [24], [25]. 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 [26], a lightweight dynamic binary instrumentation framework.

II. Related Works

Our work is related to previous studies in three categories: 1) synchronization bug detection for multi-threaded programs, 2) bug detection for parallel and distributed programs, and 3) problem diagnosis for large-scale systems.

Synchronization bug detection. Much research has been conducted on detecting synchronization bugs (e.g., data races [27], [28], [29], [30], [31], atomicity violations [32], [33], [34], and order violations [35], [36], [37]) in multi-threaded programs. These approaches can be classified into two categories: dynamic and static approaches. Dynamic approaches [20], [29], [32] 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 [20], although sampling [31] or new hardware [33], [36] can alleviate this situation. On the contrary, our approach exploits semantic-relevant dynamic optimizations to significantly reduce runtime overhead.

Static approaches [38], [39] analyze program source code and identify potential synchronization bugs based on programming axioms such as "unlock must be paired with lock". While incurring no runtime overhead, static methods typically report many false positives because of lacking accurate runtime information. This is also true for detecting synchronization errors in MPI nonblocking communication. For example, MPI programs may use aliased pointers to access the message buffers that are passed to nonblocking MPI calls. Without runtime information, it is very challenging for static methods to accurately detect the bugs.

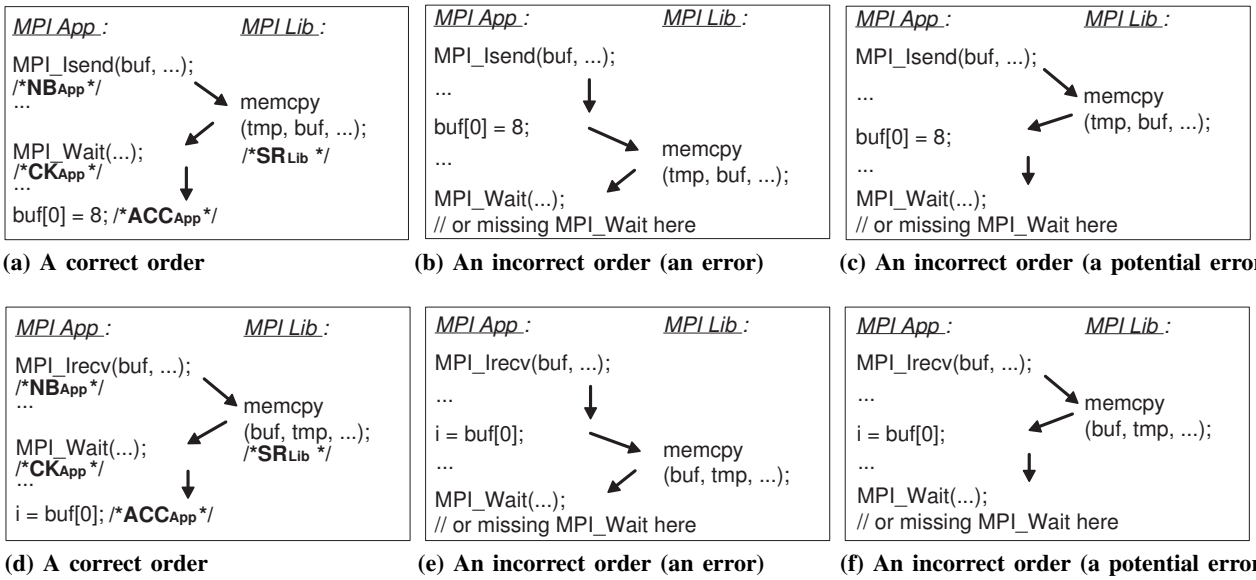
Bug detection for parallel and distributed programs. Realizing the importance of the reliability of parallel and distributed programs, researchers have proposed many dynamic techniques for interactive parallel debugging [40], [41], [42], [43], [44], [45], [46] and automatic bug detection [12], [19], [24], [47], [48], [49], [50]. Interactive parallel debuggers help programmers identify the bugs by exploiting automated information collection, aggregation, and visualization techniques. Unlike interactive debuggers, automatic bug detection approaches check program runtime behaviors, without manual intervention, against either specific programming rules (e.g., [19], [49], [50]) or extracted invariants based on temporal and/or spatial similarity (e.g., [24]). While these bug detection techniques can detect software bugs in MPI programs and/or MPI libraries, none of them except for Umpire [19] can detect synchronization errors in MPI nonblocking communication, which is the focus of our paper.

Problem diagnosis for large-scale systems. Research has been conducted on problem diagnosis for large-scale systems [25], [51], [52], [53], [54], [55], [56], [57], [58], [59]. 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]. Complementary to these approaches, SyncChecker leverages semantic information of MPI programs and libraries for detecting and diagnosing synchronization errors in nonblocking communication.

III. Main Idea

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 [60], 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 1 shows six scenarios with different execution orders of the relevant runtime events for nonblocking sends and receives. Specifically, Figure 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 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



(a) A correct order **(b) An incorrect order (an error)** **(c) An incorrect order (a potential error)**

(d) A correct order **(e) An incorrect order (an error)** **(f) An incorrect order (a potential error)**

Fig. 1. Six scenarios of the execution orders among the runtime events (at both MPI application and library levels) in nonblocking communication.

the message is copied out or sent out by the underlying MPI library. Figure 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 [19]. Similarly, Figure 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 VIII). 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 [19] cannot handle the cases where the completion check is missing since it recalculates message checksum at the completion check.

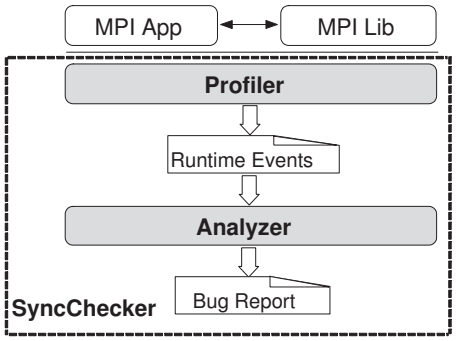


Fig. 2. Design overview of SyncChecker

IV. Design and Implementation

A. Design Overview and Challenges

SyncChecker is composed of two main components, including Profiler and Analyzer, as shown in Figure 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 IV and Section V, 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 `MPI_INT` to derived non-contiguous datatypes. Similarly, MPI nonblocking communication supports various completion checking semantics, including MPI blocking calls (e.g., `MPI_Wait`) and

MPI nonblocking calls (e.g., `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 [20], [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.

B. 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 `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`, and `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., `MPI_Type_struct`). These functions create new datatypes from primitive types (e.g., `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 V. 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 [24], [62].

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 [60] and recording the clock value for each profiled runtime event.

We leverage a lightweight dynamic binary instrumentation tool Pin [26] 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.

C. 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.

1) 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 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

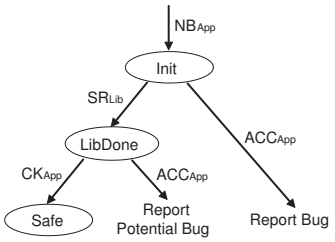


Fig. 3. The state machine for detecting synchronization errors

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 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.

2) 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 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 [24], [62]. Otherwise, Analyzer performs the state transition for the identified message buffer based on the error detection state machine in Figure 3.

V. Profiler Optimizations

The basic design of Profiler presented in Section IV 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 VIII), 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.

A. 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.

B. 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 paper 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.

C. 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

Algorithm 1 Profiler Optimizations

```
1: for each memory write access addr do
2:   if sendlist is non-empty then
3:     if addr ≥ sendlist.minaddr and
           addr ≤ sendlist.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 ≥ recvlist.minaddr and
           addr ≤ recvlist.maxaddr then
11:      Profiling the memory read access
12:    end if
13:  end if
14: end for
```

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.

D. Summary of the Optimizations

Algorithm 1 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 1, 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.

VI. 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.

MPI Apps	#LOC	Bug IDs	Bug Locations
Athena-r1086	89,549	#1093	nonblocking send
Athena-r1090	89,749	#1095	nonblocking send and receive
octopus-r1278	37,772	#1284	nonblocking send
Boost-app	133	10/2010	nonblocking send
Sort	156	07/2007	nonblocking receive
Athena-r1086-cc	89,549	#1093-cc	nonblocking send
octopus-r1278-cc	37,772	#1284-cc	nonblocking send

TABLE I
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.

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.

VII. Evaluation Methodology

Our experiments are conducted on two partitions of the Glenn cluster at Ohio Supercomputer Center [63]. One partition contains 877 computer nodes. Each node 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 [26], 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 I, including (1) Athena [21], a grid-based application for astrophysical magnetohydrodynamics; (2) octopus [22], a simulator for electron-ion dynamics; (3) Boost-app, an application using Boost.MPI, which is a C++-friendly interface to the standard MPI [23]; 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

MPI Apps	Languages	Bug IDs	Detected?		Missing Completion Check?	Error Locations	Failure Symptoms	# of Processes
			Umpire	SyncChecker				
Athena-r1086	C	#1093	No	Yes	Yes	send	program crash	64
Athena-r1090	C	#1095	No	Yes	Yes	send/receive	program crash	8
octopus-r1278	Fortran	#1284	No	Yes	Yes	send	program hang	64
Boost-app	C++	10/2010	No	Yes	Yes	send	incorrect results	8
Sort	C	07/2007	No	Yes	Yes	receive	program crash	8
Athena-r1086-cc	C	#1093-cc	Yes	Yes	No	send	program crash	64
octopus-r1278-cc	Fortran	#1284-cc	Yes	Yes	No	send	program hang	64

TABLE II
OVERALL EFFECTIVENESS OF SYNCCHECKER

errors with mislocated completion checks, i.e., the potential errors shown in Figure 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) [11] 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
- 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).

VIII. EXPERIMENTAL RESULTS

A. Overall Effectiveness

Table II 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 [19]. 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., nonblocking 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 II, 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 recalculating 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 II 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 [33], [64]. In contrast, previous statistics-based approaches [24], [25] 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 II, 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., [50]) can only deal with applications written in one programming language. The reason for SyncChecker's language independence is that it utilizes a binary instrumentation

```
1:  void bvals_mhd(DomainS *pD){
2:      ...
3:      ierr = MPI_Isend(send_buf[0], cnt, MPI_DOUBLE, ...);
4:      ...
5:      pack_ix1(pGrid);
6:      ...
7:  }

8:  static void pack_ix1(GridS *pG){
9:      ...
10:     double *pSnd = send_buf[0];
11:     ...
12:     *(pSnd++) = pG->U[k][j][i].d;          // buggy access
13:     ...
14: }
```

Fig. 4. Bug case: Sending buffer overwritten in Athena (written in C)

framework, Pin [26], to directly work on the binary code of the MPI applications and the MPI library for profiling and analyzing runtime events.

B. Case Studies

This subsection presents a representative case of synchronization errors from Athena written in C.

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 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. 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 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.

C. Runtime Overhead

Figure 5 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 5, 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.

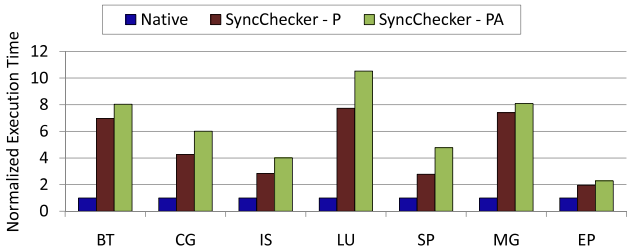


Fig. 5. Runtime overhead of SyncChecker. "Native" means execution without applying SyncChecker, "SyncChecker-P" means execution with Profiler only, "SyncChecker-PA" means execution with both Profiler and Analyzer enabled.

Additionally, Figure 5 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.

D. Effects of the Optimizations

Figure 6 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 7. 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 6, 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 7. For example, after applying ER, the number of profiled memory accesses per

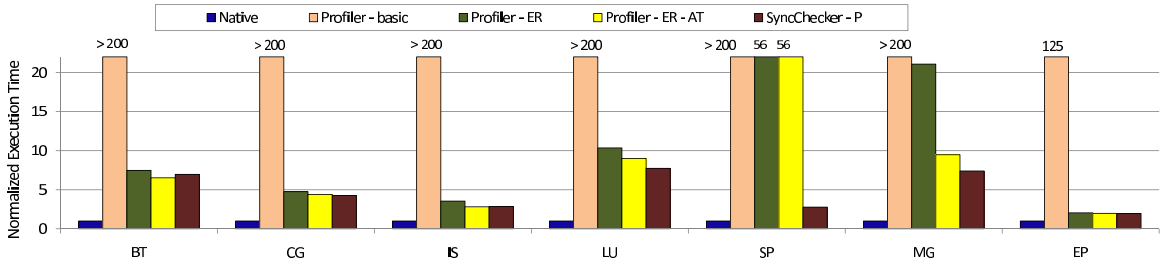


Fig. 6. Optimization effects of runtime overhead with Profiler enabled only. The runtime overhead of "> 200" incurred by Profiler-basic is because we terminate the program execution after that long time. "Native" means execution without applying SyncChecker, "Profiler-basic" means execution without any 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.

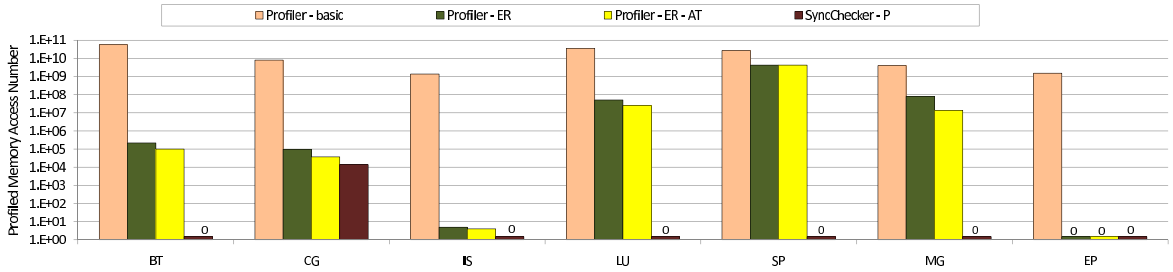


Fig. 7. The numbers of profiled memory accesses per process with different level of optimizations. For clarity, we put a tiny bar with "0" on the top indicating no memory access. Also note that y-axis is in a logarithmic scale. "Profiler-basic", "Profiler-ER", "Profiler-ER-AT", and "SyncChecker-P" have the same meanings as those in Figure 6.

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 6, 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 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 6. This is because the number of profiled memory accesses for SP is reduced from 4.28 billions to zero, as shown in Figure 7. 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.

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.

IX. Conclusions

This paper 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.

X. Acknowledgments

We thank the anonymous reviewers for invaluable feedback. We appreciate useful discussion with Yang Zhang, Mai Zheng, and Dachuan Huang. This work was supported in part by an allocation of computing time from the Ohio Supercomputer Center, and by the NSF grant #CCF-0953759 (CAREER Award). This work was also partially sponsored by the National Natural Science Foundation of China under Grant No. 61173004.

References

[1] "Message passing interface forum," http://www.mpi-forum.org.

[2] "Papers About MPI," http://www.mcs.anl.gov/research/projects/mpi/papers.

[3] "Architecture share in top 500 supercomputers for 11/2010," http://www.top500.org/stats/list/36/archtype.

[4] "MPI document of nonblocking communication," http://www.mpi-forum.org/docs/mpi22-report/node57.htm#Node57.

[5] C. Iancu, P. Husbands, and P. Hargrove, "Hunting the overlap," in *PACT*, 2005.

[6] S. Chakrabarti, M. Gupta, and J.-D. Choi, "Global communication analysis and optimization," in *PLDI*, 1996.

[7] Y. Zhu and L. J. Hendren, "Communication optimizations for parallel c programs," in *PLDI*, 1998.

[8] InfiniBand Trade Association, http://www.infinibandta.org.

[9] Myricom, http://www.myri.com.

[10] T. B. Tabe and Q. F. Stout, "The use of the mpi communication library in the NAS parallel benchmark," *Technical Report CSE-TR-386-99, University of Michigan*, 1999.

[11] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "NAS parallel benchmark results," in *Supercomputing*, 1992.

[12] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of mpi programs with intel message checker," in *SE-HPCS*, 2005.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *EuroPVM/MPI*, 2004.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, 1996.

[15] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," in *ICS*, 2003.

[16] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," *Lect. Notes Comput. Sci.*, vol. 2840/2003, 2003.

[17] E. Demaine, "A threads-only mpi implementation for the development of parallel programs," in *HPCS*, 1997.

[18] H. Tang, K. Shen, and T. Yang, "Compile/run-time support for threaded mpi execution on multiprogrammed shared memory machines," in *PPoPP*, 1999.

[19] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with Umpire," in *Supercomputing*, 2000.

[20] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[21] Athena, https://trac.princeton.edu/Athena.

[22] octopus, http://www.tddft.org/programs/octopus.

[23] D. Gregor, "Boost.MPI," http://www.boost.org/doc/libs/1_46_1/doc/html/mpi.html.

[24] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *Supercomputing*, 2007.

[25] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *Supercomputing*, 2006.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[27] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *PLDI*, 2002.

[28] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races," in *ISCA*, 2010.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, 1997.

[30] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *SOSP*, 2005.

[31] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: proportional detection of data races," in *PLDI*, 2010.

[32] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *POPL*, 2004, pp. 256–267.

[33] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006.

[34] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *PLDI*, 2005.

[35] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndstrike: toward manifesting hidden concurrency typestate bugs," in *ASPLOS*, 2011.

[36] B. Lucia and L. Ceze, "Finding concurrency bugs with context-aware communication graphs," in *MICRO*, 2009.

[37] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *ICSE*, 2010.

[38] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[39] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, 2002.

[40] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *Supercomputing*, 2009.

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

[42] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden, "Extending a traditional debugger to debug massively parallel applications," *J. Parallel Distrib. Comput.*, vol. 64, no. 5, 2004.

[43] Etnus, LLC., "TotalView," http://www.etnus.com/TotalView.

[44] S. S. Lumetta and D. E. Culler, "The mantis parallel debugger," in *SPDT*, 1996.

[45] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for mpi programs," in *Supercomputing*, 2010.

[46] F. Gioachin, G. Zheng, and L. V. Kalé, "Debugging large scale applications in a virtualized environment," in *LCPC*, 2011.

[47] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "A portable method for finding user errors in the usage of mpi collective operations," *IJHPCA*, vol. 21, no. 2, pp. 155–165, 2007.

[48] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for mpi deadlock detection," in *ICS*, 2009.

[49] B. Krammera, K. Bidmona, M. S. Mullera, and M. M. Rescha, "Marmot: An mpi analysis and checking tool," in *PARCO*, 2003.

[50] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: a tool for checking Fortran 90 MPI programs," *Concurr. Comput. Pract. Exp.*, vol. 15, 2003.

[51] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP*, 2003.

[52] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, "Automaded: Automata-based debugging for dissimilar parallel tasks," in *DSN*, Jun 2010.

[53] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *DSN*, 2002.

[54] K. L. Karavanic and B. P. Miller, "Improving online performance diagnosis by the use of historical performance data," in *Supercomputing*, 1999.

[55] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *TPDS*, vol. 21, pp. 174–187, 2010.

[56] N. Maruyama and S. Matsuoka, "Model-based fault localization in large-scale computing systems," in *IPDPS*, Apr 2008.

[57] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, 2006.

[58] L. Yang, C. Liu, J. M. Schopf, and I. Foster, "Anomaly detection and diagnosis in grid environments," in *SC*, 2007.

[59] S. Park, R. W. Vuduc, and M. J. Harrold, "extscFalcon: Fault localization for concurrent programs," in *Proc. ACM/IEEE Int'l. Conf. Software Eng.*, May 2010.

[60] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[61] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Winter USENIX Conference*, 1992.

[62] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in mpi libraries via message flow checking," in *Supercomputing*, 2010.

[63] Ohio Supercomputer Center, http://www.osc.edu.

[64] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatially detecting memory-related bugs via program counter-based invariants," in *MICRO*, 2004.