FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking

Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin
Department of Computer Science and Engineering
The Ohio State University
{*chenzhe, gaoq, zhangwen, qin*}*@cse.ohio-state.edu*

***Abstract*—Many MPI libraries have suffered from software bugs, which severely impact the productivity of a large number of users. This paper presents a new method called FlowChecker for detecting communication-related bugs in MPI libraries. The main idea is to extract program intentions of message passing (MP-intentions), and to check whether these MP-intentions are fulfilled correctly by the underlying MPI libraries, i.e., whether messages are delivered correctly from specified sources to specified destinations. If not, FlowChecker reports the bugs and provides diagnostic information.**

We have built a FlowChecker prototype on Linux and evaluated it with five real-world bug cases in three widely-used MPI libraries, including Open MPI, MPICH2, and MVAPICH2. Our experimental results show that FlowChecker effectively detects all five evaluated bug cases and provides useful diagnostic information. Additionally, our experiments with HPL and NPB show that FlowChecker incurs low runtime overhead (0.9-9.7% on three MPI libraries).

I. Introduction

A. Motivation

The Message Passing Interface (MPI) [2] is being widely used to develop parallel programs on computing systems such as clusters. This is evidenced by a plethora of MPI applications across many disciplines and activities, such as astronomy, bioinformatics, weather forecasting, and financial modeling [6]. As clusters continue to be a major component of High Performance Computing (HPC) environments [1], MPI is becoming increasingly prevalent.

Despite the success of MPI, many MPI library implementations [3], [4], [27], [51] have suffered from software bugs (also referred to as software defects). For example, more than 2000 bug tickets have been created for various versions of Open MPI [27] since 2006. Similarly, about 700 bug tickets have been reported for MPICH2 [3] since August 2008. The bugs in MPI libraries severely impact the productivity of a large number of users (users in this paper refer to MPI application developers). For example, these bugs have caused program crashes, hangs, or generation of incorrect results that are often hard to notice. Moreover, before reporting these bugs to library developers, users may have already spent days or weeks in vain trying to locate the root causes of the bugs in their own applications.

It is challenging and time-consuming for library developers to detect and locate software bugs in MPI libraries [5]. To locate such a bug that occurs at users' sites, library developers need to reproduce the bug at their own sites. This task is formidable due to platform differences (e.g., architectures and system scales) between users' and developers' sites. Often certain bugs only occur on large-scale systems [11]. Moreover, some MPI library bugs can only be triggered by real-world MPI applications. As a result, users need to share their MPI applications. If the applications are proprietary, users usually have to generate a small test program to trigger the bug, which could be a time-consuming process. Therefore, *it is imperative to devise low-overhead mechanisms for detecting MPI library bugs during production runs.*

Much research has been conducted to detect software bugs in HPC systems at run time. Among previous studies, many [21], [31], [33], [35], [54] focus on MPI applications. For example, Umpire [54] checks MPI function calls at run time against certain rules such as "all members of one process group must execute collective operations over the same communicator in the same order". Similarly, Intel Message Checker [21] traces MPI calls during execution and detects incorrect usage of MPI functions based on the traced events. These approaches are effective in detecting bugs in MPI applications. However, they cannot handle bugs in MPI library implementations since they assume the underlying MPI libraries are correct.

In recent years, researchers have explored temporal or spatial similarity exhibited in HPC systems for bug detection [28], [39]. The basic idea is to extract program runtime invariants [23], [29] within one process or across multiple processes, and to identify abnormal process behaviors for detecting software bugs. Examples of such program invariants include distribution of function execution time [39] and distribution of data movement chains [28]. While these statistics-based approaches may detect software bugs in MPI libraries, they cannot identify software bugs that occur in normal scenarios or bugs that only happen within a small number of processes. Furthermore, these methods may generate many false positives or false negatives due to the difficulty of setting thresholds to differentiate anomalies from invariants.

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

B. Our Approach

In this paper we present FlowChecker, a low-overhead method for detecting communication-related bugs in MPI libraries. A communication-related bug in this paper 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 paper, 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 execution, 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 [28]. 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.

It is challenging to perform the tasks above due to the complexity of MPI semantics. First, representing MP-intentions is intricate because of the variety of MPI communication patterns, ranging from point-to-point communications to collective communications, from contiguous datatypes to non-contiguous datatypes. Similarly, blocking and non-blocking MPI calls impose another challenge for identifying initial data movements that are relevant to each MP-intention. Furthermore, using optimized communication algorithms (e.g., using intermediate nodes for relaying messages in `MPI_Gather`) largely complicates message flow tracking. We address these challenges in Section III.

Based on the above ideas, we have implemented a prototype of FlowChecker on Linux. We have evaluated FlowChecker with five real-world bug cases from three popular MPI libraries, including Open MPI [27], MPICH2 [3], and MVAPICH2 [4]. 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 five real-world 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) [7] and NAS Parallel Benchmarks (NPB) [13] 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 algorithm 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 [36], 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.

II. Main Idea of FlowChecker

A. 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 `MPI_Send` and `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 [28]. Examples of 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.

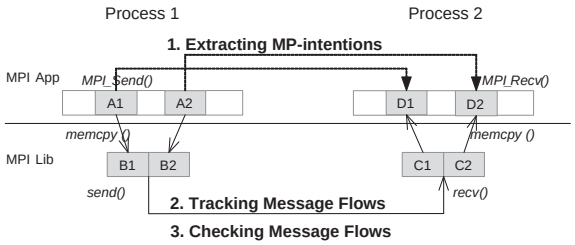


Fig. 1. An example to illustrate the main idea of FlowChecker. The MP-intention here is {A1→D1, A2→D2} and the message flows are A1→B1→C1→D1 and A2→B2→C2→D2.

Figure 1 illustrates the main idea of FlowChecker. In this example, process 1 invokes `MPI_Send` to send a message stored at the buffer {`A1`, `A2`} to process 2, while process 2 invokes `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 1, FlowChecker first extracts the MP-intention {`A1`→`D1`,`A2`→`D2`} based on the matched `MPI_Send` and `MPI_Recv` at process 1 and process 2, respectively. Then FlowChecker tracks the message flows, `A1`→`B1`→`C1`→`D1` and `A2`→`B2`→`C2`→`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.

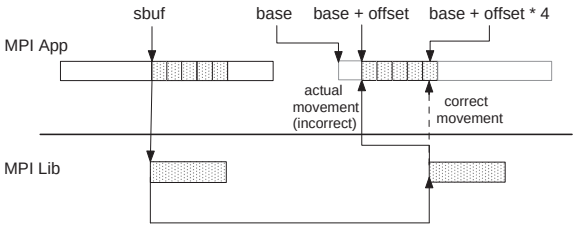
B. A Real-World Bug Case

Figure 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 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

```
int mod_mca_coll_self_gatherv_intra(void *sbuf, int scount, ...)
        ....  // code that calculates the offset
1       sdtype = rdtype = MPI_INT;
2       dst = ((char *) base) + offset;
                        /* BUG here: should be (offset * 4) */
3       return ompi_ddt_sndrcv(sbuf, scount, sdtype,
                               dst, rcount, rdtype);
    }
```

(a) the simplified source code



(b) the message flow in the MPI library

Fig. 2. A simplified bug case from an MPI library

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 [28], [39].

III. FlowChecker Design and Implementation

A. Design Overview and Challenges

FlowChecker consists of four major components: Profiler, MP-Extractor, MF-Tracker, and MF-Checker. As shown in Figure 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 III-B, III-C, and III-D, 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

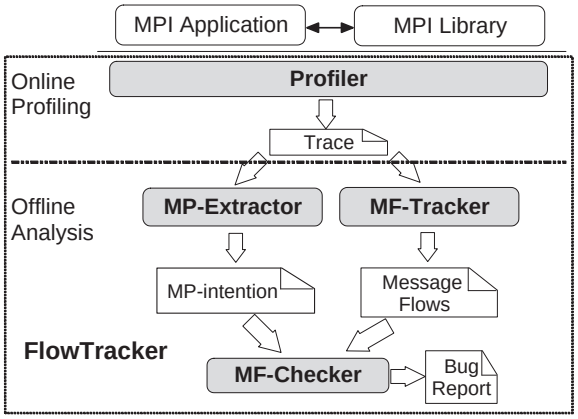


Fig. 3. FlowChecker design overview

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.

B. Profiler: Collecting Communication Traces

Profiler collects communication and message flow events into trace files during program execution. More specifically, Profiler logs MPI function calls involved by MPI applications as well as data movement operations made by MPI libraries. While 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: a) communication routines, e.g., `MPI_Send`, `MPI_Gather`; b) datatype manipulation routines, e.g., `MPI_Type_contiguous`, `MPI_Type_struct`; and c) basic supporting routines, e.g., `MPI_Comm_rank`, `MPI_Comm_size`. Additionally, Profiler logs the following data movement operations: memory copy routines such as `memcpy` within one process and network communication routines such as `writev` and `readv` across multiple processes.

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 VI-D). 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 [45], node-local storage [16], and ScalaTrace [44].

The current implementation of Profiler uses Pin [36], 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 recompilation of MPI libraries and MPI applications.

C. 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. Then it matches corresponding MPI calls from different processes by following MPI standards, e.g., using the `tag`s 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_Gather`s in different processes. This matching method was also used by previous work on detecting bugs in MPI applications [33], [49]. 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_Gather`s in N processes, there are N MPI call-pairs, each with two `MPI_Gather`s that transfer message data from a process to the root process.

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

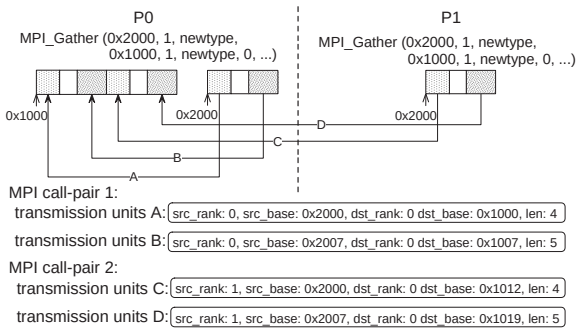


Fig. 4. An example of an MP-intention. The MP-intention is implied by a group of `MPI_Gather`s with non-contiguous data in two processes. MP-Extractor represents the MP-intention as two MPI call-pairs, each with two transmission units.

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 4 shows the representation of a group of corresponding `MPI_Gather`s with non-contiguous data in two processes.

D. 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 <`rank`, `base`, `len`>, 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 involves 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 preprocesses 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 blocking MPI calls, the MPI standard requires them not to return until the message data in the sending buffer are consumed (i.e., either stored in another buffer or sent over the network). Therefore, MF-Tracker limits its search scope within the period between the invocation of the blocking MPI call and its return. Note that this rule also applies to all collective MPI calls (e.g., `MPI_Bcast` and `MPI_Gather`) since they all follow blocking semantics.

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 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

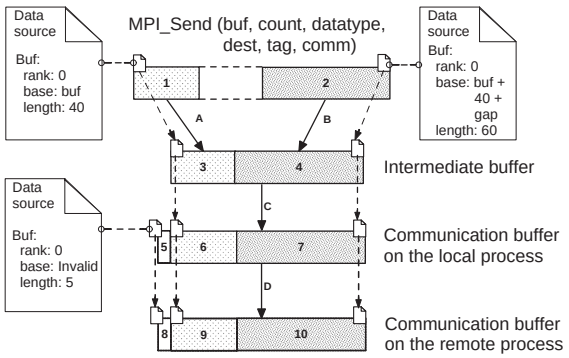


Fig. 5. An example of data source propagation for two transmission units. A, B, C, and D are four data movement operations. Buffers 1, 2, 3, 4, 6, 7, 9, and 10 store message data specified by the MPI application, while buffers 5 and 8 store control data used by the underlying MPI library.

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 5, buffer 5 and buffer 8 are labeled as "invalid" data sources in the data movement D.

E. 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.

There are two types of mismatches between the 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. To help diagnose the bug, MF-Checker provides the failed MPI call-pairs, the failed transmission unit, all data movement operations of the message flow, and the broken point. 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, in addition to the diagnostic information provided for the first type, MF-Checker traces back through the message flow and reports the first data movement with a mismatch. The diagnostic information provided by FlowChecker can help programmers quickly pinpoint the root causes (more details in Section VI).

IV. Issues and Discussion

Application-level bugs: When the MPI applications themselves have bugs, e.g., the buffer of `MPI_Isend` is used in the subsequent `MPI_Send` before the corresponding `MPI_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 [21], [33], [35], [54] can be applied together with or before FlowChecker.

Data movements via value assignments: Although MPI libraries usually use the standard C library functions such as `memcpy` and `memmove` for data movements within a process, there are cases where data movements are implemented by value assignments. For example, `MPI_Reduce` requires calculation on the data being transmitted. After calculating the data values, MPI libraries can directly assign them to the destination buffer. To address this issue, FlowChecker can leverage static analysis techniques [40] to identify such value assignments and transform them as data movements.

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 [41] and Quadrics [47]. 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 [2]. The nondeterministic constructs such as `MPI_ANY_SOURCE` are not supported. To address this issue, we can modify the MPI libraries to pass the information of sending processes to FlowChecker when returning the relevant MPI calls. Advanced features such as one-sided communication and MPI-I/O, are not supported either. 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.

V. 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

MPI Libs	Versions	LOC	Network Interfaces	Bug IDs	Bug Desc.
Open MPI	1.2	410,151	Gigabit Ethernet	#209	Datatype construction error
				#689	Packing/unpacking mismatch
				#1157	Incorrect pointer offset
MPICH2	1.0.6	447,093	Gigabit Ethernet	#280	Incorrect memory copy direction
MVAPICH2	0.9.8	480,310	InfiniBand	03/2007	Incorrect sending buffer size

TABLE I
MPI LIBRARIES AND BUG CASES USED IN THE EVALUATION. LIBS MEANS LIBRARIES, LOC MEANS LINES OF CODE, AND DESC. MEANS DESCRIPTION.

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 [36], 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 bug cases from three widely-used MPI library implementations, including Open MPI [27], MPICH2 [3], and MVAPICH2 [4] [1]. We choose these bugs since they represent different types of mistakes, including a datatype construction error, packing/unpacking mismatch, incorrect pointer offset, incorrect memory copy direction, and incorrect sending buffer size. Table I shows the five 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 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 [7] with the input sizes of 30,000, 40,000, and 50,000, and six NAS Parallel Benchmarks [13] 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. 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.

[1]These bug cases do not imply the reliability of the MPI library implementations by any means.

VI. EXPERIMENTAL RESULTS

A. Overall Results

Table II 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 II shows that FlowChecker detects all five evaluated bugs in the three MPI library implementations. Additionally, FlowChecker locates the exact functions that contain the bug for four out of five 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 II 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 [58]. On the contrary, previous statistics-based approaches [28], [39] 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 II 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 profiles data movement related functions and MPI function calls. Section VI-D 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

MPI Libs	Bug IDs	Detected?	Pinpoint Root Causes?	# of Processes	Trace Size		Execution Time (sec.)	
					Total	Per Process	Profiling	Analyzing
Open MPI-1.2	#209	Yes	No	64	31MB	496KB	7.99	4.71
	#689	Yes	Yes	64	31MB	496KB	7.24	4.35
	#1157	Yes	Yes	1	484KB	484KB	5.49	0.02
MPICH2-1.0.6	#280	Yes	Yes	64	13MB	208KB	2.15	3.39
MVAPICH2-0.9.8	03/2007	Yes	Yes	64	25MB	400KB	7.70	4.32

TABLE II
OVERALL RESULTS OF FLOWCHECKER FOR BUG DETECTION.

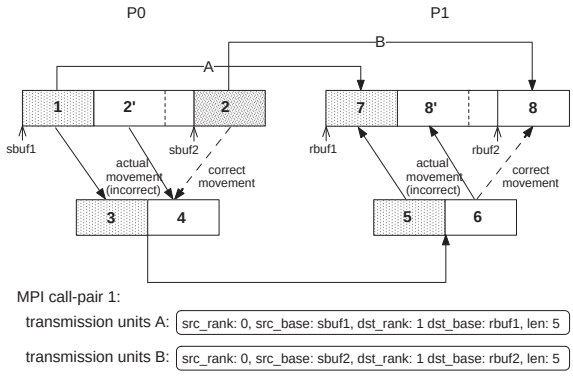


Fig. 6. Bug case 1: Datatype construction error in Open MPI

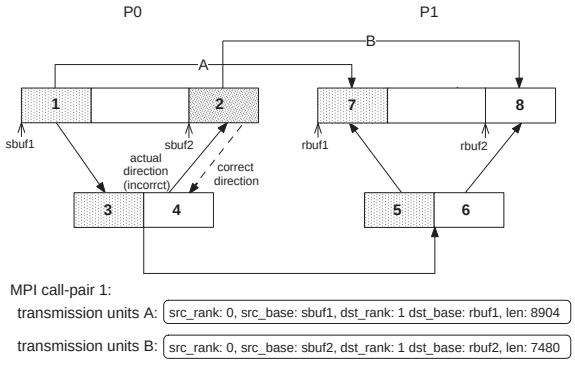


Fig. 7. Bug case 2: Incorrect memory copy direction in MPICH2

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.

B. Case Studies for Effectiveness of FlowChecker

This subsection presents two representative bug cases, one from Open MPI and the other from MPICH2.

1) 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.

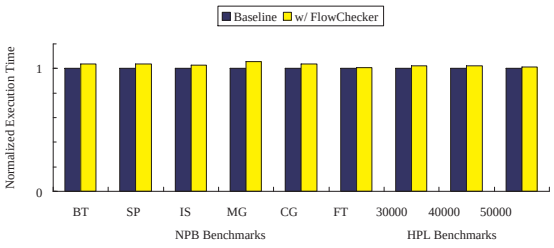
The driving MPI application contains one bug-triggering input (i.e., one pair of `MPI_Send` and `MPI_Recv` with a non-contiguous datatype), mixed with diversified normal inputs. After being applied to this case, FlowChecker detects this bug and reports one unfulfilled MP-intention that consists of the exact pair of `MPI_Send` and `MPI_Recv` that triggers the bug. As shown in Figure 6, the transmission unit B (buffer 2 $\rightarrow$ 8) fails due to incorrect message flow in the MPI library.

Among the five 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 6, 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.

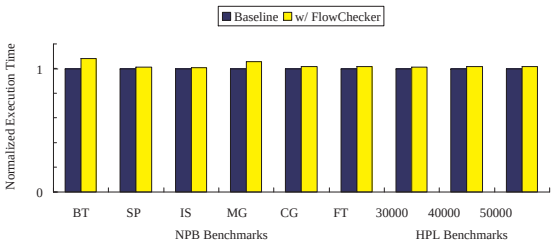
Note that this bug cannot be detected by prior statistics-based approaches [39], [28]. The reason is that this bug neither causes any observable change in the execution time of each function nor skews the data movement chain distribution.

2) 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.

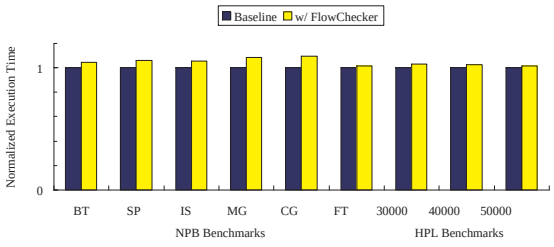
After applied to this case, FlowChecker detects the bug and reports that one out of 16,064 MP-intentions is not fulfilled by the corresponding message flows. In particular, the failed MP-intention is a group of `MPI_Gatherv` calls in 64 processes, which are equivalent to 64 MPI call-pairs. Among them, 24 call-pairs are not fulfilled due to failed transmission units. As shown in Figure 7, the transmission unit B (buffer 2 $\rightarrow$ 8) fails due to incorrect message flow in the MPI library. These failed transmission units are caused precisely by the bug.



(a) runtime overhead on Open MPI



(b) runtime overhead on MPICH2



(c) runtime overhead on MVAPICH2

Fig. 8. Runtime overhead of FlowChecker. Baseline is the native execution without applying FlowChecker.

Furthermore, FlowChecker pinpoints the root cause of this bug. As shown in Figure 7, 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.

C. Runtime Overhead

Figure 8 (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.

D. Disk Space Overhead

Table III 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 III 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.

VII. Related Work

FlowChecker is related to many previous studies. This section only describes the most relevant work in four categories: 1) bug detection for parallel and distributed programs, 2) communication error detection, 3) general software bug detection, and 4) problem diagnosis in large systems.

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 [21], [26], [28], [31], [33], [35], [54], which are partially discussed in previous sections. In addition to these dynamic approaches, researchers have explored formal verification and model checking methods [49], [55] for detecting bugs such as deadlocks in parallel and distributed programs. Moreover, interactive parallel debuggers [9], [11], [15], [24], [37] leverage automated information collection, aggregation, and visualization to help programmers manually locate root causes of software bugs in parallel programs. In summary, these studies focus on detecting software bugs at the application level. Complementary to them, FlowChecker focuses on detecting software bugs in MPI libraries.

Communication error detection. Another closely related method is the checksum feature provided in several MPI

Benchmarks		Open MPI			MPICH2			MVAPICH2		
		T-Size (MB)	E-Time (min)	G-Rate (MB/min)	T-Size (MB)	E-Time (min)	G-Rate (MB/min)	T-Size (MB)	E-Time (min)	G-Rate (MB/min)
NPB	BT	2.14	1.88	1.14	1.55	1.45	1.07	5.82	1.74	3.34
	SP	3.70	1.24	2.98	2.91	1.27	2.29	11.36	1.08	10.52
	IS	0.42	0.10	4.20	0.07	0.05	1.4	0.53	0.03	17.67
	MG	0.97	0.09	10.78	0.50	0.09	5.56	1.36	0.07	19.43
	CG	3.73	0.75	4.97	2.75	0.74	3.72	8.66	0.29	29.86
	FT	0.45	0.54	0.83	0.08	0.66	0.12	0.92	0.37	2.49
HPL	30000	2.59	2.31	1.12	2.05	2.29	0.90	6.52	1.74	3.75
	40000	3.30	5.05	0.65	2.69	4.91	0.55	9.09	4.12	2.21
	50000	4.00	9.15	0.44	3.33	9.16	0.36	11.66	7.80	1.49

TABLE III
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.

libraries [4], [12], [27], 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, which only provides limited help in finding the root cause. Furthermore, the checksum feature can significantly slow down the communication since it needs to performance computation on the data content, e.g., generating a CRC code [46].

Unlike the checksum feature, FlowChecker only profiles and analyzes data communication operations, not the data being communicated, and directly compares these operations with extracted MP-intentions. It allows FlowChecker to detect software bugs efficiently and effectively in MPI libraries. From this perspective, FlowChecker is complementary to the checksum feature while incurring low runtime overhead. In addition to report a bug, FlowChecker provides detailed diagnostic information because it analyzes each step in data communication and can identify the faulty steps and relevant functions.

General software bug detection. There are many research studies on detecting software bugs in general systems. Examples include program assertions [10], [32], static analysis methods [14], [20], [22], [25], dynamic checking approaches [30], [43], [48], model checking [56], and formal verification [42], [52]. 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 [53] and better bug reporting and understanding [18], [50]. These studies can be used to improve our work.

Problem diagnosis in large systems. Problem diagnosis in large systems mainly focuses on isolating the root causes of system failures or performance problems. Most existing studies [8], [19], [34], [38], [39], [57] utilize machine learning or statistical methods to study error propagation or identify program anomalies. These methods provide useful hints on diagnosing system problems. For example, Maruyama and Matsuoka propose comparing function traces from normal runs to those of failed runs for fault localization [38]. Carrozza et al. leverage Support Vector Machine classifiers to detect and locate software faults in complex safety-critical software systems [17]. Unlike these approaches, FlowChecker exploits program semantics and message flows to detect and diagnose software bugs in MPI libraries.

VIII. CONCLUSIONS

In this paper, 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 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 four out of five 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.

IX. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for useful feedback. We thank Dr. Pavan Balaji and Dr. Darius Buntinas for helpful discussions on MPICH2 and related bug cases. We are also grateful to Dr. Umit Catalyurek for providing the BMI cluster. We appreciate useful discussions with Adam C. Champion, Dacong Yan, and Mai Zheng. This research is supported in part by NSF grants #CCF-0953759 (CAREER Award) and #CNS-0403342.

REFERENCES

[1] Architecture share in top 500 supercomputers for 06/2009. http://www.top500.org/stats/list/33/archtype.

[2] Message Passing Interface Forum. http://www.mpi-forum.org.

[3] MPICH2: A high-performance and widely portable implementation of the Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpich2.

[4] MVAPICH2: MPI-2 over OpenFabrics-IB, OpenFabrics-iWARP, PSM, uDAPL and TCP/IP. http://mvapich.cse.ohio-state.edu/overview/mvapich2.

[5] Open MPI Bug Tickets. https://svn.open-mpi.org/trac/ompi/ticket/689.

[6] Papers about MPI. http://www.mcs.anl.gov/research/projects/mpi/papers.

[7] A. Petitet and R. C. Whaley and J. Dongarra and A. Cleary. High Performance Linpack. http://www.netlib.org/benchmark/hpl.

[8] 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.

[9] 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.

[10] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6), 1999.

[11] 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.

[12] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. W. Sukalski, and M. A. Taylor. Network fault tolerance in LA-MPI. In *EuroPVM/MPI*, 2003.

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

[14] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[15] 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.*, 64(5), 2004.

[16] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[17] G. Carrozza, D. Cotroneo, and S. Russo. Software faults diagnosis in complex OTS based safety critical systems. In *EDCC*, 2008.

[18] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, 2008.

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

[20] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI*, 2003.

[21] 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.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[23] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

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

[25] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *FSE*, 1994.

[26] C. Falzone, A. Chan, E. Lusk, and W. Gropp. A portable method for finding user errors in the usage of MPI collective operations. *International Journal of High Performance Computing Applications*, 21(2), 2007.

[27] 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.

[28] 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.

[29] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

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

[31] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A graph based approach for MPI deadlock detection. In *ICS*, 2009.

[32] T. Knauth, C. Fetzer, and P. Felber. Assertion-driven development: Assessing the quality of contracts using meta-mutations. In *ICST*, 2009.

[33] B. Krammera, K. Bidmona, M. S. Muller, and M. M. Rescha. MARMOT: An MPI analysis and checking tool. In *PARCO*, 2003.

[34] Z. Lan, Z. Zheng, and Y. Li. Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(2), 2010.

[35] 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.*, 15(2), 2003.

[36] 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.

[37] S. S. Lumetta and D. E. Culler. The mantis parallel debugger. In *SPDT*, 1996.

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

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

[40] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[41] Myricom. http://www.myri.com.

[42] V. S. S. Nair, K. Indiradevi, and A. J. A. Formal checking of reliable user interfaces. In *Proceedings of IEEE International Conference on Fault-Tolerant Systems*, 1995.

[43] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[44] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8), 2009.

[45] R. Oldfield, A. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for scientific applications. Technical Report SAND2006-3057, Sandia National Laboratories, May 2006.

[46] A. Perez. Byte-wise CRC calculations. *IEEE Micro*, 3(3), 1983.

[47] Quadrics. http://www.Quadrics.com.

[48] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), 1997.

[49] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[50] M. Song and E. Tilevich. Enhancing source-level programming tools with an awareness of transparent program transformations. In *OOPSLA*, 2009.

[51] J. M. Squyres and A. Lumsdaine. A component architecture for LAM/MPI. In *EuroPVM/MPI*, 2003.

[52] N. Suri and P. Sinha. On the use of formal techniques for validation. In *FTCS*, 1998.

[53] M. Skraut, S. Weigert, U. Schiffel, T. Knauth, M. Nowack, D. B. de Brum, and C. Fetzer. Speculation for parallelizing runtime checks. In *SSS*, 2009.

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

[55] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. In *PPoPP*, 2009.

[56] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI*, 2004.

[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.*, 40(4), 2006.

[58] 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.