MPIRace: A Static Data Race Detector for MPI Programs

Wenwen Wang

University of Georgia

Abstract. Data races in distributed parallel programs, such as those developed with the message passing interface (MPI), can cause critical correctness and reliability issues. Therefore, it is highly necessary to detect and fix them. However, existing MPI programming error detection tools have rather limited support for data race detection. To address this problem, we present MPIRace, which is a *static* data race detector for MPI programs. It creates several novel and effective static program analysis techniques to overcome the technical challenges of conducting static data race detection for MPI programs. We also implement a research prototype of MPIRace based on LLVM, a widely-used compiler infrastructure. After applying MPIRace to MPI-CorrBench, a recent MPI correctness benchmark suite, and a broad range of real-world MPI applications, we successfully find 20 data races. Among them, 12 are found in the real-world MPI applications and it is the first time they are reported by a data race detector. Moreover, the detection speed of MPIRace is extremely fast, i.e., less than one minute for every evaluated application. We believe MPIRace will tremendously help developers in improving the correctness, reliability, and sustainability of MPI programs.

1 Introduction

The message passing interface (MPI) [18] is the industry-standard programming model of distributed parallel programs. However, even with MPI, developing highly-efficient yet *correct* distributed parallel programs is still not an easy task. This is because developers have to carefully coordinate the communications between different computer nodes. A subtle programming mistake can potentially lead to incorrect execution results. *Data races* are a common source of programming errors in MPI programs. Typically, a data race occurs when two *concurrent* memory accesses read/write the *same* memory location without any synchronization preserving their *happens-before* order, and at least one of the two accesses is *write*. As a consequence, the nondeterministic happens-before order of the two accesses can produce unexpected or incorrect execution results. Therefore, it is highly necessary to detect and fix data races in MPI programs.

Unfortunately, existing MPI programming error detection tools have very limited support for data race detection. Some static detection tools simply omit data races due to the inherent complexity of data race detection [3,24]. For example, MPI-Checker [3] is a static correctness checker to verify the usage of

MPI library routines. It only detects very simple MPI programming errors, such as MPI data type mismatches, invalid argument types, and unmatched/missing MPI calls. Some other detectors need to monitor the execution of the target MPI program to detect MPI programming errors [6,13,25,29,33,34] or synchronization errors [1,2,5,7]. For example, MUST [6] intercepts MPI library calls to examine call arguments for MPI programming error detection. But it has no support for data race detection. Traditional data race detectors, such as ThreadSanitizer [26] and Helgrind [16], are mainly developed for programs written in shared-memory programming models, e.g., OpenMP and Pthreads [31], and therefore, it is hard to apply them directly to distributed programs written in MPI.

To address the above limitations, this paper presents **MPIRace**, which is a *static* data race detector specifically designed for MPI programs. MPIRace augments existing MPI programming error detectors with the key capability of detecting data races. Further, compared to dynamic data race detectors, MPIRace does not need to run target MPI programs with sample inputs. That means it does not have the notorious coverage problem. MPIRace creates novel and effective static *compiler-based* program analysis techniques to analyze the entire code base of the target MPI program for data race detection.

We have implemented a research prototype of MPIRace based on LLVM [15]. Specifically, MPIRace takes as input the LLVM intermediate representation (IR) of the target MPI program and reports data races in the program. To evaluate MPIRace, we apply it to MPI-CorrBench [12], a recent MPI correctness benchmark suite, and a broad range of real-world MPI applications, including Ember [9], miniFE [21], Presta [10], SuperLU_DIST [14], and U.S. Naval MPI Tutorials [11]. MPIRace successfully reports 20 data races in these applications. Among them, 12 are found in real-world applications, and it is the first time these data races are discovered by a data race detector. This demonstrates the effectiveness and practicability of MPIRace. We believe MPIRace will provide strong support for developing correct, reliable, and sustainable MPI programs. The source code of MPIRace is available at `https://github.com/mpirace/mpirace`.

In summary, this paper makes the following contributions:

- We present MPIRace, which is, to the best of our knowledge, the first-ever static data race detector for MPI programs. MPIRace vastly strengthens the capability of existing MPI programming error detection tools.
- We propose novel and effective static program analysis techniques to overcome the technical challenges of MPIRace. We anticipate other MPI-related programming/debugging tools can also benefit from these techniques.
- We implement a research prototype based on LLVM, a popular compiler infrastructure, to demonstrate the feasibility of MPIRace. We also properly address several practical problems during the implementation process.
- We evaluate MPIRace using both microbenchmarks in MPI-CorrBench and real-world MPI applications. The results show that MPIRace can successfully discover previously unknown data races in these applications.

2 Background and Motivation

Data Races. In general, a data race is triggered by three conditions. First, a shared memory location is accessed *concurrently*. Second, the two accesses are *not ordered* by any synchronization. Finally, at least one access is a *write* operation. Since the *happens-before* order of the two shared memory accesses in a data race is *nondeterministic* [35], data races often lead to unexpected execution results [32]. In this paper, we use `op(start_address, size)` to represent a shared memory access. `op` denotes the *type* of the access, i.e., `R` or `W`. `start_address` and `size` are the two *operands* of the access, indicating the start address of the accessed memory location and the size of the access (in bytes), respectively.

Message Passing Interface (MPI). MPI [18] is a standardized and portable interface for programming parallel and distributed computers. It defines the syntax and semantics of a set of library routines that can be used to build highly efficient and scalable distributed parallel programs. Though MPI is not shipped with an official implementation, there are several popular MPI implementations maintained by independent organizations, such as MPICH [20], Open MPI [19], Microsoft MPI [17], and IBM Spectrum MPI [8].

In MPI, a group of processes that can communicate with one another is called a *communicator*. Each process in a communicator is assigned with a unique *rank*, which can be used to communicate with other processes in the same communicator. The communications are implemented by passing *messages*. For example, a rank can send/receive a message to/from another rank by invoking the `MPI_Send()`/`MPI_Recv()` routine and passing the receiver/sender rank to the routine. An MPI routine is usually *blocked* until the corresponding communication is completed. Nevertheless, in practice, communications between different nodes are very expensive. Therefore, in addition to blocking routines, MPI also supports *nonblocking* routines, e.g., `MPI_Isend()`/`MPI_Irecv()`, which can return immediately without waiting for the completion of the communications. Through nonblocking routines, an MPI program can hide the long latency of communications by overlapping them with local computations.

Motivation. Although MPI dramatically simplifies distributed parallel programming, it is still quite challenging for developers to write highly efficient yet correct MPI programs. One major obstacle is data races. Although MPI programs are implemented using processes and, in general, it is unlikely to form data races between different processes, there are still many scenarios in an MPI program in which *concurrent* shared memory accesses can be issued. For instance, the program code and the MPI library may be executed in parallel because of nonblocking MPI library routines. This can generate numerous concurrent shared memory accesses and therefore, potentially introduce data races.

Figure 1 shows an example of MPI data race. In this example, there are two ranks, and rank 0 needs to send the variable `data` to rank 1. To achieve this, rank 0 invokes the MPI library routine `MPI_Isend()` at line 9. This routine will read the value of `data` and send it to rank 1. Here, the first, second, third, fourth, and last arguments of `MPI_Isend()` mean the start address of the data to be sent,

```
1  MPI_Comm comm;
2  MPI_Request req;
3  int rank, data = 0;
4  comm = MPI_COMM_WORLD;
5  MPI_Comm_rank(comm, &rank);
6  if (rank == 0) {
7    data = 1;
8    // Send data to rank 1
9    MPI_Isend(&data, 1, MPI_INT, 1, ..., &req);
10   data = 2; // Conduct computations on data
11   MPI_Wait(&req, ...);
12 } else if (rank == 1) {
13   // Receive data from rank 0
14   MPI_Recv(&data, 1, MPI_INT, 0, ...);
15   if (data != 1)
16     printf("ERROR: data is wrong!\n");
17 }
```

Fig. 1: An MPI data race, caused by the accesses to `data` at line 9 and line 10.

the number of the data elements to be sent, the type of each data element, the receiver rank, and the handle of the send request, respectively. We omit other arguments to facilitate the discussion. However, `MPI_Isend()` is a nonblocking routine. That means, when the routine returns, the send operation may have not been completed. In other words, it is very likely that the MPI library reads `data` and sends it out after the routine returns. As a result, this read operation can form a data race with the following write operation to `data` at line 10, due to their nondeterministic happens-before order. If the write operation happens before the read operation, the value 2 will be sent to rank 1. This will eventually lead to an error in rank 1. As we can see in this example, data races in MPI programs can produce unexpected results and disrupt program execution. Therefore, it is of paramount importance to detect and fix them.

3 Technical Challenges

In this section, we explain the technical challenges and briefly describe our proposed solutions in MPIRace. It is worth pointing out that these challenges are *unique* to MPI programs, and therefore, not addressed by previous research work.

Process-based Parallel Execution Model. The first challenge is caused by the parallel execution model of MPI. Different from thread-based shared-memory programs, such as those written in OpenMP and Pthreads, MPI programs are often implemented using *processes*. This renders it challenging to determine whether two memory accesses in an MPI program are from the same address space. In particular, if they are issued by different ranks, they are less likely

to form a data race. The reason is that such memory accesses actually access different memory locations in different address spaces, even if the name of the accessed variable is the same. Recall the example in Figure 1. The variable `data` is accessed by both rank 0 and rank 1. But such accesses actually access different memory locations in different address spaces. For instance, the write to `data` at line 10, issued by rank 0, and the read from `data` at line 15, issued by rank 1, are from different address spaces and thus not shared memory accesses.

A straightforward solution for this challenge is to analyze all memory accesses and classify them based on the ranks by which they are issued. However, this will slow down the entire detection process and make it unscalable for large-scale MPI programs. Instead, to overcome this challenge, our key observation is that it is actually unnecessary to analyze all memory accesses in an MPI program for data race detection. This is because data races in MPI programs usually only happen in specific parallel regions. In this paper, we call such regions *may-have-races* (MHR) parallel regions. By identifying MHR regions, we can detect all potential data races in an MPI program. More importantly, it allows us to narrow down the analysis scope and boost the analysis efficiency. We will describe more details about how to identify MHR parallel regions in the next section.

Shared Memory Accesses from MPI Library. In general, to statically detect data races in a program, we need to collect memory accesses in the program and analyze those accessing the same memory location. However, the fact that most data races in MPI programs involve shared memory accesses coming from the MPI library makes this static analysis process quite challenging. Again, take the data race in Figure 1 as an example. One of the two shared memory accesses of the data race resides in the MPI routine `MPI_Isend()`. That means, if we only analyze memory accesses in the target MPI program, we will not be able to detect this data race, as line 9 is a function call not a memory access.

An intuitive solution for this problem is to also analyze memory accesses in the MPI library. However, this solution is impractical because of two reasons. First, the MPI library may contain massive memory accesses and most of them are irrelevant to our data race detection problem. Note that detecting data races in the MPI library is out of the scope of this paper. Second, even though the source code of many MPI implementations is available, it is still possible that the target MPI program uses a closed-source MPI implementation. In that case, only the executable code of the MPI library is available, which makes it extremely hard to analyze memory accesses in the library.

To address this problem, we propose a novel *memory access modeling* technique in MPIRace. This is inspired by the observation that the semantics of MPI library routines are well defined in the MPI standard [4]. Therefore, we can model the relevant memory access behavior of an MPI routine based on its semantics specified in the standard. For example, `MPI_Isend()` in Figure 1 can be modeled as a read operation from the variable `data`. This way, we can exclude irrelevant memory accesses in the MPI library without loss of the capability to detect data races in MPI programs.

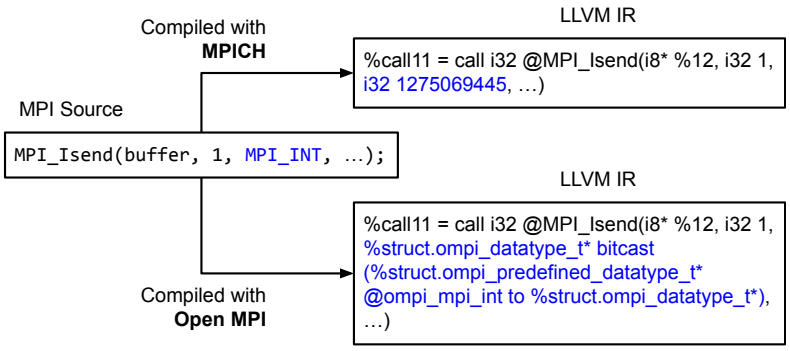

Fig. 2: The implementation divergence of `MPI_INT` leads to different LLVM IRs.

Implementation Divergences of MPI Library. The last but certainly not the least technical challenge is the divergences between different MPI implementations. In particular, the same MPI data type can be implemented in completely different ways, which makes it challenging to accurately identify MPI data types. Figure 2 shows an example. Here, we use LLVM [15] to compile the *same* MPI program with two representative MPI implementations: MPICH [20] and Open MPI [19]. As we can see, the compiled LLVM intermediate representations (IRs) are *not* the same due to different implementations of `MPI_INT`. Specifically, MPICH implements `MPI_INT` as a 32-bit integer, while Open MPI implements it as a sophisticated type. As a result, we cannot simply use the same approach to identify MPI data types compiled with different implementations.

To address this issue, our observation is that for the same MPI implementation, different MPI data types are generally implemented in the same way. For example, MPICH implements different MPI data types as different integers. This allows MPIRace to identify MPI data types in the same implementation using the same approach, e.g., checking the values of the integers. To support different MPI implementations, MPIRace further creates a *type identification table*, each entry of which describes what identification approach should be used for a specific MPI implementation. Note that this table can be created even without the source code of the MPI implementation. Also, MPIRace only needs to look up this table at the beginning of the analysis process, as an MPI program is typically compiled with only one MPI implementation.

4 MPIRace

In this section, we first present a high-level overview of MPIRace, and then dive into the technical details.

Figure 3 shows the workflow of MPIRace. As shown in the figure, MPIRace takes as input the LLVM IR of the target MPI program. The first step of MPIRace is to identify may-have-races (MHR) parallel regions. Typically, there

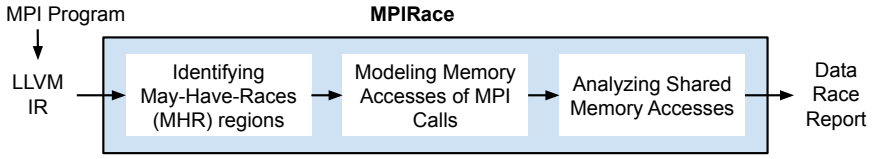

Fig. 3: The high-level workflow of MPIRace.

are two types of MHR parallel regions: *intra-rank* and *inter-rank*. In an intra-rank MHR parallel region, the MPI program code of a rank can be executed in parallel with the MPI library code, or two MPI library routines are executed concurrently. The parallel execution may produce shared memory accesses to a variable defined in the rank. Differently, an inter-rank MHR parallel region involves two ranks and its execution may generate shared memory accesses to a variable defined in one of the ranks, because another rank may remotely access the variable through MPI library routines. To detect both types of MHR parallel regions, MPIRace first analyzes every MPI library call in the target MPI program to figure out whether it is a start point of an MHR parallel region, based on the semantic of the called MPI library routine. If yes, MPIRace further analyzes the following code to find the end point(s) of the region. We will describe more details in the following subsection.

After MHR parallel regions are identified, MPIRace next collects shared memory accesses in each region and analyze them for data race detection. There are two major sources of shared memory accesses: the target MPI program and the MPI library. For shared memory accesses in the target program, MPIRace can collect them by analyzing the input LLVM IR. However, for shared memory accesses from the MPI library, it will incur heavy analysis overhead if we simply analyze the MPI library code. To avoid this problem, MPIRace models shared memory access behaviors of MPI library calls based on their semantics. More specifically, MPIRace creates *artificial* shared memory accesses to emulate the shared memory access behavior of each MPI library call. By analyzing these artificial accesses, MPIRace is able to decouple the analysis from the implementation details of the MPI library and scale up the detection process.

The final step of MPIRace is to analyze shared memory accesses in each MHR parallel region to detect data races. To this end, MPIRace checks whether two shared memory accesses may *concurrently* access the *same* memory location. If yes, they may form a data race. It is quite common in MPI programs that a memory access only touches specific elements in a memory buffer, e.g., odd- or even-indexed elements: `buf[i]`/`buf[i+1]`. That means, two shared memory accesses may not access the same memory location even if they access the same memory buffer. Besides, the memory location that is accessed by a memory access can also be influenced by the control flow, e.g., different memory addresses are assigned to the same pointer on different program paths. Hence, it is essential to thoroughly analyze shared memory accesses for accurate data race detection.

Algorithm 1: Identification of MHR Parallel Regions

Input: *LList* - The LLVM IR instruction list of a function
Output: *RSet* - The set of detected MHR parallel regions

```
1  RSet ← ∅
2  for I ∈ LList do
3  |  if I is not an MPI call then
4  |  |  continue
5  |  end
6  |  if I does not start an MHR parallel region then
7  |  |  continue
8  |  end
9  |  ESet ← ∅
10 |  for J ∈ LList do
11 |  |  if J is not reachable from I or not an MPI call then
12 |  |  |  continue
13 |  |  end
14 |  |  if J may end the MHR parallel region of I then
15 |  |  |  ESet ← ESet ∪ {J}
16 |  |  end
17 |  end
18 |  RSet ← RSet ∪ {< I, ESet >}
19 end
20 return RSet
```

To this end, MPIRace conducts *field-sensitive* and *path-sensitive* shared memory access analysis to limit potential false positives and false negatives.

Once a data race is detected, MPIRace will report detailed information about the data race to facilitate the following manual investigation and debugging activities. This includes the two shared memory accesses that form the data race and the memory location accessed by them.

4.1 Identifying MHR Parallel Regions

To identify MHR parallel regions, MPIRace analyzes every MPI library call in the target MPI program to see whether it may start an MHR region. If yes, MPIRace continues to analyze the following code to find the end of the region.

Algorithm 1 explains how MPIRace identifies MHR parallel regions. As shown in the algorithm, MPIRace conducts the identification at the function granularity. More specifically, MPIRace scans the LLVM IR instructions of the input function to find all MPI calls in the function. For each MPI call, MPIRace checks whether it starts an MHR parallel region by examining the semantic of the called MPI library routine. For example, an MPI call to a nonblocking library routine, e.g., `MPI_Isend()`, can start an intra-rank MHR parallel region, which allows the following program code to be executed in parallel with the called library routine for enhanced performance and scalability. Note that the

Table 1: Sample MPI routines denoting the start/end of an MHR parallel region.

		Sample MPI Library Routines
intra-rank	Start	`MPI_Isend()`, `MPI_Irecv()`, `MPI_Ireduce()`, `MPI_Put()`, `MPI_Get()`, ...
	End	`MPI_Wait()`, `MPI_Test()`, `MPI_Waitany()`, `MPI_Win_fence()`, ...
inter-rank	Start	`MPI_Win_fence()`, `MPI_Win_start()`, `MPI_Win_lock()`, ...
	End	`MPI_Win_fence()`, `MPI_Win_complete()`, `MPI_Win_unlock()`, ...

semantics of each MPI library routine is clearly defined in the MPI standard [4]. Table 1 lists some sample MPI library routines that can start an MHR parallel region. By identifying MPI calls to such routines, MPIRace is able to detect the start of an MHR parallel region.

If an MPI call I is recognized as the start of an MHR parallel region, MPIRace next tries to find the MPI call(s) J that can serve as the end of the region. This will delimit the scope of the following shared memory access analysis for data race detection. In general, there are three conditions for J. First, it needs to be reachable from I on the control-flow graph. This is easy to understand because J should be executed after I. Second, it should have the semantic of ending an MHR parallel region. Table 1 shows some sample MPI library routines that have such semantics. Finally, the arguments passed to it can match with those passed to I. This is important, as unmatched arguments often imply different MPI communications. For example, in Figure 1, the first argument of `MPI_Wait()` is the same as the last argument of `MPI_Isend()`, i.e., `&req`, which denotes the handle of the communication request. Therefore, they correspond to the same MPI communication. Based on these three conditions, MPIRace is able to find MPI call(s) that end an identified MHR parallel region.

4.2 Modeling Memory Accesses of MPI Library Calls

MPIRace models the shared memory access behavior of an MPI library call mainly based on the semantics of the called MPI routine. In particular, if an MPI routine reads from (or writes to) a memory buffer passed to it, MPIRace will create an artificial read (or write) operation to model the access. In other words, the artificial read (or write) operation is *semantically equivalent* to the MPI call from the perspective of data race detection.

Similarly, we use `op(start_address, size)` to represent an artificial memory access operation, where `op` is the access type, i.e., $\mathtt{R_B}$, $\mathtt{R_N}$, $\mathtt{W_B}$, or $\mathtt{W_N}$, and `B` and `N` are used to denote the type of the corresponding MPI call, i.e., blocking or nonblocking. To determine the two operands of an artificial memory access, MPIRace further analyzes the arguments of the MPI call. Typically, if an MPI library routine needs to access a memory buffer, it provides three arguments, denoted as $\mathscr{S}$, $\mathscr{C}$, and $\mathscr{T}$, to receive the information of the buffer: the start address of the buffer ($\mathscr{S}$), the number of elements to be accessed ($\mathscr{C}$), and the data type of each element ($\mathscr{T}$). For instance, the first three arguments of `MPI_Send()` are

used for this purpose. Therefore, the operands of the artificial memory access operation can be derived from the arguments as follows:

$$\begin{aligned} \texttt{start\_address} &= \mathscr{S} \\ \texttt{size} &= \mathscr{C} \times \text{sizeof}(\mathscr{T}) \end{aligned} \tag{1}$$

For the first two arguments, $\mathscr{S}$ and $\mathscr{C}$, we can simply extract them from the MPI call. But, for the argument $\mathscr{T}$, we need to know which identification approach should be used, due to the differences between different MPI implementations (see Section 3). This can be solved by looking up the type identification table. Once the type is identified, MPIRace can quickly figure out the size. For example, `MPI_CHAR` represents one byte, while `MPI_INT` indicates four bytes.

4.3 Analyzing Shared Memory Accesses

The last step of MPIRace is to analyze shared memory accesses for data race detection. Given two shared memory accesses, A and B, in an MHR parallel region $\mathcal{R}$, MPIRace classifies them as a data race if they satisfy the following conditions. First, at least one of A and B is a memory write operation. Second, A and B may access the same memory location. Third, A and B can be executed concurrently in a nondeterministic happens-before order. For instance, if $\mathcal{R}$ is an intra-rank MHR region, A and B may be executed by the MPI program and the MPI library within the rank, respectively. Or, they should be executed by different ranks if $\mathcal{R}$ is an inter-rank MHR region. Note that these conditions align with the conditions of a data race we discussed in Section 2. Next, we explain how MPIRace checks these conditions.

Analyzing Access Types and Locations. It is quite intuitive to check whether a memory access is a write operation or not, as both the memory accesses in the MPI program and the artificial memory accesses created by MPIRace have the access type information. To determine whether two memory accesses may access the same memory location, MPIRace checks whether there is any *overlap* between the two accessed memory locations. Suppose b_1 and b_2 are the start addresses, and s_1 and s_2 are the sizes of the two accesses, respectively. An overlap means the following logical expression is true: $b_1 \leq b_2 < b_1+s_1 \vee b_2 \leq b_1 < b_2+s_2$. To evaluate this expression, MPIRace conducts a path-sensitive backward dataflow analysis to find all potential definitions of b_1 and b_2. For example, an address may be derived from a pointer that points to a stack or heap buffer. In case s_1 or s_2 is not a constant, MPIRace also finds their definitions. Due to the inherent limitations of static program analysis techniques [27,28], it is possible that MPIRace cannot accurately find the definitions. Therefore, MPIRace conducts the analysis conservatively to avoid potential false negatives. Though this may pose a risk of more false positives, our experimental results on real-world MPI applications show that the detection results are satisfactory.

Analyzing Access Concurrency. To determine whether two shared memory accesses in an intra-rank MHR region can be executed concurrently, MPIRace

checks that at least one of the memory accesses is an artificial memory access and their happens-before order is not determined by the program order. This allows the two accesses to be executed concurrently by the MPI program and the MPI library within the rank. Take the following code as an example:

```
1 int buf = 0;
2 MPI_Isend(buf, 1, MPI_INT, ...); // R_N(buf, 4)
3 buf = 1; // W(buf, 4)
4 MPI_Irecv(buf, 1, MPI_INT, ...); // W_N(buf, 4)
```

In this code, `W(buf, 4)` forms a data race with $\texttt{R}_\texttt{N}\texttt{(buf, 4)}$, because they can be executed concurrently in a nondeterministic happens-before order. However, there is no data race between `W(buf, 4)` and $\texttt{W}_\texttt{N}\texttt{(buf, 4)}$. The reason is that their happens-before order is determined by the program order. That is, `W(buf, 4)` always happens before $\texttt{W}_\texttt{N}\texttt{(buf, 4)}$. Note that $\texttt{R}_\texttt{N}\texttt{(buf, 4)}$ and $\texttt{W}_\texttt{N}\texttt{(buf, 4)}$ also form a data race in this example, as they may be executed concurrently by the MPI library in a nondeterministic happens-before order.

In case the two shared memory accesses are in an inter-rank MHR region, MPIRace checks that they can be executed by two different ranks. This can be done by tracking the MPI routine `MPI_Comm_rank()` because most MPI programs use this routine to get the rank information. This allows MPIRace to figure out the rank(s) that will execute a code region.

5 Implementation

We have implemented a research prototype of MPIRace based on LLVM (version 13.0.0). In this section, we report the issues we encountered during the implementation process and our solutions.

Compiling MPI Programs to LLVM IR. To compile an MPI program to LLVM IR, we pass the LLVM compiler to MPI wrapper compilers with the flags "`-S -emit-llvm`." We also skip the linking stage at the end of the compilation process without combining all LLVM IR files to generate a single file. This allows us to apply MPIRace to an MPI program even if its source code can only be compiled partially with LLVM.

Identifying MPI Calls. In LLVM IR, MPI library calls are treated in the same way as normal library function calls. In most cases, an MPI call is compiled to a '`call`' instruction. However, in some cases, especially for C++ programs, it is possible that an MPI call is compiled to an '`invoke`' instruction. Hence, we need to check both of them to identify MPI calls.

Collecting Memory Accesses. In general, memory accesses are compiled to '`load`' and '`store`' instructions in LLVM IR. Both of them have a *pointer* operand, which specifies the address of the accessed memory location. To determine the size of a memory access, we can check the type of the pointer operand, as LLVM IR requires that the type must be a first-class type with a *known* size. Apart from these instructions, our implementation also supports C standard library routines, e.g., `memcpy()` and `strcpy()`, and C++ STL containers,

e.g., `std::vector` and `std::list`. This is realized by recognizing corresponding function calls in LLVM IR and replacing them with normal '`load`' and '`store`' instructions, similar to modeling shared memory access behaviors of MPI calls.

6 Experimental Results

In this section, we evaluate MPIRace. The goal of the evaluation is to answer the following two questions: 1) Can MPIRace detect data races in MPI programs? 2) How is the detection efficiency of MPIRace? To this end, we apply MPIRace to MPI-CorrBench [12], a benchmark suite for evaluating MPI programming error detection tools, and a wide range of real-world MPI applications, including Ember [9], miniFE [21], Presta [10], SuperLU_DIST [14], and U.S. Naval MPI Tutorials [11]. Our experimental platform is powered by an Intel Xeon E5-2697 14-core CPU with hyper-threading enabled and 194GB main memory. The operating system is Ubuntu 20.04 with the Linux kernel (version 5.11.0).

6.1 Detection Effectiveness

Table 2 shows the data races detected by MPIRace. For some data races, e.g., #3, #13, and #16, the two shared memory accesses come from the same source line. This is because the source line is in a loop and thus forms a data race with itself. To summarize, MPIRace successfully finds 20 data races. This shows the effectiveness of MPIRace on detecting data races in MPI programs.

MPI-CorrBench. For MPI-CorrBench, 8 data races are reported by MPIRace. Here, an interesting observation is that MPIRace successfully uncovers 4 data races in the presumably correct version of the microbenchmark programs, i.e., data races #3, #4, #5, and #6. The paths of the source files of these data races imply that the corresponding microbenchmarks are located in the "correct" directory. This demonstrates the natural stealthiness of data races and the inherent difficulty to recognize them during the development process.

Comparing with Existing Tools. We compare MPIRace with existing representative tools, including MPI-Checker [3], MUST [6], ThreadSanitizer [26], and Helgrind [16]. Table 3 shows the comparison results for the data races in MPI-CorrBench. As shown in the table, MPI-Checker fails to detect any data races. This is expected because MPI-Checker is designed to catch simple MPI programming errors rather than data races. Surprisingly, MUST detects six out of eight data races. Our further study shows that this is not because MUST is able to detect data races. Instead, it considers the programs have misuses of MPI library routines. Since MUST only monitors MPI calls, it cannot detect data races that involve both the MPI library and the target MPI program. Also, it is a dynamic detector and thus suffers from the well-known coverage issue. Both ThreadSanitizer and Helgrind cannot find the data races detected by MPIRace. Overall, we can conclude that MPIRace outperforms existing tools in detecting data races in MPI programs.

Table 2: The details of the data races detected by MPIRace.

Application	ID	Source File	Source Lines
MPI-CorrBench	1	micro-benches/0-level/conflo/pt2pt/ ArgMismatch-MPIIrecv-buffer-overlap.c	36 MPI_Irecv(buffer, N, ... 37 MPI_Irecv(tar_2, N/2, ...
	2	micro-benches/0-level/conflo/pt2pt/ MisplacedCall-MPIWait.c	35 MPI_Isend(buffer, 100000, ... 37 buffer[0] = 10;
	3	micro-benches/0-level/correct/pt2pt/ dtype_send.c	84 MPI_Irecv(rcv_buf, 1, ... 84 MPI_Irecv(rcv_buf, 1, ...
	4	micro-benches/0-level/correct/pt2pt/ inactivereq.c	92 MPI_Irecv(rbuf, 10, ... 92 MPI_Irecv(rbuf, 10, ...
	5	micro-benches/0-level/correct/pt2pt/ patterns.c	87 MPI_Irecv(buffer, buf_size, ... 88 MPI_Irecv(buffer, buf_size, ...
	6	micro-benches/0-level/correct/pt2pt/ patterns.c	116 MPI_Irecv(buffer, buf_size, ... 117 MPI_Irecv(buffer, buf_size, ...
	7	micro-benches/0-level/pt2pt/ ArgMismatch-MPIIrecv-buffer-overlap.c	28 MPI_Irecv(buffer, N, ... 29 MPI_Irecv(&buffer[N/2], N/2, ...
	8	micro-benches/0-level/pt2pt/ MisplacedCall-MPIWait.c	35 MPI_Isend(buffer, 100000, ... 36 buffer[0] = 10;
Ember	9	mpi/halo3d-26/halo3d-26.c	446 MPI_Irecv(edge, nz*vars, ... 488 MPI_Irecv(edge, ny*vars, ...
	10	mpi/halo3d-26/halo3d-26.c	446 MPI_Irecv(edge, nz*vars, ... 495 MPI_Irecv(edge, ny*vars, ...
	11	mpi/halo3d-26/halo3d-26.c	488 MPI_Irecv(edge, ny*vars, ... 495 MPI_Irecv(edge, ny*vars, ...
miniFE	12	ref/src/make_local_matrix.hpp	259 MPI_Irecv(&tmp_buf[i], 1, ... 266 MPI_Send(&tmp_buf[i], 1, ...
Presta	13	com.c	582 MPI_Irecv(comBuf, bufsize, ... 582 MPI_Irecv(comBuf, bufsize, ...
	14	com.c	764 MPI_Irecv(rBuf, bufsize, ... 764 MPI_Irecv(rBuf, bufsize, ...
SuperLU_DIST	15	SRC/psymbfact.c	4776 MPI_Irecv (&sz_msg, 1, ... 4782 if (sz_msg > INT_MAX)
U.S. Naval MPI Tutorials	16	src/MPI_Recv_init.c	150 MPI_Irecv(rbuf, 10, ... 150 MPI_Irecv(rbuf, 10, ...
	17	src/MPI_Request_free.c	135 MPI_Irecv(rbuf, 10, ... 135 MPI_Irecv(rbuf, 10, ...
	18	src/MPI_Send_init.c	159 MPI_Irecv(rbuf, 10, ... 159 MPI_Irecv(rbuf, 10, ...
	19	src/MPI_Ssend_init.c	152 MPI_Irecv(rbuf, 10, ... 152 MPI_Irecv(rbuf, 10, ...
	20	src/MPI_Start.c	106 MPI_Irecv(rbuf, 10, ... 106 MPI_Irecv(rbuf, 10, ...

Real-World MPI Applications. As shown in Table 2, MPIRace successfully reports 12 data races in the evaluated real-world MPI applications. To our surprise, every evaluated MPI application has at least one data race. This shows the importance of conducting data race detection for MPI programs. Among the 12 data races, #15 is caused by concurrent shared memory accesses between the MPI application and a nonblocking MPI library routine, while #12 is caused by a nonblocking MPI routine and a blocking MPI routine. Also, the shared variable of data race #12 is a C++ STL container. This shows that MPIRace is capable of detecting MPI data races in various types.

False Positives and False Negatives. Although MPIRace is a static detector, it does not report any false positives in the detection results. A potential reason

Table 3: The comparison between existing tools and MPIRace.

	1	2	3	4	5	6	7	8
MPI-Checker	✗	✗	✗	✗	✗	✗	✗	✗
MUST	✓	✗	✓	✓	✓	✓	✓	✗
ThreadSanitizer	✗	✗	✗	✗	✗	✗	✗	✗
Helgrind	✗	✗	✗	✗	✗	✗	✗	✗
MPIRace	✓	✓	✓	✓	✓	✓	✓	✓

is that most MPI programs have regular shared memory accesses, which makes static analyses more accurate, compared to general multithreaded programs. Regarding false negatives, we manually created a group of microbenchmarks with artificially injected data races to evaluate MPIRace. The evaluation results show that MPIRace can detect all injected data races. We believe the probability that MPIRace may miss a data race is rather low because of its general design.

6.2 Detection Efficiency

Figure 4 shows the detection time (in seconds) required by MPIRace. For all evaluated MPI applications, the detection time is less than one minute. The longest detection time is spent on MPI-CorrBench, because it contains more than 500 microbenchmarks. As a result, it takes a relatively long time for MPIRace to load the LLVM IR files. In fact, the actual detection time is quite short, i.e., less than 15% of the total time. Overall, we are confident that MPIRace can be applied to large-scale MPI applications for data race detection.

7 Related Work

Many MPI programming tools have been proposed to help developers to find programming errors in MPI programs [1,2,3,5,6,7,13,24,25,29,33,34]. For example, MPI-Checker [3] analyzes the AST of the target MPI program to detect MPI programming errors. MUST [6] dynamically verifies the correctness of each MPI library call. Besides, it can also detect deadlock errors [5,7]. MC-Checker [1] and SyncChecker [2] are two dynamic checkers that aim to detect memory consistency and synchronization errors in MPI programs. However, they are limited to specific types of MPI programs. Similarly, PARCOACH [24] aims to validate MPI collective communications. ParaStack [13] is dedicated to MPI hang detection. NINJA [25] explores noise injection techniques to expose MPI message races. In addition, some other tools [29,33,34] leverage symbolic execution and formal verification techniques to check the correctness of MPI programs.

Though data race detection has been researched for decades, traditional detectors, such as ThreadSanitizer [26] and Helgrind [16], can only detect data races in thread-based shared-memory programs [30]. Some previous research work attempts to detect data races in distributed programs [22,23]. Unfortunately, they only support the UPC programming model, rather than MPI.

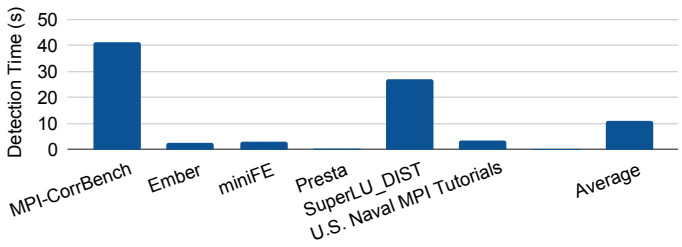

Fig. 4: Detection time of MPIRace.

MPIRace is clearly distinguished from existing tools in two aspects. First, it focuses on MPI data race detection, which is generally much harder than detecting simple MPI programming errors. Therefore, MPIRace augments existing tools, as most of them lack support for data race detection. Second, it is a static data race detector. That means, it can analyze the entire code base of the target MPI program without the need to run the program with sample inputs. In other words, it does not suffer from the notorious coverage issue that confronts existing dynamic tools, such as MUST. Also, since it does not need to run the MPI program, the analysis process can be completely decoupled from the execution environment of the program. This further enhances the practicability of MPIRace, as many MPI programs need to run on dedicated supercomputers, which are typically not available for a tool like MPIRace.

8 Conclusion

In this paper, we present MPIRace, a static data race detector designed specifically for MPI programs. It creates several novel and effective static analysis techniques to overcome the technical challenges of conducting static data race detection for MPI programs. We also implement a research prototype of MPIRace based on the popular LLVM compiler infrastructure. After applying MPIRace to MPI-CorrBench and a wide range of real-world MPI applications, we successfully find 20 data races. Among them, 12 are discovered in real-world MPI applications. Besides, the detection efficiency of MPIRace is extremely high, i.e., less than one minute for every evaluated application. We hope that MPIRace will provide strong support for developers to improve the correctness, reliability, and sustainability of MPI programs.

Acknowledgments

We are very grateful to anonymous reviewers for their valuable feedback and comments. This work was supported in part by the M. G. Michael Award funded by the Franklin College of Arts and Sciences at the University of Georgia and a faculty startup funding offered by the University of Georgia.

References

1. Chen, Z., Dinan, J., Tang, Z., Balaji, P., Zhong, H., Wei, J., Huang, T., Qin, F.: Mc-checker: Detecting memory consistency errors in mpi one-sided applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. p. 499–510. SC '14, IEEE Press (2014). https://doi.org/10.1109/SC.2014.46
2. Chen, Z., Li, X., Chen, J.Y., Zhong, H., Qin, F.: Syncchecker: Detecting synchronization errors between mpi applications and libraries. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium. pp. 342–353 (2012). https://doi.org/10.1109/IPDPS.2012.40
3. Droste, A., Kuhn, M., Ludwig, T.: Mpi-checker: Static analysis for mpi. LLVM '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2833157.2833159
4. Forum, M.P.I.: Mpi: A message-passing interface standard, version 4.0 (Accessed: January 2022), `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`
5. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: Mpi runtime error detection with must: Advances in deadlock detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12, IEEE Computer Society Press, Washington, DC, USA (2012)
6. Hilbrich, T., Schulz, M., de Supinski, B.R., Müller, M.S.: Must: A scalable approach to runtime error detection in mpi programs. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009. pp. 53–66. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
7. Hilbrich, T., de Supinski, B.R., Nagel, W.E., Protze, J., Baier, C., Müller, M.S.: Distributed wait state tracking for runtime mpi deadlock detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2503210.2503237
8. IBM: Ibm spectrum mpi: Accelerating high-performance application parallelization (Accessed: January 2022), `https://www.ibm.com/products/spectrum-mpi`
9. Laboratories, S.N.: Ember communication pattern library (Accessed: January, 2022), `https://proxyapps.exascaleproject.org/app/ember-communication-patterns`
10. Laboratory, L.L.N.: Presta mpi benchmark 1.3.0 (Accessed: January, 2022), `https://github.com/LLNL/phloem/tree/master/presta-1.3.0`
11. Laboratory, U.N.R.: Message passing interface (mpi) tutorials (Accessed: January, 2022), `https://github.com/USNavalResearchLaboratory/mpi_tutorials`
12. Lehr, J.P., Jammer, T., Bischof, C.: Mpi-corrbench: Towards an mpi correctness benchmark suite. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing. p. 69–80. HPDC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3431379.3460652
13. Li, H., Chen, Z., Gupta, R.: Parastack: Efficient hang detection for mpi programs at large scale. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3126908.3126938
14. Li, X.S., Demmel, J.W.: Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans. Math. Softw. **29**(2), 110–140 (Jun 2003). https://doi.org/10.1145/779359.779361

15. LLVM: The llvm compiler infrastructure (Accessed: January, 2022), https://llvm.org
16. Manual, V.U.: Helgrind: a thread error detector (Accessed: January, 2022), https://www.valgrind.org/docs/manual/hg-manual.html
17. Microsoft: Microsoft mpi (Accessed: January 2022), https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
18. MPI: Mpi forum (Accessed: January 2022), https://www.mpi-forum.org
19. MPI, O.: Open source high performance computing (Accessed: January 2022), https://www.open-mpi.org
20. MPICH: High-performance portable mpi (Accessed: January 2022), https://www.mpich.org
21. Organization, M.: minife finite element mini-application (Accessed: January, 2022), https://proxyapps.exascaleproject.org/app/minife
22. Park, C.S., Sen, K., Hargrove, P., Iancu, C.: Efficient data race detection for distributed memory parallel programs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063452
23. Park, C.S., Sen, K., Iancu, C.: Scaling data race detection for partitioned global address space programs. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. p. 47–58. ICS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2464996.2465000
24. Saillard, E., Carribault, P., Barthou, D.: Parcoach: Combining static and dynamic validation of mpi collective communications. The International Journal of High Performance Computing Applications **28**(4), 425–434 (2014). https://doi.org/10.1177/1094342014552204
25. Sato, K., Ahn, D.H., Laguna, I., Lee, G.L., Schulz, M., Chambreau, C.M.: Noise injection techniques to expose subtle and unintended message races. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 89–101. PPoPP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3018743.3018767
26. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. p. 62–71. WBIA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1791194.1791203
27. Tan, T., Li, Y., Ma, X., Xu, C., Smaragdakis, Y.: Making pointer analysis more precise by unleashing the power of selective context sensitivity. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485524
28. Thiessen, R., Lhoták, O.: Context transformations for pointer analysis. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 263–277. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062359
29. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., Supinski, B.R.d., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for mpi programs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. p. 1–10. SC '10, IEEE Computer Society, USA (2010). https://doi.org/10.1109/SC.2010.7
30. Wang, W., Lin, P.H.: Does it matter? ompsanitizer: An impact analyzer of reported data races in openmp programs. In: Proceedings of the ACM International Confer-

ence on Supercomputing. p. 40–51. ICS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3447818.3460379
31. Wang, W., Wang, Z., Wu, C., Yew, P.C., Shen, X., Yuan, X., Li, J., Feng, X., Guan, Y.: Localization of concurrency bugs using shared memory access pairs. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 611–622. ASE '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642972
32. Wang, W., Wu, C., Ratanaworabhan, P., Yuan, X., Wang, Z., Li, J., Feng, X.: Dynamically tolerating and detecting asymmetric races. Journal of Computer Research and Development **51**(8), 1748–1763 (Aug 2014). https://doi.org/10.7544/issn1000-1239.2014.20130123
33. Ye, F., Zhao, J., Sarkar, V.: Detecting mpi usage anomalies via partial program symbolic execution. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. SC '18, IEEE Press (2018). https://doi.org/10.1109/SC.2018.00066
34. Yu, H., Chen, Z., Fu, X., Wang, J., Su, Z., Sun, J., Huang, C., Dong, W.: Symbolic verification of message passing interface programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 1248–1260. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380419
35. Yuan, X., Wang, Z., Wu, C., Yew, P.C., Wang, W., Li, J., Xu, D.: Synchronization identification through on-the-fly test. In: Proceedings of the 19th International Conference on Parallel Processing. p. 4–15. Euro-Par'13, Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_3