Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.047392

ARTICLE

Static Analysis Techniques for Fixing Software Defects in MPI-Based Parallel Programs

Norah Abdullah Al-Johany[1,*], **Sanaa Abdullah Sharaf**[1,2], **Fathy Elbouraey Eassa**[1,2] and **Reem Abdulaziz Alnanih**[1,2,*]

[1]Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, 21589, Saudi Arabia

[2]Software Engineering and Distributed System Research Group, King Abdulaziz University, Jeddah, 21589, Saudi Arabia

*Corresponding Authors: Norah Abdullah Al-Johany. Email: nora-abdallah@hotmail.com; Reem Abdulaziz Alnanih. Email: ralnanih@kau.edu.sa

Received: 10 November 2023 Accepted: 20 December 2023 Published: 15 May 2024

ABSTRACT

The Message Passing Interface (MPI) is a widely accepted standard for parallel computing on distributed memory systems. However, MPI implementations can contain defects that impact the reliability and performance of parallel applications. Detecting and correcting these defects is crucial, yet there is a lack of published models specifically designed for correcting MPI defects. To address this, we propose a model for detecting and correcting MPI defects (DC_MPI), which aims to detect and correct defects in various types of MPI communication, including blocking point-to-point (BPTP), nonblocking point-to-point (NBPTP), and collective communication (CC). The defects addressed by the DC_MPI model include illegal MPI calls, deadlocks (DL), race conditions (RC), and message mismatches (MM). To assess the effectiveness of the DC_MPI model, we performed experiments on a dataset consisting of 40 MPI codes. The results indicate that the model achieved a detection rate of 37 out of 40 codes, resulting in an overall detection accuracy of 92.5%. Additionally, the execution duration of the DC_MPI model ranged from 0.81 to 1.36 s. These findings show that the DC_MPI model is useful in detecting and correcting defects in MPI implementations, thereby enhancing the reliability and performance of parallel applications. The DC_MPI model fills an important research gap and provides a valuable tool for improving the quality of MPI-based parallel computing systems.

KEYWORDS

High-performance computing; parallel computing; software engineering; software defect; message passing interface; deadlock

1 Introduction

Distributed computing systems have become an essential component of current high-performance computing (HPC) and data-intensive applications. The Message Passing Interface (MPI) has gained widespread acceptance as a typical tool for facilitating message passing in distributed computing environments. It provides a robust programming model and communication protocol that allow developers to harness the full potential of parallel systems. MPI is used in numerous scientific and engineering fields, including computational physics, computational biology, weather forecasting, and large-scale simulations. Its versatility and scalability have made it a go-to solution for parallel computing on a wide range of architectures, from small clusters to supercomputers [1].

MPI allows multiple processes or threads to exchange messages and synchronize their activities in a distributed computing environment. It provides a set of functions and routines that enable communication, data transfer, and coordination among processes. In MPI, data exchange between different processes is achieved using various communication functions provided by the MPI library. The most commonly used functions for data exchange are MPI_Send and MPI_Recv. Here are the steps to follow when using MPI for data exchange [2]:

1. Initialize MPI: Before performing any MPI operations, you need to initialize the MPI library by calling the MPI_Init function. This initializes the MPI environment and sets up communication channels between processes.
2. Create Communicator: After initialization, you can create a communicator that represents a group of processes involved in the data exchange. MPI_COMM_WORLD is the predominant communicator utilized in MPI, encompassing all processes within a program.
3. Assign Rank: Each process in the communicator is assigned a unique rank, which is an integer value ranging from 0 to the total number of processes minus one. The rank is used to distinguish between different processes.
4. Data Preparation: The processes need to prepare the data that they want to send or expect to receive. This could involve creating arrays, buffers, or any other data structure required for the computation.
5. Send Data: To send data from one process to another, the MPI_Send function specifies the data buffer, the number of elements to send, the data type of the elements, the rank of the receiving process and a tag for message identification.
6. Receive Data: To receive incoming data, the receiving process employs the MPI_Recv function, which requires specifying the data buffer, the maximum number of elements to be received, the data type of the elements, the rank of the sending process, and the message identification tag.
7. Synchronization: After the data exchange, it is often necessary to synchronize the processes to ensure that all sending and receiving has been completed. This is achieved using the MPI_Barrier function, which acts as a synchronization point.
8. Finalize MPI: Once the data exchange is complete, the MPI environment is finalized by calling the MPI_Finalize function, which releases any resources associated with MPI and terminates the MPI execution.

MPI provides various other functions for different communication patterns and requirements. These include non-blocking point-to-point (NBPTP) communication operations such as MPI_Isend and MPI_Irecv, which enable asynchronous communication, as well as collective communication (CC) operations such as MPI_Bcast and MPI_Reduce, which facilitate communication among multiple processes.

The specific functions and approach you choose depend on the communication pattern and the structure of parallel program.

While MPI offers significant benefits in terms of scalability and performance, it also introduces challenges related to deadlock (DL), race conditions (RC), and message mismatches (MM). These issues can undermine the correctness and efficiency of parallel applications, requiring detection and correction mechanisms to be implemented [2].

DL occurs when multiple processes or threads in a parallel program become indefinitely blocked, unable to make progress due to circular dependencies in resource allocation. The detection and resolution of DL are vital to ensure the accuracy and efficiency of parallel computations. MPI provides a powerful mechanism for inter-process communication and coordination; however, the inherent complexity of parallel programs combined with the distributed nature of MPI applications can give rise to DL. These DL can lead to system failures, loss of computational progress, and inefficient resource utilization [3].

RC arise when multiple processes or threads simultaneously access shared resources, leading to non-deterministic and frequently incorrect computation outcomes. Detecting and mitigating RC is crucial to ensuring the correctness and reliability of parallel computations. MPI provides a powerful framework for inter-process communication and coordination in distributed computing environments. However, the shared memory nature of MPI programs can introduce RC when multiple processes concurrently access and modify shared data. RC can lead to data corruption, inconsistent results, and program crashes, all of which undermine the integrity of parallel computations [4].

MM occurs when the communication patterns and data structures used by different processes or threads do not correspond, leading to incorrect data transfers, synchronization problems, and program failure. Although MPI provides a powerful framework for inter-process communication and coordination in distributed computing environments, the distributed nature of MPI applications, combined with the complexity of parallel programs, can introduce various types of mismatches. These include data type mismatches (DTM), buffer size mismatches, tag mismatches, and communication pattern mismatches, among others. DTM occur when processes expect different data types for the same communication operation. For example, one process may send an integer value while another process expects to receive a floating-point value. Such mismatches can lead to data corruption, interpretation errors, and incorrect computation results. Processes must agree on the data types used in communication operations to avoid DTM [5]. Buffer size mismatches arise when the sizes of the sending and receiving buffers do not match. This can result in data truncation, buffer overflows, or memory corruption. Buffer sizes between communicating processes need to be the same to ensure the integrity of data transfers. Tag mismatches occur when the tags assigned to communication operations do not match between the sender and receiver processes. MPI uses tags to identify and order messages, and mismatched tags can lead to incorrect message routing, lost messages, or unexpected message deliveries. Coordinating the use of tags between communicating processes will prevent tag mismatches and maintain the correct message semantics. Communication pattern mismatches arise when the communication patterns expected by different processes do not align. For example, one process may initiate a collective operation while another process is not ready to participate. This can result in synchronization issues, DL, or incorrect computation outcomes. Ensuring consistent communication patterns among processes is essential to avoid communication pattern mismatches and maintain the accuracy of parallel computations [5].

These issues can be highly detrimental to the performance and accuracy of MPI applications, and they are often difficult to detect and correct manually. As a result, there has been a growing interest in developing automated tools to detect and correct these defects in MPI applications. Static analysis tools are one solution being used at present. These tools analyze a program's source code or intermediate representation without running it. They examine the program's structure, syntax, and data flow to find potential problems and breaches of coding standards [6]. Dynamic-analysis tools are another solution. They examine programs during runtime and monitor the program's behavior, execution paths, and data flow to identify problems that may occur during program execution [7].

Detecting and correcting these defects is crucial, yet there is a lack of published models specifically designed for correcting MPI defects. We propose a model, DC_MPI, that aims to detect and correct defects in various types of MPI communication. The DC_MPI model encompasses the detection and correction of defects in both blocking point-to-point (BPTP) and NBPTP communication, as well as CC. Defects addressed by the DC_MPI model include illegal MPI calls, DL, RC, and MM. This approach has the potential to significantly reduce the time and effort required for MPI correction applications, improving reliability and performance. The scope of this research encompasses the following aspects:

- Defect Types: The DC_MPI model is specifically designed to detect and correct defects pertaining to BPTP, NBPTP and CC in MPI code. These defects encompass illegal MPI calls, DL, RC, and MM.
- Evaluation Dataset: The model's performance has been evaluated using a 40-example dataset. These examples were carefully selected to represent a diverse range of defect scenarios commonly encountered in MPI-based communication.
- Detection Accuracy: Within the evaluated dataset, the model successfully detected 37 out of the 40 MPI codes, indicating a detection accuracy rate of 92.5%.
- Execution Duration: The model's processing time was measured. The execution duration ranged between 0.81 and 1.36 s. This demonstrates the model's efficiency in providing timely feedback on defect detection and correction.

The scope of this research is specific to the proposed DC_MPI model and its application in detecting and correcting defects in BPTP, NBPTP and CC scenarios within MPI code. The evaluation results, including the detection accuracy and execution duration, provide insights into the model's performance within this defined scope.

The paper is organized as follows: Other recent research on detecting MPI defects is introduced in Section 2, the design of our DC_MPI model is described in Section 3, and Section 4 provides an explanation of the results obtained. A discussion of the obtained results is presented in Section 5, while Section 6 outlines the limitations of the model. The conclusion of the paper is presented in Section 7.

2 Related Work

Recent research on detecting DL, RC, and MM defects in MPI applications has focused on developing efficient and scalable techniques for preventing these defects. In this section, we introduce recent work on this topic.

Droste et al. [8] develop and implement MPI-Checker, a static analysis tool specifically designed to verify the proper usage of the MPI API in C and C++ code. The foundation of MPI-Checker lies in Clang's Static Analyzer and supports path-sensitive and non-path-sensitive analysis. It identifies issues like double NBPTP calls, unmatched waits, and waiting for unused requests. The tool also provides experimental features to detect potential DL and MM partner calls in point-to-point function invocations. The results show that MPI-Checker effectively detects a wide range of errors, including DL, buffer overflows, and invalid MPI usage.

Vetter et al. [9] describe Umpire, a dynamic software testing tool for MPI programs. Umpire uses a combination of runtime instrumentation and machine-learning techniques to automatically generate test cases for MPI programs. The tool is designed to be highly scalable and can be used to test large-scale HPC applications running on thousands of processors. The paper presents several case studies where Umpire was used to identify errors in real-world MPI applications. The results show that Umpire effectively detects a wide range of errors, including RC, DL, and buffer overflows.

Nguyen et al. [10] describe Parcoach, a static analysis tool for MPI programs that adds support for validating NBPTP and persistent communications. The Parcoach extension uses a combination of static analysis techniques, including dataflow analysis and abstract interpretation, to identify potential errors in NBPTP and persistent communication operations. The tool can detect a wide range of issues, including DL, RC, and incorrect usage of MPI functions.

Alnemari et al. [11] introduce a methodology that merges static and dynamic analysis to effectively identify dynamic errors in MPI programs. The technique employs static analysis to identify potential errors in the program's source code and dynamic analysis to detect errors that may arise during runtime. The static analysis component uses a dataflow analysis technique to identify potential buffer overflows, use-after-free errors, and other memory-related errors in the program. The dynamic analysis component uses an MPI profiling tool to monitor the communication patterns between MPI processes and detect errors such as DL and incorrect usage of MPI functions.

Saillard et al. [12] outline a methodology for the combined static and dynamic validation of MPI CC, which are a set of MPI functions that allow groups of processes to perform coordinated operations. The approach leverages a combination of static analysis and runtime monitoring to identify errors in MPI CC. The static analysis component uses a dataflow analysis technique to identify potential errors in the program's source code, such as incorrect usage of MPI functions and buffer overflows. The dynamic monitoring component uses an MPI profiling tool to monitor the communication patterns between MPI processes and detect errors, such as DL and incorrect usage of MPI collective functions.

Saillard et al. [13] describe a technique for detecting local concurrency errors in MPI-RMA (Remote Memory Access) programs using static analysis. Local concurrency errors occur when multiple threads or processes access shared memory regions concurrently, leading to RC, DL, and other reliability issues. The technique uses dataflow analysis to identify potential local concurrency errors in the program's source code. The analysis focuses on the access to shared memory regions in MPI-RMA operations and identifies potential conflicts in the access.

Protze et al. [14] describe Marmot Umpire Scalable Tool (MUST), an MPI runtime error detection tool that provides advanced error reports for MPI applications. MUST leverages a combination of static and dynamic analysis techniques to detect errors in MPI programs, including memory, communication, and synchronization errors. The tool provides detailed error reports that include information on the error's location, the MPI function that caused it, and the call stack that led to it.

Wei et al. [15] describe the MPI runtime communication deadlock detection framework, which detects communication DL in MPI programs at runtime. The framework uses a combination of runtime monitoring and static analysis to detect communication DL. The runtime monitoring component of the framework uses MPI profiling tools to capture communication patterns between MPI processes, while the static analysis component uses a dataflow analysis technique to identify potential DL scenarios in the program's source code.

Vakkalanka et al. [16] present in-situ partial order, a tool specifically designed for model checking MPI programs. Model checking verifies the correctness of concurrent systems by exploring all possible executions of the system and checking whether they satisfy a set of properties. ISP uses static and dynamic analysis techniques to model-check MPI programs. The static analysis component of ISP uses a dataflow analysis technique to identify potential errors in the program's source code, while the dynamic analysis component uses runtime monitoring to explore all possible executions of the program.

Alghamdi et al. [17] present a hybrid testing technique that combines static and dynamic analysis to effectively test MPI-based programs and detect runtime errors associated with various types of MPI communications. It is designed to detect errors in MPI programs, such as DL, buffer overflows, and RC. The static analysis component of this technique uses a dataflow analysis technique to identify potential errors in the program's source code. The dynamic analysis component uses a combination of runtime monitoring and symbolic execution to detect errors that may occur at runtime.

Altalhi et al. [18] devise a testing tool for parallel hybrid systems. This tool is designed to dynamically analyze the source code of C++, MPI, OpenMP, and Compute Unified Device Architecture (CUDA) systems, with the objective of detecting runtime errors in real-time. The tool showcases efficient identification and categorization of runtime errors in tri-level programming models, underscoring the necessity for a dedicated parallel testing tool for MPI, OpenMP, and CUDA.

Li et al. [19] propose a deadlock detection approach based on match sets that avoids the need to explore numerous execution paths. The approach employs a match detection rule that utilizes the Lazy Lamport Clocks Protocol to generate initial match sets. To further refine these match sets, three algorithms are developed based on the non-overtaking rule and the MPI communication mechanism. The refined match sets are then analyzed to detect DL. The authors conducted experimental evaluations of 15 different programs to assess the effectiveness of the approach. The results indicate that it is exceptionally efficient in detecting DL in MPI programs, especially in scenarios characterized by a significant number of interleavings.

We have identified no published models for correcting MPI defects in the literature. To fill this gap, we introduce the DC_MPI model, specifically designed to both detect and correct the defects commonly encountered in MPI programs. These defects include illegal MPI calls, DL, RC, and MM. By leveraging the capabilities of the DC_MPI model, developers can effectively enhance the reliability and robustness of MPI-based applications.

3 Detect and Correct Software Defects in Message-Passing Interfaces

MPI is a widely adopted standard for parallel programming, extensively utilized in various domains that enables multiple processes to communicate and collaborate in HPC on distributed memory systems; however, any communication mechanism can introduce defects that lead to program failure or performance degradation. Various types of MPI communication exist, including BPTP, NBPTP, and CC, each with its own potential sources of defects. To identify and correct these defects, we propose DC_MPI, a static analysis tool designed to analyze the source code of MPI programs without the need for execution. This tool aims to detect and correct illegal MPI calls, DL, RC, and MM defects specific to each type of MPI communication.

DC_MPI operates by performing lexical analysis and parsing of MPI calls to ensure they conform to MPI standards. It has a specific focus on identifying illegal MPI calls, both prior to MPI_Init and subsequent to MPI_Finalize, which are the initialization and finalization stages of MPI programs, respectively. Additionally, DC_MPI conducts source code analysis to detect MPI-related calls, their locations, types, and arguments. DC_MPI utilizes vectors to store relevant sending and receiving arguments. It then compares these vectors to identify instances where MPI defects need to be addressed. Fig. 1 provides a visual representation of DC_MPI in action.

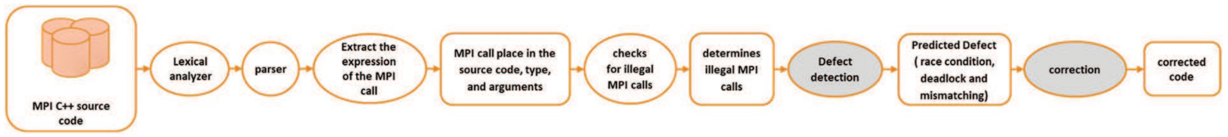

MPI C++ source code
Lexical analyzer
parser
Extract the expression of the MPI call
MPI call place in the source code, type, and arguments
checks for illegal MPI calls
determines illegal MPI calls
Defect detection
Predicted Defect (race condition, deadlock and mismatching)
correction
corrected code

Figure 1: DC_MPI workflow for detecting and correcting software defects in MPI programs

3.1 Detection and Correction of Software Defects in BPTP Communication

DC_MPI has tackled three essential directives for BPTP communication: MPI_Send, MPI_Recv, and MPI_Sendrecv. These directives were chosen based on their extensive utilization in important applications [17].

To ensure MPI calls follow MPI call rules, DC_MPI parses and performs a lexical analysis of each one. Following that, it examines for illegal MPI calls prior to MPI_Init and subsequent to MPI_Finalize. Then it corrects any illegal MPI calls by relocating the call to before MPI_Init or after MPI_Finalize, as appropriate. The illegal call correction algorithm is shown in Fig. 2.

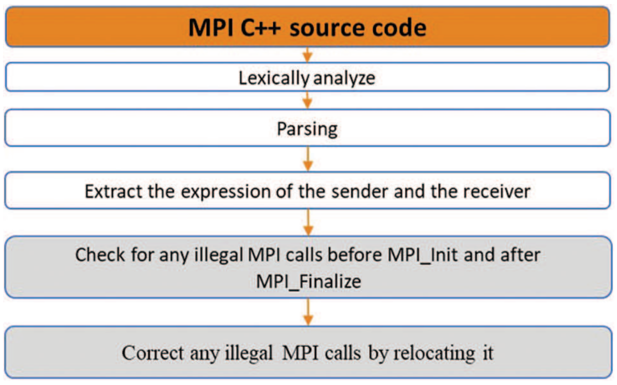

MPI C++ source code
Lexically analyze
Parsing
Extract the expression of the sender and the receiver
Check for any illegal MPI calls before MPI_Init and after MPI_Finalize
Correct any illegal MPI calls by relocating it

Figure 2: Correct any illegal MPI calls

In addition, DC_MPI conducts source code analysis to identify the existence of the three MPI-related function calls. It ascertains their specific location within the source code, their respective types, and thoroughly examines the contents of any sender arguments involved (such as the sender line number, sender buffer, count, data type of the elements in the buffer, receiver, message tag, sender communicator, and sender) that are present. It also determines the receiver arguments (receiver line number, receiver buffer, count, data type, receiver, message tag, communicator, MPI status, and sender). Since the MPI_Sendrecv calls display the same behavior but have different structures, they are examined and analyzed similarly to the first two calls. DC_MPI creates several vectors to store sending and receiving call arguments. The pairings are then recognized, and each sender is associated with the appropriate receiver and recorded in a vector for use in defect detection. The algorithm for determining sender and receiver pairs and saving them in the send_recv_info vector is shown in Fig. 3.

The number of senders and receivers is then compared, and it is determined whether each sender corresponds to exactly one receiver. A RC may exist when there are more senders than receivers because it suggests a lack of resources. On the other hand, when there are more receivers than senders, there is a potential risk of DL occurring. This is because all the receivers will be waiting for messages that they are not receiving [20].

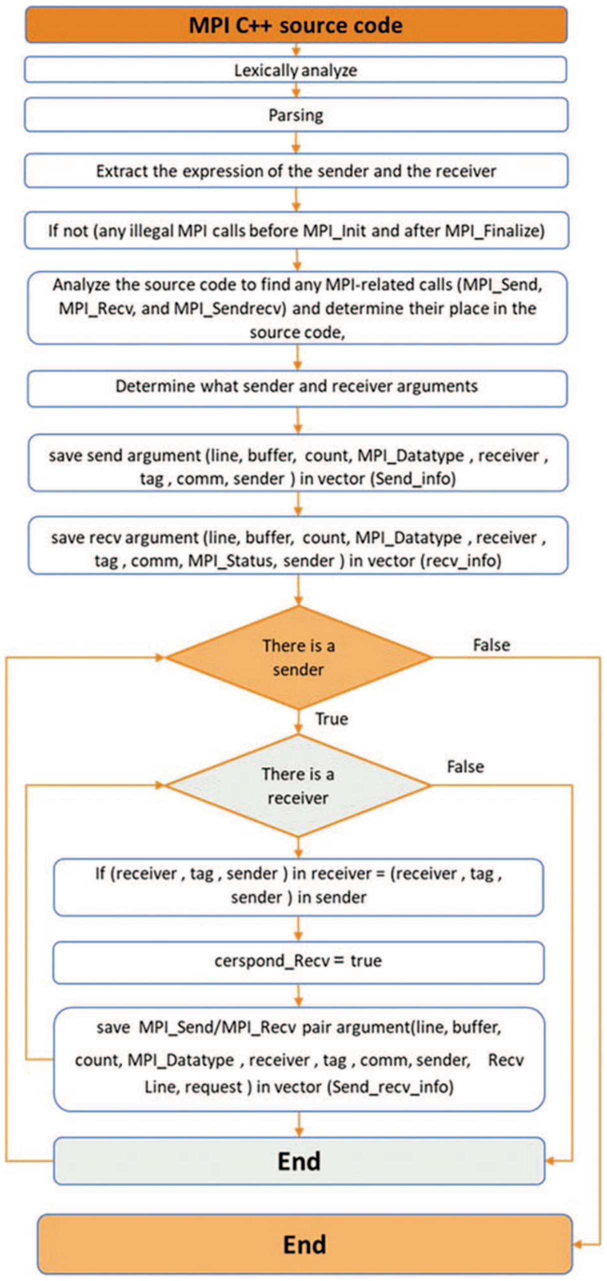

MPI C++ source code
Lexically analyze
Parsing
Extract the expression of the sender and the receiver
If not (any illegal MPI calls before MPI_Init and after MPI_Finalize)
Analyze the source code to find any MPI-related calls (MPI_Send, MPI_Recv, and MPI_Sendrecv) and determine their place in the source code,
Determine what sender and receiver arguments
save send argument (line, buffer, count, MPI_Datatype , receiver , tag , comm, sender) in vector (Send_info)
save recv argument (line, buffer, count, MPI_Datatype , receiver , tag , comm, MPI_Status, sender) in vector (recv_info)
There is a sender
False
True
There is a receiver
False
If (receiver , tag , sender) in receiver = (receiver , tag , sender) in sender
cerspond_Recv = true
save MPI_Send/MPI_Recv pair argument(line, buffer, count, MPI_Datatype , receiver , tag , comm, sender, Recv Line, request) in vector (Send_recv_info)
End
End

Figure 3: Determining sender and receiver pairs and saving them

The defect of a sender without a matching receiver is corrected by attaching a corresponding receiver to the sender. First, the best line is selected to add the receiver's call so that it defines all the variables used in the receiver's call. Then the receiver's call is added after the last change in the variable's value. Next, the optimal buffer is determined, for which an appropriate size was previously utilized. Similarly, the appropriate sender is added to the receiver's call. The algorithm for adding the receiver's call is shown in Fig. 4.

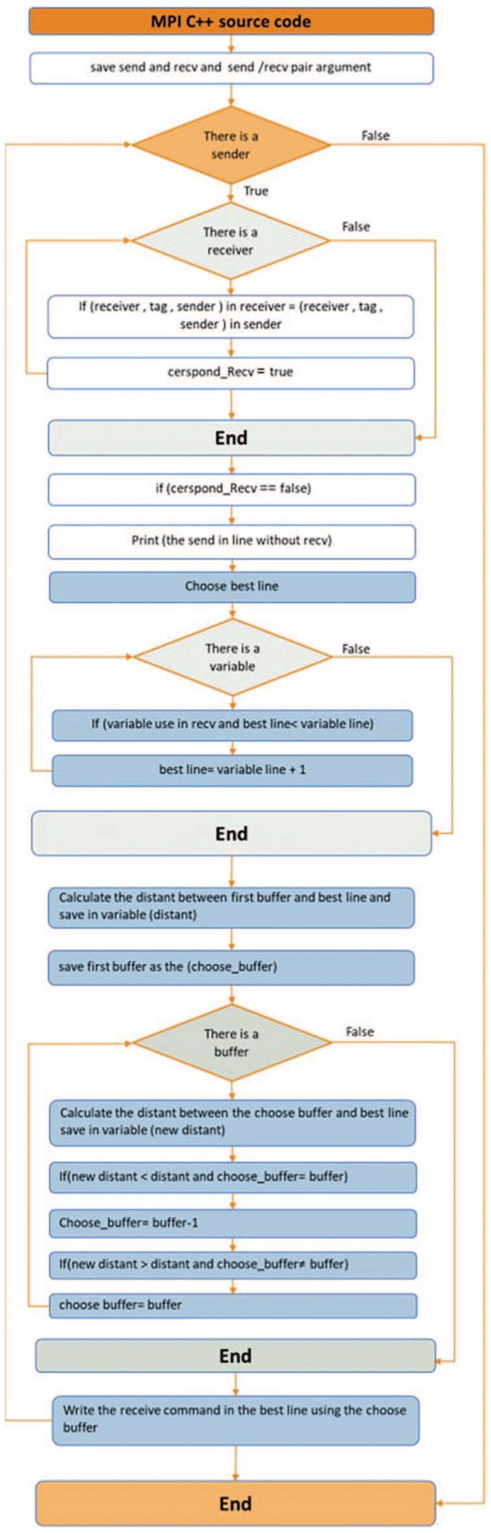

MPI C++ source code
save send and recv and send /recv pair argument
There is a sender
False
True
There is a receiver
False
If (receiver , tag , sender) in receiver = (receiver , tag , sender) in sender
cerspond_Recv = true
End
if (cerspond_Recv == false)
Print (the send in line without recv)
Choose best line
There is a variable
False
If (variable use in recv and best line< variable line)
best line= variable line + 1
End
Calculate the distant between first buffer and best line and save in variable (distant)
save first buffer as the (choose_buffer)
There is a buffer
False
Calculate the distant between the choose buffer and best line save in variable (new distant)
If(new distant < distant and choose_buffer= buffer)
Choose_buffer= buffer-1
If(new distant > distant and choose_buffer≠ buffer)
choose buffer= buffer
End
Write the receive command in the best line using the choose buffer
End

Figure 4: Algorithm for adding receiver call

The incorrect sequence of tags might also result in a defect. If a send is finished before the associated receive is posted, a DL will result. Depending on the order in which calls are implemented and the size of the message, MPI_Send may cause a real or potential DL [20]. This can be corrected by looking at the vector of the pairs and determining the order of the sender and receiver. If the first sender's line is smaller than the second sender's line, the first receiver's line must also be smaller than the second receiver's line so the first and second senders' messages are received. If not, the first and second receivers are swapped, as shown in Fig. 5.

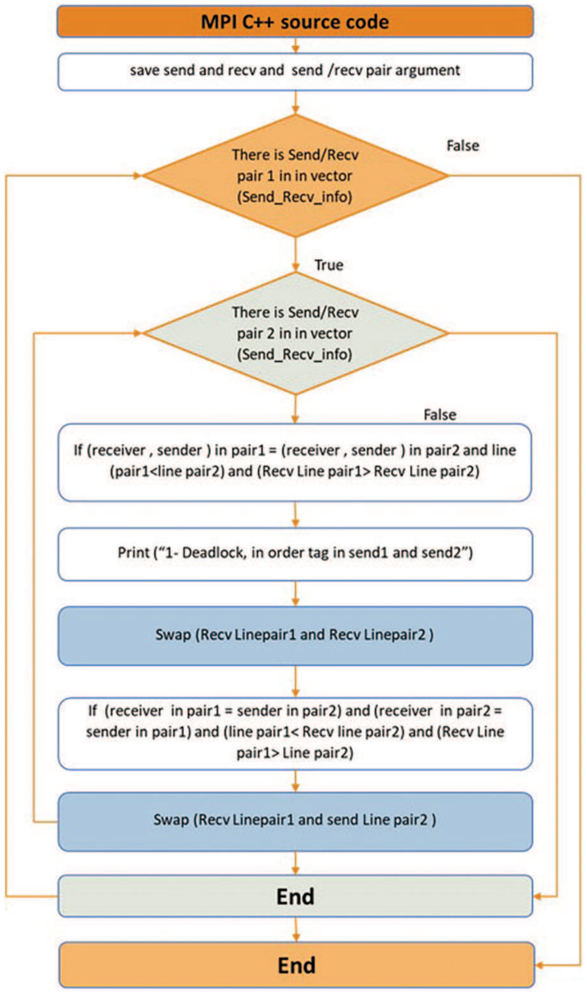

MPI C++ source code
save send and recv and send /recv pair argument
There is Send/Recv pair 1 in in vector (Send_Recv_info)
False
True
There is Send/Recv pair 2 in in vector (Send_Recv_info)
False
If (receiver , sender) in pair1 = (receiver , sender) in pair2 and line (pair1<line pair2) and (Recv Line pair1> Recv Line pair2)
Print ("1- Deadlock, in order tag in send1 and send2")
Swap (Recv Linepair1 and Recv Linepair2)
If (receiver in pair1 = sender in pair2) and (receiver in pair2 = sender in pair1) and (line pair1< Recv line pair2) and (Recv Line pair1> Line pair2)
Swap (Recv Linepair1 and send Line pair2)
End
End

Figure 5: Algorithm to swap the tag's order if it is incorrect

MM can occur in MPI when the sender and receiver have differing message sizes and data types for the same communication. In MPI implementations, the sender's data type is not explicitly specified in messages, making it challenging to detect if the incoming data is being incorrectly interpreted. As a result, transmission issues stemming from DTM may remain undetected [20]. The MM defect is corrected by changing the receiver's message size and data type to correspond with the size and type of the sender call, as shown in Fig. 6.

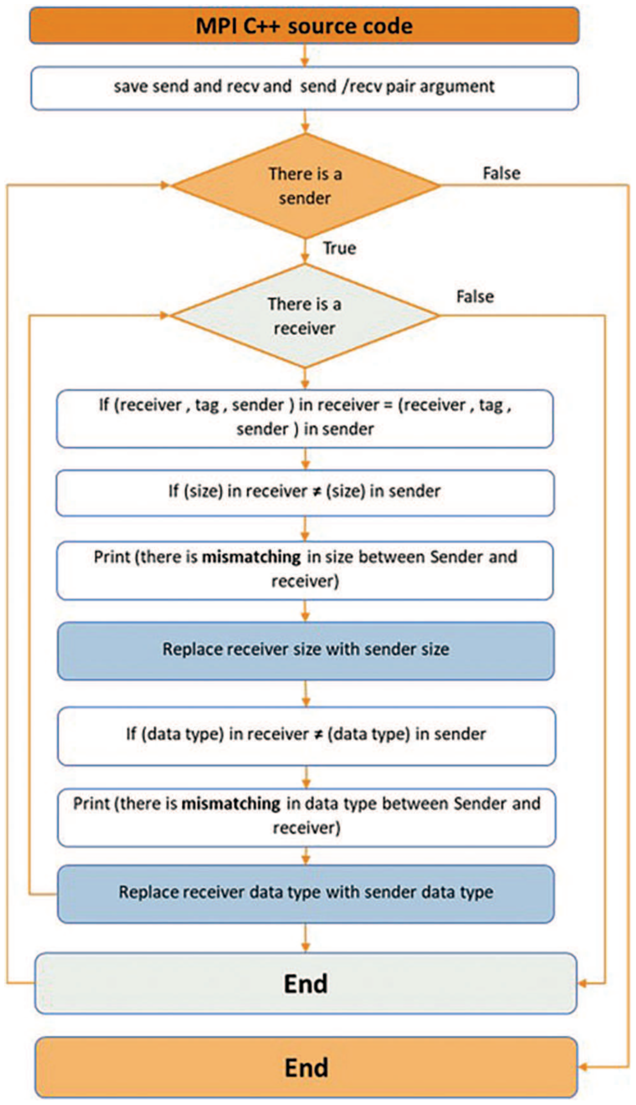

MPI C++ source code
save send and recv and send /recv pair argument
There is a sender
False
True
There is a receiver
False
If (receiver , tag , sender) in receiver = (receiver , tag , sender) in sender
If (size) in receiver ≠ (size) in sender
Print (there is mismatching in size between Sender and receiver)
Replace receiver size with sender size
If (data type) in receiver ≠ (data type) in sender
Print (there is mismatching in data type between Sender and receiver)
Replace receiver data type with sender data type
End
End

Figure 6: MM defect correction algorithm

Reading and writing to the same MPI buffer (RWSB) simultaneously may also result in RC. This defect can be resolved by selecting the most recently utilized buffer and replacing the current one, as shown in Fig. 7.

Using "AS" or "AT" (a wildcard) might lead to a DL or RC defect [20]. These can be fixed by requesting that the user input the proper tag or source rather than any, as shown in Fig. 8.

Moreover, DL can occur in scenarios where multiple messages are either sent to the same recipient or received from the same sender using the identical tag number. If a sender tries to dispatch two messages to the same destination, both associated with the same sender, the operation will fail to receive the second message [20]. This can be resolved by removing one of the calls if there is no corresponding response, as shown in Fig. 9.

3.2 Detection and Correction of Software Defects in NBPTP Communication

DC_MPI detects the usage of MPI_Isend and MPI_Irecv in NBPTP communication. These directives were selected due to their popularity and common usage in various programs [20]. The code is then analyzed for adherence to MPI call rules and checks for illegal MPI calls both prior to MPI_Init and subsequent to MPI_Finalize.

Similarly, in BPTP communication, DC_MPI scans the source code for MPI-related calls, determining their order, kind, and sender arguments in MPI_Isend (such as line number, buffer, count, data type, receiver, tag, communicator, sender, and request). It also identifies receiver arguments in MPI_Irecv, similar to MPI_Isend, along with waiting arguments (line, request, corresponding sender or receiver). DC_MPI creates vectors to store pairs of Isend, Irecv, and wait calls.

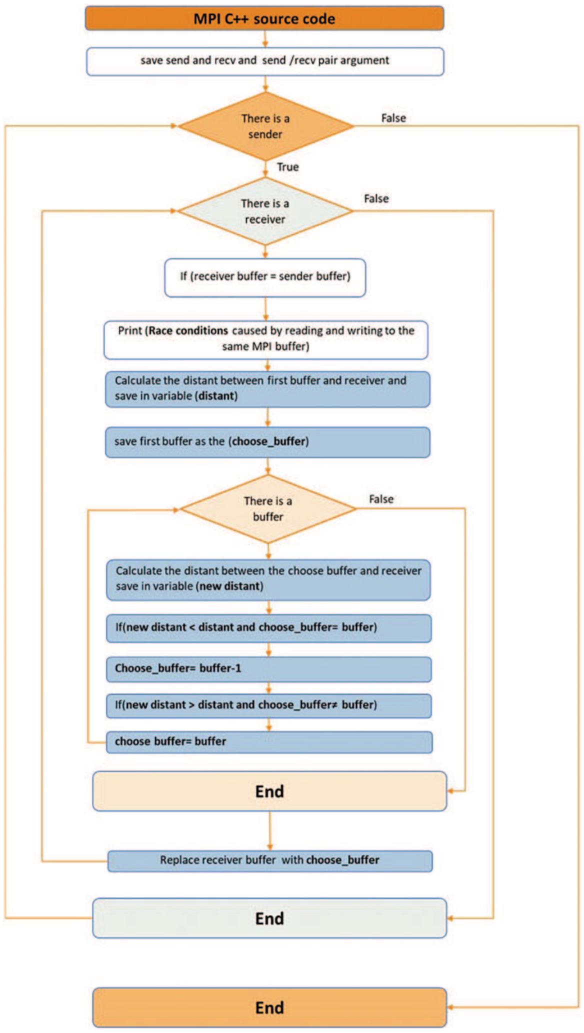

MPI C++ source code
save send and recv and send /recv pair argument
There is a sender
False
True
There is a receiver
False
If (receiver buffer = sender buffer)
Print (Race conditions caused by reading and writing to the same MPI buffer)
Calculate the distant between first buffer and receiver and save in variable (distant)
save first buffer as the (choose_buffer)
There is a buffer
False
Calculate the distant between the choose buffer and receiver save in variable (new distant)
If(new distant < distant and choose_buffer= buffer)
Choose_buffer= buffer-1
If(new distant > distant and choose_buffer≠ buffer)
choose buffer= buffer
End
Replace receiver buffer with choose_buffer
End
End

Figure 7: RWSB correction algorithm

The vector is employed to identify and rectify various defects, including a sender without a matching receiver, simultaneous reading and writing into the same buffer, in-order tag violations, DTM and message sizes mismatched (MSM), multiple messages sent to the same destination, receiving identical messages from the same source, any source (AS) or any tag (AT) discrepancies. The same approach is used in BPTP communication to detect and correct these defects.

Another defect in this scenario is the occurrence of lost requests, which happens when multiple MPI_Isend and MPI_Irecv calls utilize the same request variable within the same processor. It is crucial to recognize this situation, as it can lead to requests for overwriting [20]. The requests are modified to address this defect.

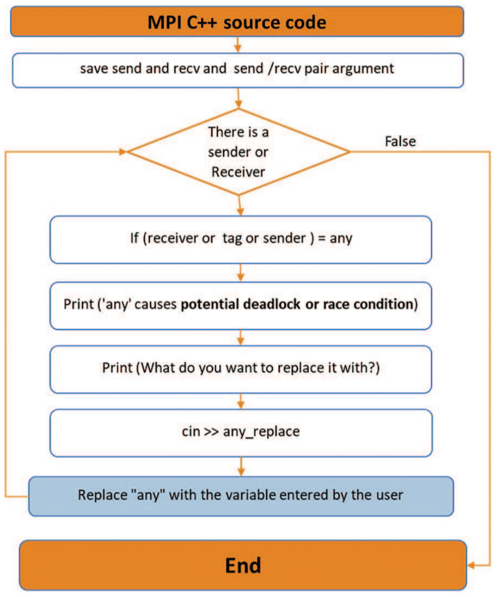

MPI C++ source code
save send and recv and send /recv pair argument
There is a sender or Receiver
False
If (receiver or tag or sender) = any
Print ('any' causes potential deadlock or race condition)
Print (What do you want to replace it with?)
cin >> any_replace
Replace "any" with the variable entered by the user
End

Figure 8: Wildcard correction algorithm

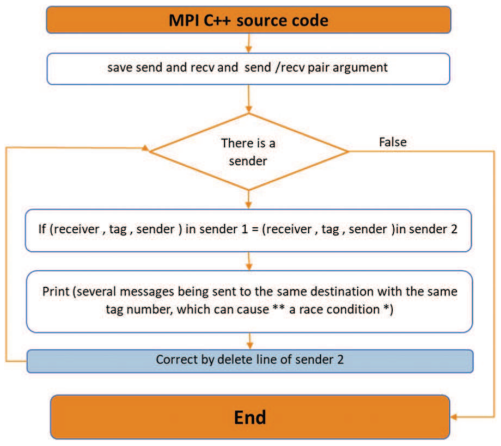

MPI C++ source code
save send and recv and send /recv pair argument
There is a sender
False
If (receiver , tag , sender) in sender 1 = (receiver , tag , sender)in sender 2
Print (several messages being sent to the same destination with the same tag number, which can cause ** a race condition *)
Correct by delete line of sender 2
End

Figure 9: Correction of several messages sent to the same destination

Furthermore, there is an additional defect related to missing MPI_Wait calls. In NBPTP communication, a RC can arise, particularly when operations need to be completed prior to sending or receiving. Therefore, it is imperative to utilize MPI_Wait calls to ensure the proper finalization of NBPTP communication [20]. This defect is resolved by storing the existing wait calls and selecting the associated call. If there are sending or receiving calls without a corresponding wait, the appropriate location for the wait call is identified and added, as illustrated in Fig. 10.

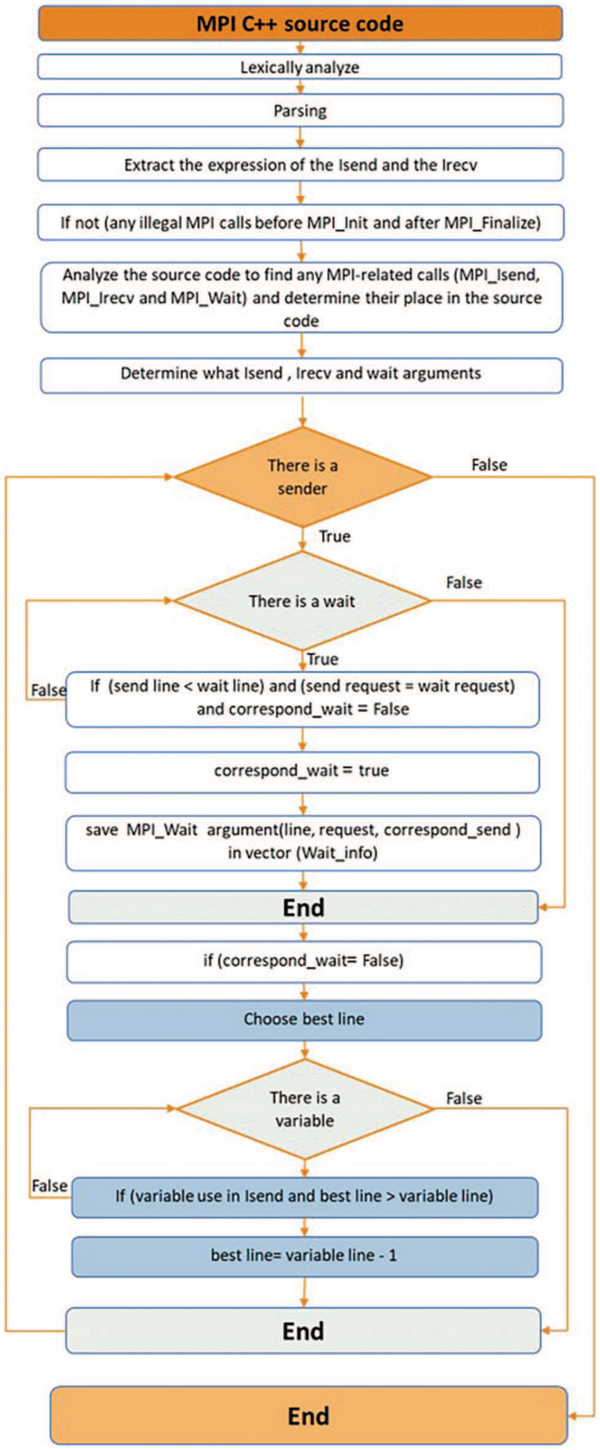

MPI C++ source code
Lexically analyze
Parsing
Extract the expression of the Isend and the Irecv
If not (any illegal MPI calls before MPI_Init and after MPI_Finalize)
Analyze the source code to find any MPI-related calls (MPI_Isend, MPI_Irecv and MPI_Wait) and determine their place in the source code
Determine what Isend , Irecv and wait arguments
There is a sender
False
True
There is a wait
False
True
False
If (send line < wait line) and (send request = wait request) and correspond_wait = False
correspond_wait = true
save MPI_Wait argument(line, request, correspond_send) in vector (Wait_info)
End
if (correspond_wait= False)
Choose best line
There is a variable
False
False
If (variable use in Isend and best line > variable line)
best line= variable line - 1
End
End

Figure 10: Correct missing MPI_Wait calls

Moreover, when a variable is modified before waiting, such as when buffer1 is altered after a sending call that uses it, there is a risk of the buffer1 value changing before the MPI_Wait function, leading to incorrect data transfer. To rectify this issue, the MPI_Wait call is moved to the line preceding the modification of buffer1, ensuring that the buffer1 value remains unchanged until the send operation is completed, as depicted in Fig. 11.

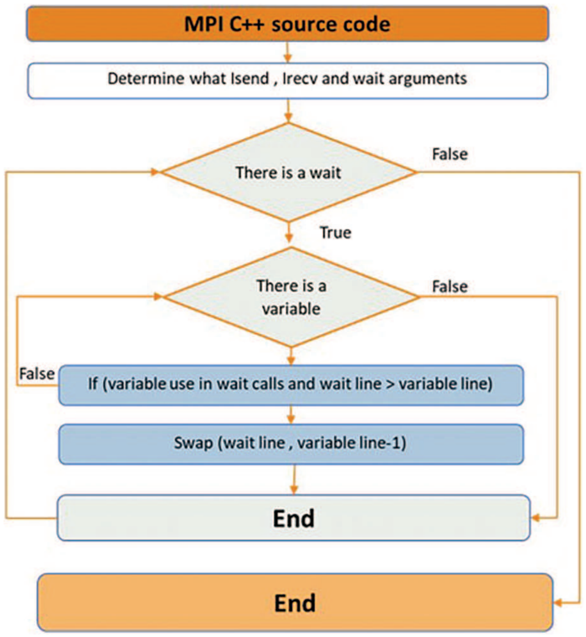

MPI C++ source code
Determine what Isend , Irecv and wait arguments
There is a wait
False
True
There is a variable
False
False
If (variable use in wait calls and wait line > variable line)
Swap (wait line , variable line-1)
End
End

Figure 11: Waiting happens after changing a variable

3.3 Detection and Correction of Software Defects in CC

DC_MPI also detects CC, such as MPI_Bcast. It ensures adherence to MPI call rules by parsing and lexically analyzing MPI calls. Unauthorized MPI calls made before MPI_Init and after MPI_Finalize are also detected.

Moreover, the DC_MPI analysis of the source code involves identifying MPI-related calls and determining their location, type, and arguments within the code. For MPI_Bcast, this includes details such as the sender's line number, buffer, count, type, root, communicator, and rank. Similarly, for barrier calls, it identifies the barrier line, MPI communicator, and rank. Vectors are then created to store the relevant MPI_Bcast and barrier arguments.

By utilizing this vector, defects such as reading and writing to the same buffer, DTM, MSM, AS and AT can be detected and corrected. This vector is also employed to identify and rectify similar defects in BPTP communication.

An example of a defect in CC is the incorrect ordering of collective operations within the same MPI communicator [21]. To address this issue, the MPI_Bcast and barrier lines are switched, as depicted in Fig. 12.

Furthermore, a potential DL may occur if not all processes in the MPI communicator invoke the MPI collective operation. An example of a situation that can lead to a DL is when two processes employ MPI_gather while the third process abstains from participating in the operation. To resolve this defect, all processes are searched for MPI_gather, and if it is not found, it is added to ensure proper execution.

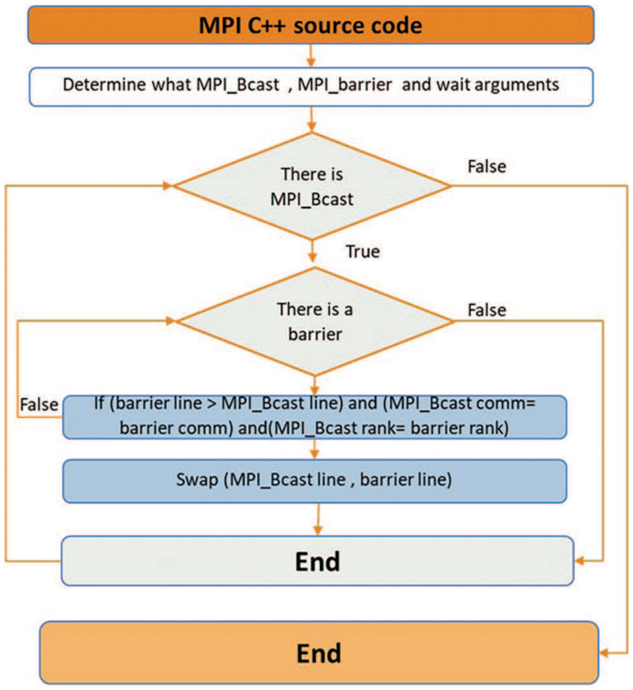

MPI C++ source code
Determine what MPI_Bcast , MPI_barrier and wait arguments
There is MPI_Bcast
False
True
There is a barrier
False
False
If (barrier line > MPI_Bcast line) and (MPI_Bcast comm= barrier comm) and(MPI_Bcast rank= barrier rank)
Swap (MPI_Bcast line , barrier line)
End
End

Figure 12: Corrected by swapping the MPI_Bcast and barrier lines

4 Results

In this section, we present the results achieved using a desktop computing setup comprising an Intel (R) i7-7700 HQ CPU operating at 2.80 Gigahertz, running on Windows 10 64-bit with 16 gigabytes of RAM. The system also incorporates a 256 gigabytes SSD (Solid State Drive) and a Graphics Processing Unit (GPU) known as Giga Texel Shader eXtreme (GTX). This GPU is a variant within the GeForce brand, which is owned by Nvidia and features the 1060 graphics card. The DC_MPI framework, implemented in C++, was utilized in the experiments.

In the upcoming sections, we will provide an overview of the database used in our study and present the results of the extract arguments process. We then analyze the outcomes in terms of detecting and correcting specific types of defects, including illegal MPI calls, DL, RC, and MM, with respect to each type of MPI communication. These findings offer valuable insights into the capabilities of the DC_MPI model in effectively identifying and resolving such issues. Ultimately, this research contributes to enhancing the reliability and performance of MPI programs.

4.1 Database Collection

We gathered data from a total of 40 MPI programs, including BPTP, NBPTP and CC. Among these programs, some contained defects while others did not [20]. The test database used to evaluate the DC_MPI model consists of 40 MPI programs that cover a range of defects, including illegal MPI calls, DL, RC, and MM. Additionally, the database includes six defect-free programs. Fig. 13 visualizes the distribution of defects across each program in the database.

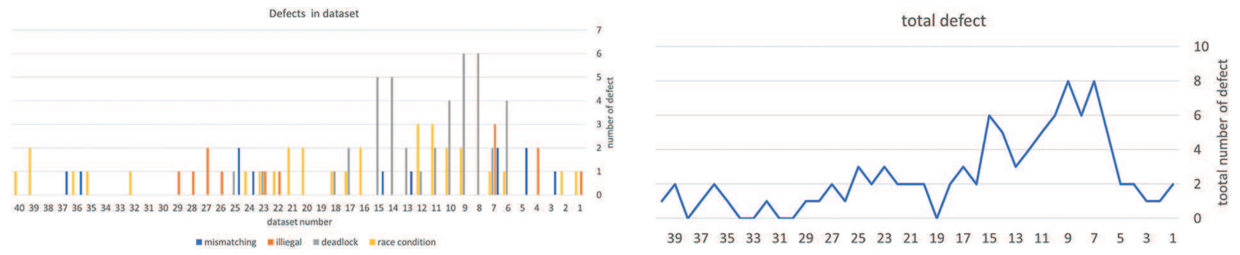
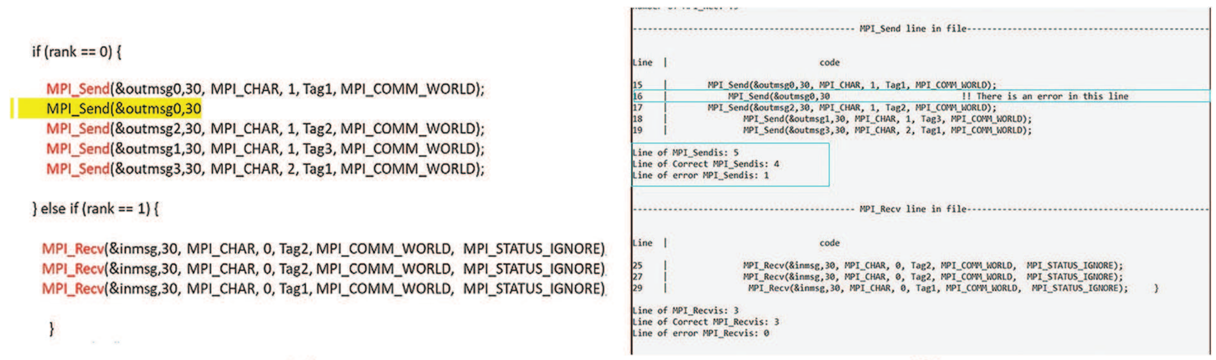


Figure 13: The number of defects

4.2 *Extract Arguments*

First, the MPI calls were lexically analyzed by DC_MPI, which then parsed them to identify any MPI-related calls and checked that they adhered to MPI call rules. For example, in the code in Fig. 14a, MPI calls were recognized, and the call that did not match the call rules was then discovered, as shown in Fig. 14b.



```
if (rank == 0) {

   MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
   MPI_Send(&outmsg0,30
   MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
   MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
   MPI_Send(&outmsg3,30, MPI_CHAR, 2, Tag1, MPI_COMM_WORLD);

} else if (rank == 1) {

  MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
  MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
  MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

   }
```

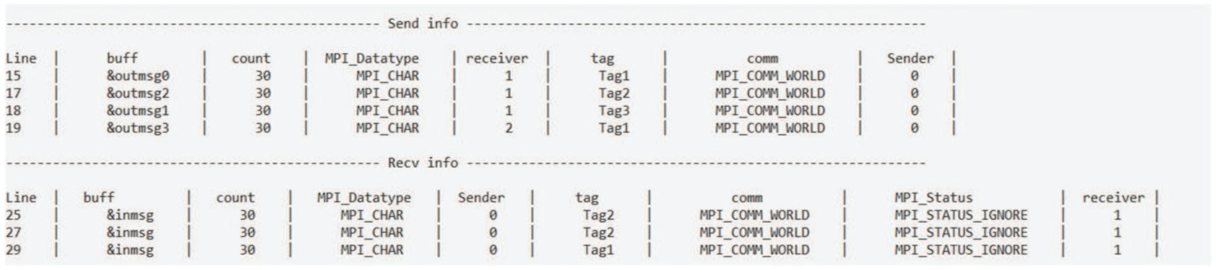
```
----------------------------------------- MPI_Send line in file-----------------------------------------

Line  |                    code

15    |       MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
16    |          MPI_Send(&outmsg0,30                        !! There is an error in this line
17    |       MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
18    |            MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
19    |            MPI_Send(&outmsg3,30, MPI_CHAR, 2, Tag1, MPI_COMM_WORLD);

Line of MPI_Sendis: 5
Line of Correct MPI_Sendis: 4
Line of error MPI_Sendis: 1

----------------------------------------- MPI_Recv line in file-----------------------------------------

Line  |                    code

25    |            MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
27    |            MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
29    |             MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);    }

Line of MPI_Recvis: 3
Line of Correct MPI_Recvis: 3
Line of error MPI_Recvis: 0
```

(a) (b)

Figure 14: (a) Part of the MPI code; (b) Find any MPI-related calls then ensures they follow MPI call rules

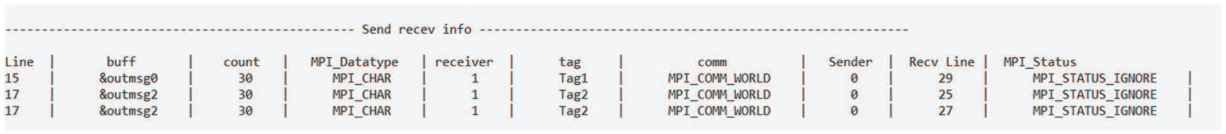
DC_MPI then proceeded to extract the arguments and establish multiple vectors to store the corresponding information. For BPTP communication, the arguments of the sender and receiver were extracted, this process is depicted in Fig. 15. DC_MPI constructs various vectors to hold the pertinent send and receive arguments. The pairings are then recognized, and each sender is associated with the appropriate receiver and recorded in a vector for use in defect detection, as shown in Fig. 16. The MPI_Sendrecv calls exhibit consistent behavior despite having varying structures, we examine and analyze them separately from the first two calls, but in a similar way. In this case, we modified the previous code and added MPI_Sendrecv in line 22. As a result, DC_MPI extracts the sender data and adds it to the send vector and extracts the receiver data and adds it to the receiver vector, as shown in Fig. 17.

Fig. 18a demonstrates that the MPI_Isend function in NBPTP communication includes sender arguments, waiting arguments, and the receiver arguments in MPI_Irecv. Then, as illustrated in Fig. 18b, it creates several vectors to store the arguments. The pairings are then recognized, and each sender is associated with the appropriate receiver and recorded in a vector for use in defect detection, as shown in Fig. 19.



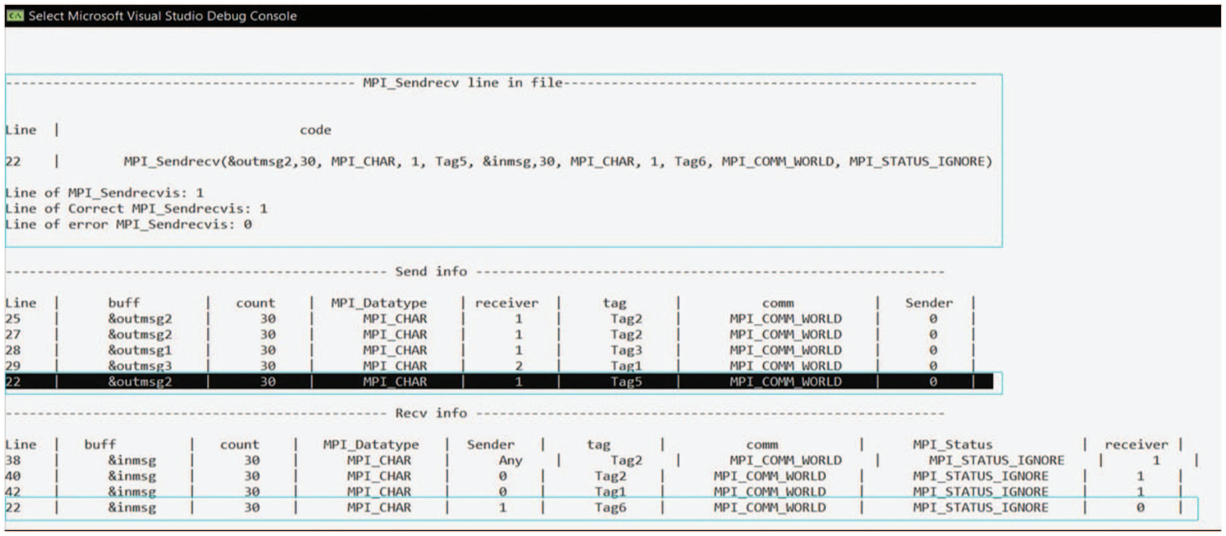
```
---------------------------------------------- Send info ---------------------------------------------------------

Line  |      buff      |  count  |  MPI_Datatype  | receiver |     tag     |        comm         |  Sender  |
15    |     &outmsg0   |    30   |     MPI_CHAR   |     1    |     Tag1    |   MPI_COMM_WORLD    |    0     |
17    |     &outmsg2   |    30   |     MPI_CHAR   |     1    |     Tag2    |   MPI_COMM_WORLD    |    0     |
18    |     &outmsg1   |    30   |     MPI_CHAR   |     1    |     Tag3    |   MPI_COMM_WORLD    |    0     |
19    |     &outmsg3   |    30   |     MPI_CHAR   |     2    |     Tag1    |   MPI_COMM_WORLD    |    0     |

---------------------------------------------- Recv info ---------------------------------------------------------

Line  |   buff        |  count  |  MPI_Datatype  |  Sender  |    tag    |       comm        |     MPI_Status      | receiver |
25    |      &inmsg   |    30   |    MPI_CHAR    |     0    |    Tag2   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    1     |
27    |      &inmsg   |    30   |    MPI_CHAR    |     0    |    Tag2   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    1     |
29    |      &inmsg   |    30   |    MPI_CHAR    |     0    |    Tag1   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    1     |
```

Figure 15: The results of extracting sending and receiving arguments in DC_MPI



```
-------------------------------------------- Send recev info ----------------------------------------------------

Line  |    buff      |  count  | MPI_Datatype | receiver |    tag    |       comm       |  Sender  | Recv Line |  MPI_Status
15    |   &outmsg0   |    30   |   MPI_CHAR   |     1    |   Tag1    |  MPI_COMM_WORLD  |    0     |     29    |     MPI_STATUS_IGNORE   |
17    |   &outmsg2   |    30   |   MPI_CHAR   |     1    |   Tag2    |  MPI_COMM_WORLD  |    0     |     25    |     MPI_STATUS_IGNORE   |
17    |   &outmsg2   |    30   |   MPI_CHAR   |     1    |   Tag2    |  MPI_COMM_WORLD  |    0     |     27    |     MPI_STATUS_IGNORE   |
```

Figure 16: The results of combining each sender with the corresponding receiver in DC_MPI



```
Select Microsoft Visual Studio Debug Console

-------------------------------------------- MPI_Sendrecv line in file-----------------------------------------------

Line  |                          code

22    |         MPI_Sendrecv(&outmsg2,30, MPI_CHAR, 1, Tag5, &inmsg,30, MPI_CHAR, 1, Tag6, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

Line of MPI_Sendrecvis: 1
Line of Correct MPI_Sendrecvis: 1
Line of error MPI_Sendrecvis: 0

---------------------------------------------- Send info ---------------------------------------------------------

Line  |      buff      |  count  |  MPI_Datatype  | receiver |     tag     |        comm         |  Sender  |
25    |     &outmsg2   |    30   |     MPI_CHAR   |     1    |     Tag2    |   MPI_COMM_WORLD    |    0     |
27    |     &outmsg2   |    30   |     MPI_CHAR   |     1    |     Tag2    |   MPI_COMM_WORLD    |    0     |
28    |     &outmsg1   |    30   |     MPI_CHAR   |     1    |     Tag3    |   MPI_COMM_WORLD    |    0     |
29    |     &outmsg3   |    30   |     MPI_CHAR   |     2    |     Tag1    |   MPI_COMM_WORLD    |    0     |
22    |     &outmsg2   |    30   |     MPI_CHAR   |     1    |     Tag5    |   MPI_COMM_WORLD    |    0     |

---------------------------------------------- Recv info ---------------------------------------------------------

Line  |   buff        |  count  |  MPI_Datatype  |  Sender  |    tag    |       comm        |     MPI_Status      | receiver |
38    |      &inmsg   |    30   |    MPI_CHAR    |    Any   |    Tag2   |   MPI_COMM_WORLD  |   MPI_STATUS_IGNORE  |    1     |
40    |      &inmsg   |    30   |    MPI_CHAR    |     0    |    Tag2   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    1     |
42    |      &inmsg   |    30   |    MPI_CHAR    |     0    |    Tag1   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    1     |
22    |      &inmsg   |    30   |    MPI_CHAR    |     1    |    Tag6   |  MPI_COMM_WORLD   |  MPI_STATUS_IGNORE  |    0     |
```

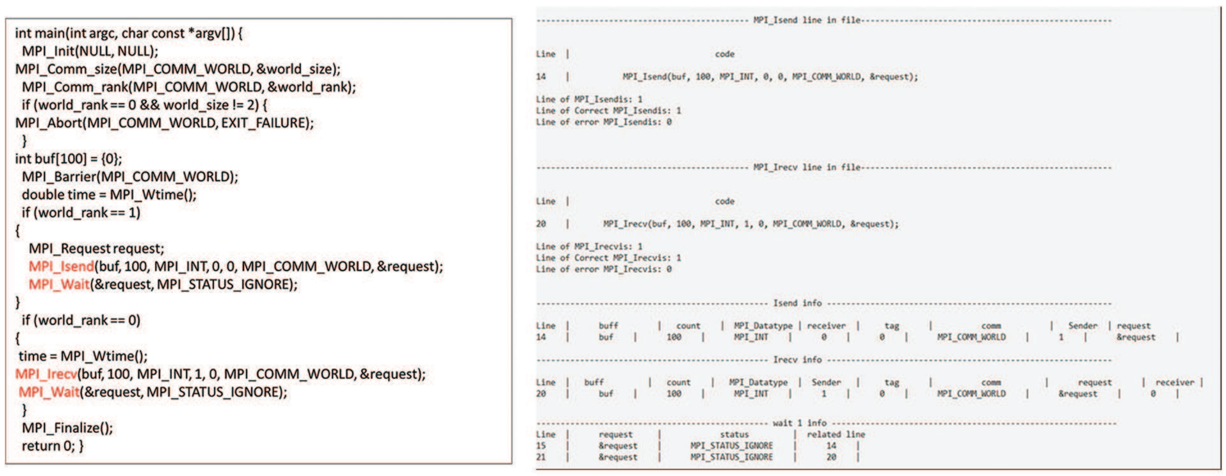
Figure 17: Analyzing MPI_Sendrecv calls and preserving it in the sender and receiver vectors

The MPI_Bcast arguments and barrier arguments are retrieved from the CC code, as depicted in Fig. 20a. Then, as demonstrated in Fig. 20b, it constructs several vectors for holding associated MPI_Bcast and barrier arguments.

4.3 Detecting and Correcting Illegal MPI Calls Defect

DC_MPI ensures that MPI calls adhere to the prescribed MPI call rules and actively scans for any unauthorized or illegal calls. As an illustration, in BPTP communication, MPI_Init and MPI_Finalize were introduced after the MPI function and before the MPI function, respectively, as illustrated in Fig. 21a. Fig. 21b displays the detection stage and shows how we discovered two illegal MPI calls. The corrected code is shown in Fig. 21c, where the order of the MPI_Init function was changed before the

MPI functions were utilized, and the order of the MPI_Finalize function was changed after the MPI functions.



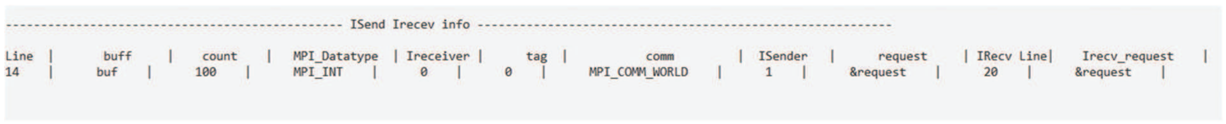
```
int main(int argc, char const *argv[]) {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0};
  MPI_Barrier(MPI_COMM_WORLD);
  double time = MPI_Wtime();
  if (world_rank == 1)
{
    MPI_Request request;
    MPI_Isend(buf, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
  if (world_rank == 0)
{
 time = MPI_Wtime();
MPI_Irecv(buf, 100, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  MPI_Finalize();
  return 0; }
```

(a)

```
------------------------------------------ MPI_Isend line in file-----------------------------------------------

Line  |                         code

14    |          MPI_Isend(buf, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

Line of MPI_Isendis: 1
Line of Correct MPI_Isendis: 1
Line of error MPI_Isendis: 0


------------------------------------------ MPI_Irecv line in file-----------------------------------------------

Line  |                         code

20    |       MPI_Irecv(buf, 100, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

Line of MPI_Irecvis: 1
Line of Correct MPI_Irecvis: 1
Line of error MPI_Irecvis: 0


------------------------------------------- Isend info -----------------------------------------------------

Line  |     buff      |  count    | MPI_Datatype | receiver |    tag     |        comm          |  Sender  | request
14    |     buf   |    100    |    MPI_INT   |     0    |    0    |    MPI_COMM_WORLD    |    1    |     &request    |
------------------------------------------- Irecv info -----------------------------------------------------

Line  |   buff        |   count    |  MPI_Datatype  | Sender   |    tag     |        comm         |     request       | receiver |
20    |     buf   |    100    |    MPI_INT   |     1    |    0    |    MPI_COMM_WORLD    |    &request    |    0    |

------------------------------------------- wait 1 info -----------------------------------------------------
Line  |     request    |          status          | related line
15    |     &request   |    MPI_STATUS_IGNORE     |     14     |
21    |     &request   |    MPI_STATUS_IGNORE     |     20     |
```

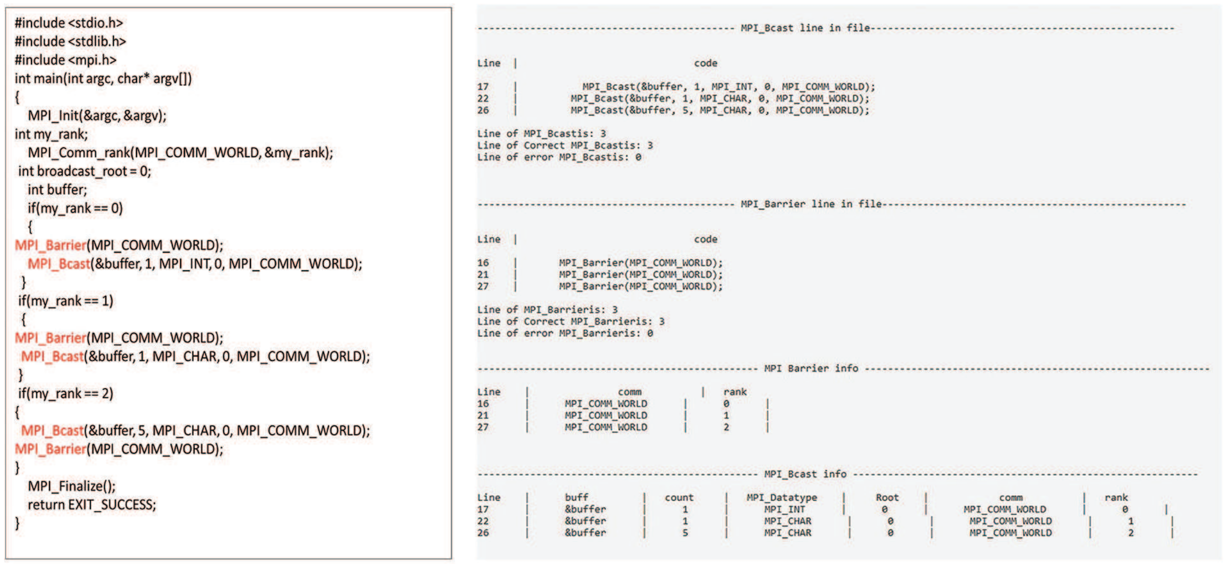
(b)

Figure 18: (a) The code for NBPTP communication; (b) Isend, Ireceive and wait arguments



```
---------------------------------------------- ISend Irecev info ---------------------------------------------------------

Line |       buff      |    count    | MPI_Datatype | Ireceiver |      tag  |         comm          | ISender   |      request    | IRecv Line|    Irecv_request    |
14   |       buf    |    100    |    MPI_INT    |    0    |    0    |    MPI_COMM_WORLD    |    1    |   &request    |    20    |    &request    |
```

Figure 19: Isend is associated with the appropriate Ireceive



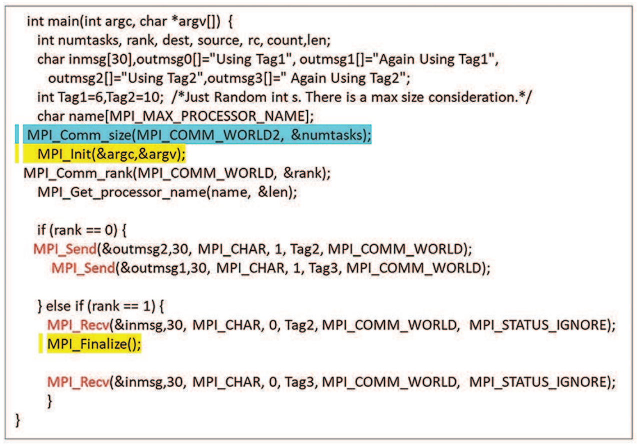
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 int broadcast_root = 0;
   int buffer;
   if(my_rank == 0)
   {
MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
 }
 if(my_rank == 1)
  {
MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
 }
 if(my_rank == 2)
{
 MPI_Bcast(&buffer, 5, MPI_CHAR, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
}
   MPI_Finalize();
   return EXIT_SUCCESS;
}
```

(a)

```
------------------------------------------ MPI_Bcast line in file-----------------------------------------------

Line  |                         code

17    |          MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
22    |        MPI_Bcast(&buffer, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
26    |        MPI_Bcast(&buffer, 5, MPI_CHAR, 0, MPI_COMM_WORLD);

Line of MPI_Bcastis: 3
Line of Correct MPI_Bcastis: 3
Line of error MPI_Bcastis: 0


------------------------------------------ MPI_Barrier line in file-----------------------------------------------

Line  |                         code

16    |      MPI_Barrier(MPI_COMM_WORLD);
21    |      MPI_Barrier(MPI_COMM_WORLD);
27    |      MPI_Barrier(MPI_COMM_WORLD);

Line of MPI_Barrieris: 3
Line of Correct MPI_Barrieris: 3
Line of error MPI_Barrieris: 0

-------------------------------------------- MPI Barrier info ---------------------------------------------------

Line    |         comm          |   rank
16      |     MPI_COMM_WORLD    |    0      |
21      |     MPI_COMM_WORLD    |    1      |
27      |     MPI_COMM_WORLD    |    2      |


-------------------------------------------- MPI_Bcast info ---------------------------------------------------

Line    |     buff       |   count    |  MPI_Datatype   |    Root   |        comm         |   rank
17      |     &buffer    |     1      |    MPI_INT      |    0      |   MPI_COMM_WORLD    |    0     |
22      |     &buffer    |     1      |    MPI_CHAR      |    0      |   MPI_COMM_WORLD    |    1     |
26      |     &buffer    |     5      |    MPI_CHAR      |    0      |   MPI_COMM_WORLD    |    2     |
```

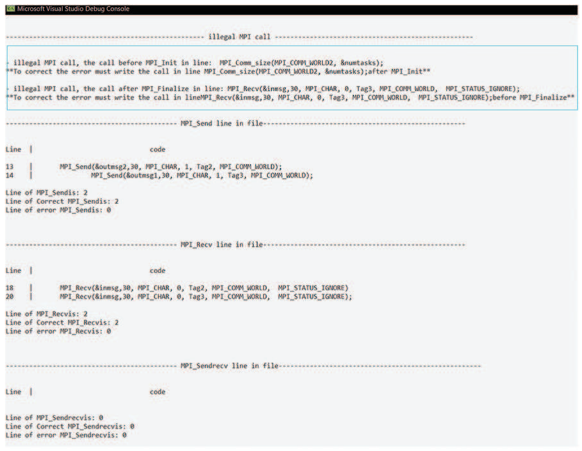
(b)

Figure 20: (a) The CC code; (b) MPI_Bcast and barrier arguments



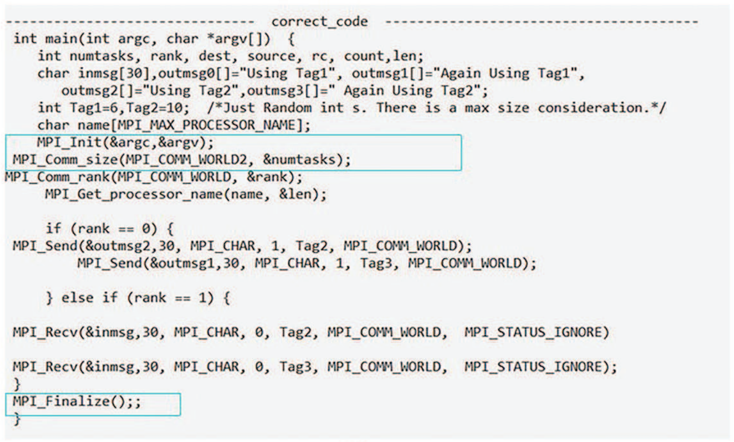
```
int main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count,len;
  char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
    outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
  int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
  char name[MPI_MAX_PROCESSOR_NAME];
MPI_Comm_size(MPI_COMM_WORLD2, &numtasks);
  MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(name, &len);

  if (rank == 0) {
 MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
    MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);

  } else if (rank == 1) {
    MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
    MPI_Finalize();

    MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
    }
}
```

(a)



```
------------------------------------------------ illegal MPI call ------------------------------------------------

 illegal MPI call, the call before MPI_Init in line:  MPI_Comm_size(MPI_COMM_WORLD2, &numtasks);
**To correct the error must write the call in line MPI_Comm_size(MPI_COMM_WORLD2, &numtasks);after MPI_Init**

 illegal MPI call, the call after MPI_Finalize in line: MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
**To correct the error must write the call in lineMPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);before MPI_Finalize**

-------------------------------------------- MPI_Send line in file--------------------------------------------------

Line  |                         code

13    |      MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
14    |              MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);

Line of MPI_Sendis: 2
Line of Correct MPI_Sendis: 2
Line of error MPI_Sendis: 0


-------------------------------------------- MPI_Recv line in file--------------------------------------------------

Line  |                         code

18    |      MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE)
20    |      MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);

Line of MPI_Recvis: 2
Line of Correct MPI_Recvis: 2
Line of error MPI_Recvis: 0


-------------------------------------------- MPI_Sendrecv line in file----------------------------------------------

Line  |                         code


Line of MPI_Sendrecvis: 0
Line of Correct MPI_Sendrecvis: 0
Line of error MPI_Sendrecvis: 0
```

(b)



```
------------------------------  correct_code  --------------------------------------
 int main(int argc, char *argv[])  {
    int numtasks, rank, dest, source, rc, count,len;
    char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
       outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
    int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD2, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Get_processor_name(name, &len);

     if (rank == 0) {
 MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
         MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);

     } else if (rank == 1) {

 MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE)

 MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
 }
 MPI_Finalize();;
 }
```

(c)

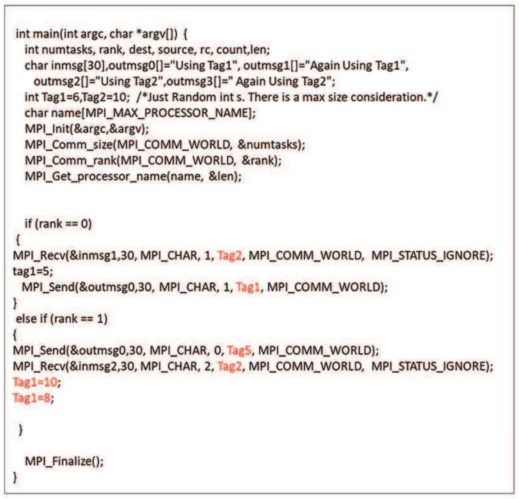
Figure 21: (a) Code with illegal calls; (b) Detection stage and discovered two illegal MPI calls; (c) Corrected code

4.4 DL and RC Defects

In this section, we examine several cases that have been identified and rectified, involving DL and RC defects in MPI programs. These cases include sender without corresponding receiver (SWCR) and receiver without corresponding sender (RWCS), incorrect tag sequence (ITS), AS and AT (wildcard), multiple messages are sent to the same destination or received from the same sender, MPI_Wait Defects (MWD), incorrectly ordered collective operations (IOCO), and RWSB.

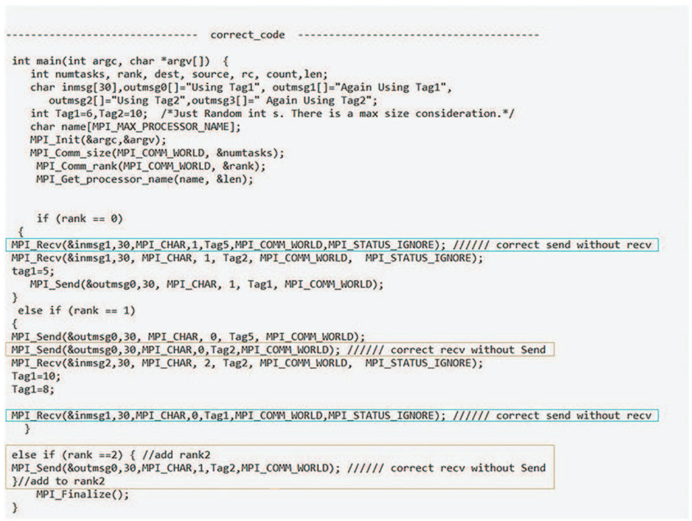
4.4.1 SWCR and RWCS

The number of receivers and senders is compared to see if a comparable sender or receiver exists. The defect of there being a sender without a matching receiver, leading to RC, can be corrected by attaching a corresponding receiver to the sender. First, the best line is selected to add the receiver's call so that all the variables used in the receiver's call are defined. Then, the receiver's call is added after the last change in the variable's value. Next, the optimal buffer is determined, which was previously utilized. Similarly, the appropriate sender is added to the receiver's call. An example of this is the code shown in Fig. 22a: processor 0 contains a call to receiver tag 2 without a corresponding sender and a call to send tag 1 without a receiver for it. Processor 1 contains a call to send tag 5 without a receiver for it and a call to send tag 2 without a corresponding sender. In Fig. 22b, a corresponding receiver was added to receive tag 5 in processor 0, and a corresponding receiver was added to receive tag 1 in processor 1. We note that receiver tag 1 uses the variable tag 1, and the value of the variable was changed in processor 1, so the receiver call was added after changing the value of tag 1.



```
int main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count,len;
  char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
    outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
  int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(name, &len);


  if (rank == 0)
 {
MPI_Recv(&inmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
tag1=5;
 MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
}
 else if (rank == 1)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 0, Tag5, MPI_COMM_WORLD);
MPI_Recv(&inmsg2,30, MPI_CHAR, 2, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
Tag1=10;
Tag1=8;

 }

  MPI_Finalize();
}
```

(a)

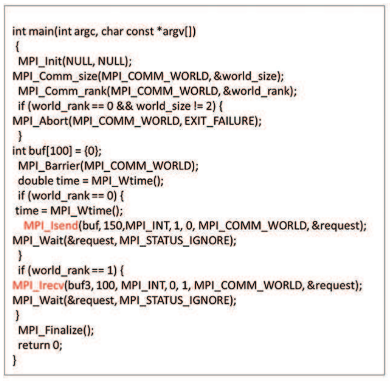


```
------------------------------  correct_code  ---------------------------------------

int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
      outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
   int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);


    if (rank == 0)
  {
MPI_Recv(&inmsg1,30,MPI_CHAR,1,Tag5,MPI_COMM_WORLD,MPI_STATUS_IGNORE); ////// correct send without recv
MPI_Recv(&inmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
tag1=5;
   MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
}
 else if (rank == 1)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 0, Tag5, MPI_COMM_WORLD);
MPI_Send(&outmsg0,30,MPI_CHAR,0,Tag2,MPI_COMM_WORLD); ////// correct recv without Send
MPI_Recv(&inmsg2,30, MPI_CHAR, 2, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
Tag1=10;
Tag1=8;

MPI_Recv(&inmsg1,30,MPI_CHAR,0,Tag1,MPI_COMM_WORLD,MPI_STATUS_IGNORE); ////// correct send without recv
  }

else if (rank ==2) { //add rank2
MPI_Send(&outmsg0,30,MPI_CHAR,1,Tag2,MPI_COMM_WORLD); ////// correct recv without Send
}//add to rank2
    MPI_Finalize();
}
```

(b)

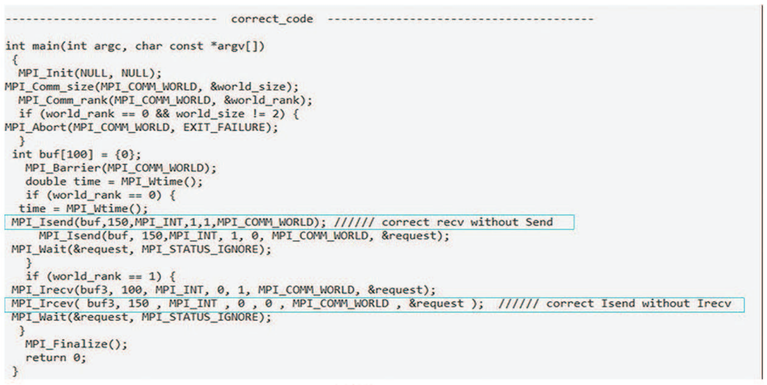
Figure 22: (a) A defect in which there is a sender but no matching receiver, and vice versa; (b) Corrected code

On the other hand, if there are more receivers than senders, it can potentially result in a DL scenario where all receivers are waiting for messages they are not receiving. A corresponding sender has been added to send tag 2 in processor 1. We also note that processor 2 has been defined, and a corresponding sender has been added to send tag 2. Figs. 23a and 23b illustrate how to correct this defect in NBPTP communication.



```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0};
  MPI_Barrier(MPI_COMM_WORLD);
  double time = MPI_Wtime();
  if (world_rank == 0) {
 time = MPI_Wtime();
   MPI_Isend(buf, 150,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1) {
MPI_Irecv(buf3, 100, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
}
```

(a)



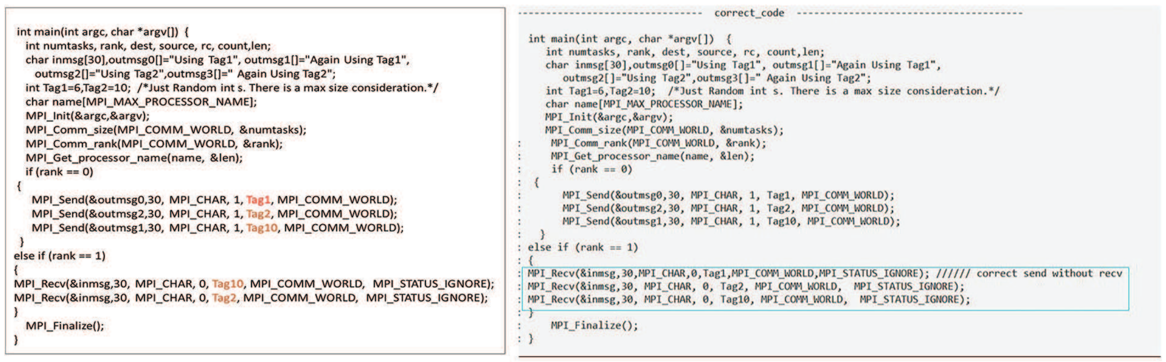
```
------------------------------  correct_code  --------------------------------------

int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
 int buf[100] = {0};
   MPI_Barrier(MPI_COMM_WORLD);
   double time = MPI_Wtime();
   if (world_rank == 0) {
  time = MPI_Wtime();
 MPI_Isend(buf,150,MPI_INT,1,1,MPI_COMM_WORLD); ////// correct recv without Send
     MPI_Isend(buf, 150,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
   }
   if (world_rank == 1) {
 MPI_Irecv(buf3, 100, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);
 MPI_Ircev( buf3, 150 , MPI_INT , 0 , 0 , MPI_COMM_WORLD , &request );  ////// correct Isend without Irecv
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
   return 0;
 }
```

(b)

Figure 23: (a) The defect of having an Isend without a corresponding Irecv and vice versa; (b) Corrected code

4.4.2 ITS

ITS leads to either an actual or potential DL. If the line of the first sender is smaller than the line of the second sender, the line of the first receiver must also be smaller than that of the second receiver. If this is not the case, the first receiver and the second receiver are swapped. For example, processor 0 is defined and sends data to processor 1 with tags 1, 2, and 10. We note that processor 1 receives tags 10 and then 2, and at the same time, a receive call for tag 1 was not created, as shown in Fig. 24a. We note that by correcting the code, processor 1 can receive tag 1. The order of receiving calls is then changed so that tags 1, 2, and 10 are received in that order, as shown in Fig. 24b. Figs. 25a and 25b depict this defect in NBPTP communication.



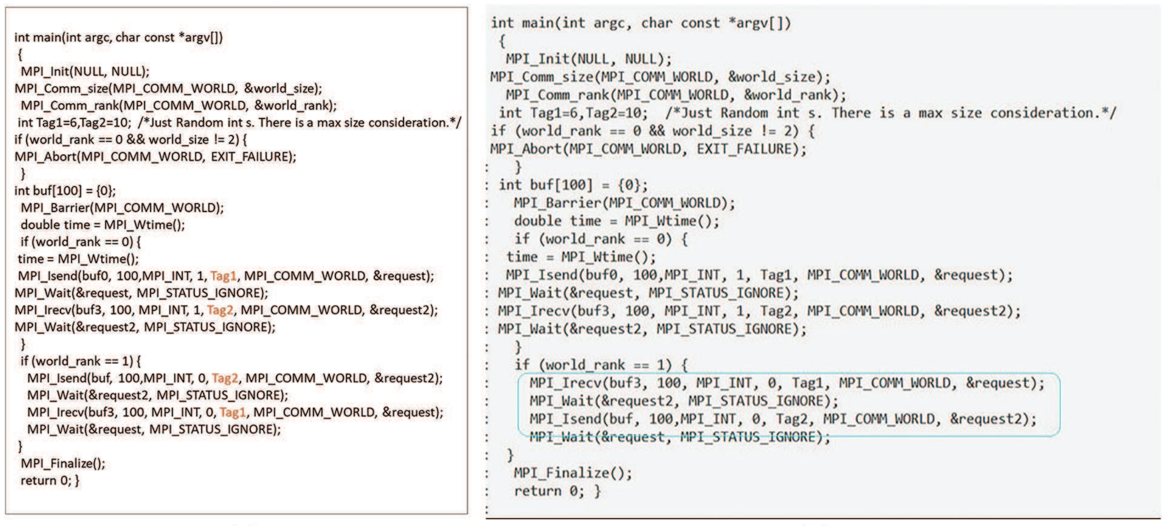
```
int main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count,len;
  char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
    outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
  int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(name, &len);
  if (rank == 0)
{
   MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
   MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
   MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag10, MPI_COMM_WORLD);
 }
else if (rank == 1)
{
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag10, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
}
  MPI_Finalize();
}
```

(a)

```
------------------------------- correct_code ---------------------------------------

int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
      outmsg2[]="Using Tag2",outmsg3[]=" Again Using Tag2";
   int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
:   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
:   MPI_Get_processor_name(name, &len);
:   if (rank == 0)
:  {
:     MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
:     MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
:     MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag10, MPI_COMM_WORLD);
:   }
: else if (rank == 1)
: {
: MPI_Recv(&inmsg,30,MPI_CHAR,0,Tag1,MPI_COMM_WORLD,MPI_STATUS_IGNORE); ////// correct send without recv
: MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
: MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag10, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
: }
:    MPI_Finalize();
: }
```

(b)

Figure 24: (a) The sequence of the tags is incorrect in BPTP communication; (b) Corrected tag order



```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0};
  MPI_Barrier(MPI_COMM_WORLD);
  double time = MPI_Wtime();
  if (world_rank == 0) {
 time = MPI_Wtime();
 MPI_Isend(buf0, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
MPI_Irecv(buf3, 100, MPI_INT, 1, Tag2, MPI_COMM_WORLD, &request2);
MPI_Wait(&request2, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1) {
    MPI_Isend(buf, 100,MPI_INT, 0, Tag2, MPI_COMM_WORLD, &request2);
    MPI_Wait(&request2, MPI_STATUS_IGNORE);
    MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
  MPI_Finalize();
  return 0; }
```

(a)

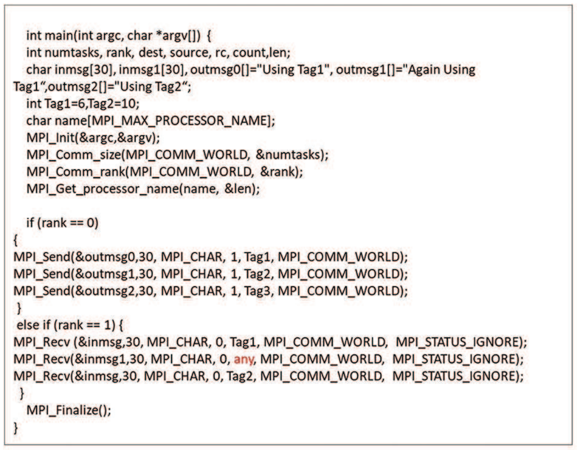
```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int Tag1=6,Tag2=10;  /*Just Random int s. There is a max size consideration.*/
if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
:   }
: int buf[100] = {0};
:   MPI_Barrier(MPI_COMM_WORLD);
:   double time = MPI_Wtime();
:   if (world_rank == 0) {
:  time = MPI_Wtime();
:  MPI_Isend(buf0, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
: MPI_Wait(&request, MPI_STATUS_IGNORE);
: MPI_Irecv(buf3, 100, MPI_INT, 1, Tag2, MPI_COMM_WORLD, &request2);
: MPI_Wait(&request2, MPI_STATUS_IGNORE);
:   }
:   if (world_rank == 1) {
:     MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
:     MPI_Wait(&request2, MPI_STATUS_IGNORE);
:     MPI_Isend(buf, 100,MPI_INT, 0, Tag2, MPI_COMM_WORLD, &request2);
:     MPI_Wait(&request, MPI_STATUS_IGNORE);
:  }
:   MPI_Finalize();
:   return 0; }
```

(b)

Figure 25: (a) The sequence of the tags is incorrect in NBPTP communication; (b) Corrected tag order

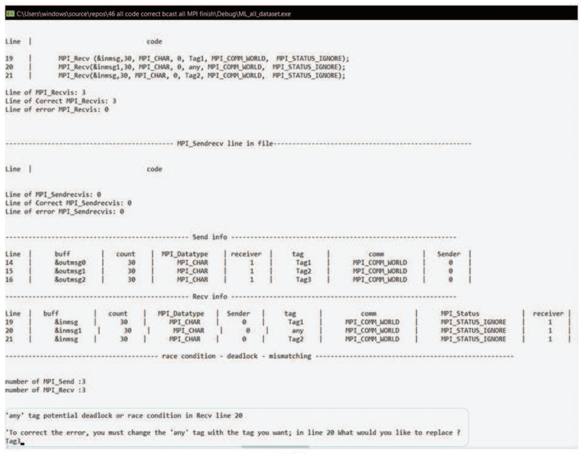
4.4.3 The AS and AT

The "AS" or "AT" (wildcard) is corrected by demanding that the user enter the right tag or source rather than any. As shown in Fig. 26a, tag 1 will be sent and received correctly, but when tag 2 is sent and received by the receiver of "AT," a DL will result where both the sender of tag 3 and the receiver of tag 2 will stay in a waiting state. This defect was found and corrected by asking the user which tag they wished to use instead of the "AT," as shown in Fig. 26b. Then "any" was replaced with the tag entered by the user. We note tag 2 was sent before tag 3. The DC_MPI changed the sequence of receive calls for tags 2 and 3, as depicted in Fig. 26c. Fig. 27a depicts AS defects in NBPTP communication, while Fig. 27b depicts the correct code.



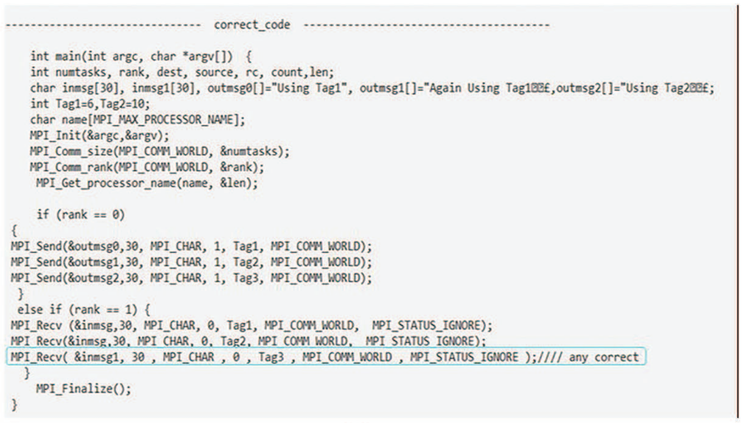
```
   int main(int argc, char *argv[]) {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using
Tag1",outmsg2[]="Using Tag2";
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);

   if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg1,30, MPI_CHAR, 0, any, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
 }
   MPI_Finalize();
}
```

(a)



```
C:\Users\windows\source\repos\46 all code correct bcast all MPI finish\Debug\ML_all_dataset.exe

Line  |                         code

19    |      MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
20    |      MPI_Recv(&inmsg1,30, MPI_CHAR, 0, any, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
21    |      MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);

Line of MPI_Recvis: 3
Line of Correct MPI_Recvis: 3
Line of error MPI_Recvis: 0


-------------------------------------------- MPI_Sendrecv line in file---------------------------------------------------

Line  |                         code


Line of MPI_Sendrecvis: 0
Line of Correct MPI_Sendrecvis: 0
Line of error MPI_Sendrecvis: 0


----------------------------------------------- Send info ----------------------------------------------------------

Line  |     buff      |   count   |  MPI_Datatype  | receiver |    tag    |        comm        |  Sender  |
14    |     &outmsg0  |     30    |     MPI_CHAR   |     1    |    Tag1   |   MPI_COMM_WORLD   |     0    |
15    |     &outmsg1  |     30    |     MPI_CHAR   |     1    |    Tag2   |   MPI_COMM_WORLD   |     0    |
16    |     &outmsg2  |     30    |     MPI_CHAR   |     1    |    Tag3   |   MPI_COMM_WORLD   |     0    |
----------------------------------------------- Recv info ----------------------------------------------------------
Line  |  buff        |   count   |  MPI_Datatype  |  Sender  |    tag    |       comm       |      MPI_Status       | receiver |
19    |     &inmsg   |    30     |    MPI_CHAR    |    0     |    Tag1   |  MPI_COMM_WORLD  |  MPI_STATUS_IGNORE    |    1     |
20    |     &inmsg1   |    30    |     MPI_CHAR    |     0    |    any    |  MPI_COMM_WORLD  |  MPI_STATUS_IGNORE    |    1     |
21    |     &inmsg   |    30     |    MPI_CHAR    |    0     |    Tag2   |  MPI_COMM_WORLD  |  MPI_STATUS_IGNORE    |    1     |
-------------------------------------------- race condition - deadlock - mismatching ----------------------------------------------------------


number of MPI_Send :3
number of MPI_Recv :3

'any' tag potential deadlock or race condition in Recv line 20

'To correct the error, you must change the 'any' tag with the tag you want; in line 20 What would you like to replace ?
Tag3
```

(b)



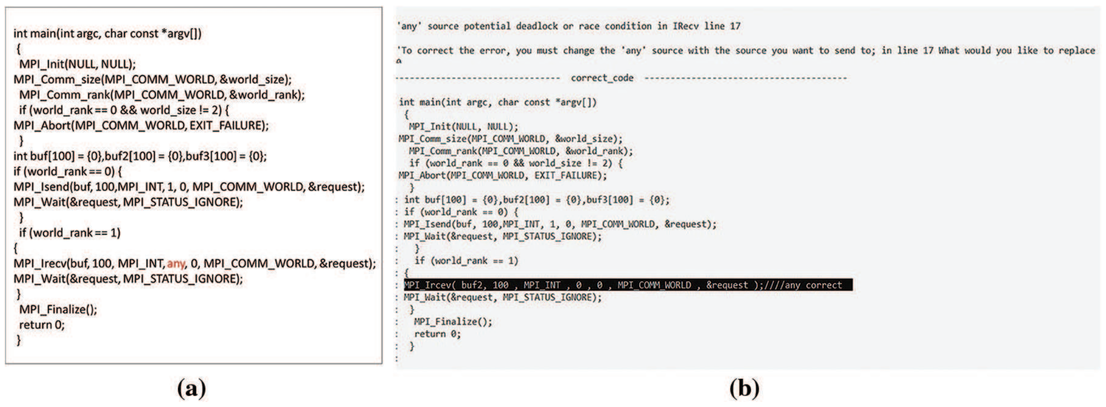
```
------------------------------  correct_code  -------------------------------------

   int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1��£,outmsg2[]="Using Tag2��£;
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    if (rank == 0)
 {
 MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
 MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
 MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
 MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
 MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
 MPI_Recv( &inmsg1, 30 , MPI_CHAR , 0 , Tag3 , MPI_COMM_WORLD , MPI_STATUS_IGNORE );//// any correct
   }
    MPI_Finalize();
 }
```

(c)

Figure 26: (a) The BPTP communication code contains wildcard; (b) Detecting the defect and requesting the user to enter an alternative; (c) The code after correcting the defect

4.4.4 *Several Messages Send to the Same Destination or Received from the Same Sender*

Fig. 28a illustrates that a potential DL or RC can arise when multiple messages are sent to a shared destination or received from a common sender with identical tag numbers. Fig. 28b demonstrates that this defect is addressed by removing the second call when there is no corresponding response. Conversely, if a corresponding call exists, as depicted in Fig. 29a, it is not removed, as illustrated in Fig. 29b. In the context of NBPTP communication, Fig. 30a exhibits the scenario where multiple messages are received from the same sender. However, Fig. 30b portrays the corrected code, addressing this issue.



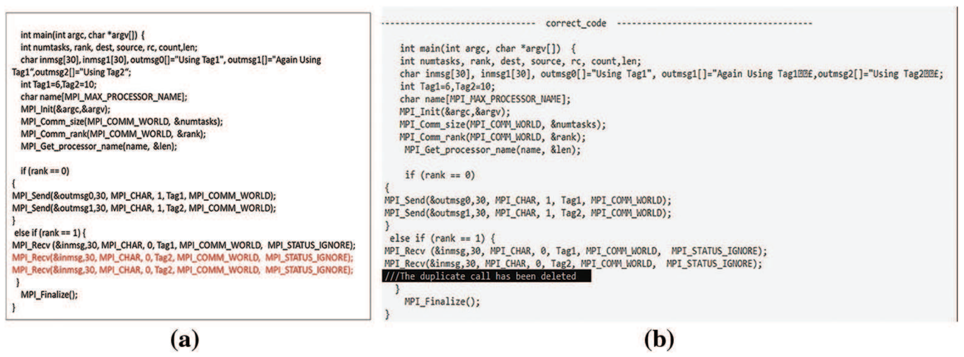
```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1)
{
MPI_Irecv(buf, 100, MPI_INT, any, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
 }
```

(a)

```
'any' source potential deadlock or race condition in IRecv line 17

'To correct the error, you must change the 'any' source with the source you want to send to; in line 17 What would you like to replace
a
------------------------------  correct_code  -------------------------------------

 int main(int argc, char const *argv[])
  {
   MPI_Init(NULL, NULL);
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
   if (world_rank == 0 && world_size != 2) {
 MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }
: int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
: if (world_rank == 0) {
: MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
: MPI_Wait(&request, MPI_STATUS_IGNORE);
:   }
:   if (world_rank == 1)
: {
: MPI_Ircev( buf2, 100 , MPI_INT , 0 , 0 , MPI_COMM_WORLD , &request );////any correct
: MPI_Wait(&request, MPI_STATUS_IGNORE);
:  }
:   MPI_Finalize();
:   return 0;
:  }
:
```

(b)

Figure 27: (a) A wildcard is present in the NBPTP communication code; (b) Detecting the defect, requesting the user to enter an alternative, and then correcting



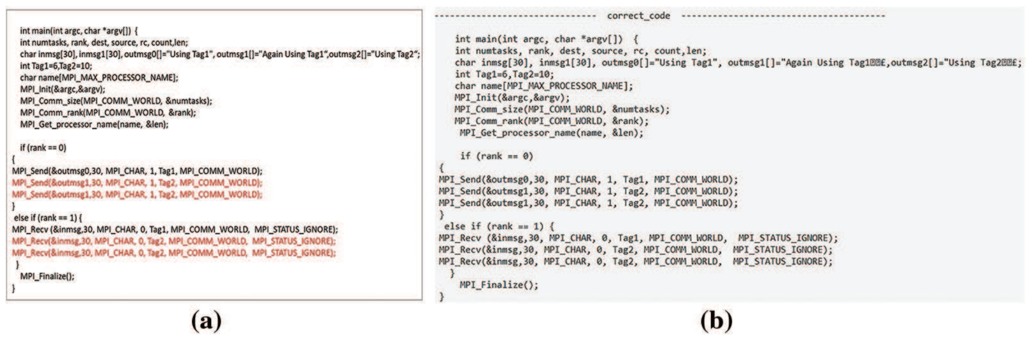
```
   int main(int argc, char *argv[]) {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using
Tag1",outmsg2[]="Using Tag2";
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);

   if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
}
 else if (rank == 1) {
MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
}
```

(a)

```
------------------------------  correct_code  -------------------------------------

   int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1��£,outmsg2[]="Using Tag2��£;
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
}
 else if (rank == 1) {
MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
///The duplicate call has been deleted
  }
    MPI_Finalize();
}
```

(b)

Figure 28: (a) Several messages with the same tag number were received from the same sender; (b) Corrected code by removing the second call



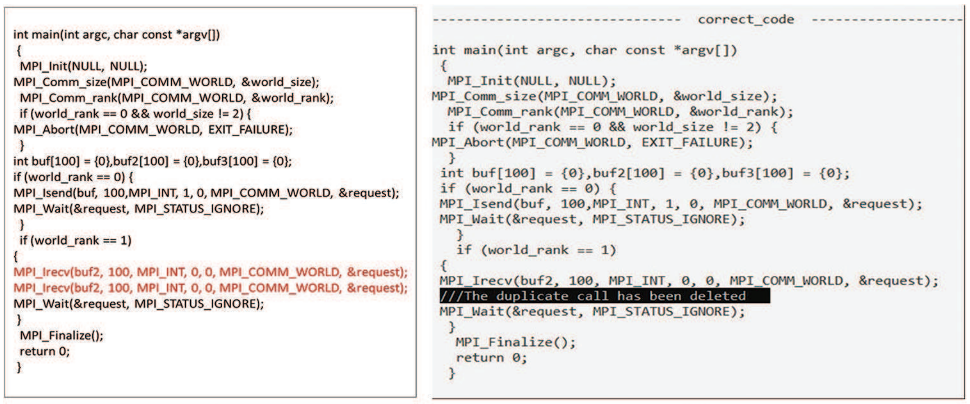
```
   int main(int argc, char *argv[]) {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",outmsg2[]="Using Tag2";
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);

   if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
}
 else if (rank == 1) {
MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
}
```

(a)

```
------------------------------  correct_code  -------------------------------------

   int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1��£,outmsg2[]="Using Tag2��£;
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
}
 else if (rank == 1) {
MPI_Recv (&inmsg,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
  }
    MPI_Finalize();
}
```

(b)

Figure 29: (a) If there is a repeat call but it has a corresponding call; (b) The code does not change



```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1)
{
MPI_Irecv(buf2, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
MPI_Irecv(buf2, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
 }
```

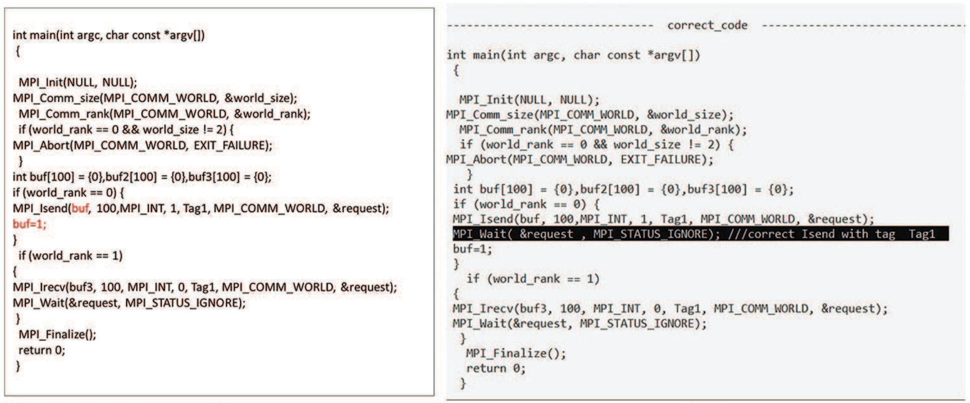
```
------------------------------  correct_code  ------------------

int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
 int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
   }
   if (world_rank == 1)
 {
 MPI_Irecv(buf2, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
 ///The duplicate call has been deleted
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
   return 0;
  }
```

(a) (b)

Figure 30: (a) Multiple messages received from the same sender in NBPTP communication; (b) Correcting code

4.4.5 *MWD*

The MPI_Wait calls are essential to complete NBPTP communication. This defect is corrected by storing the existing wait calls and then selecting the call associated with them. As shown in Fig. 31a, if a sending or receiving call does not contain a wait, the suitable position for the wait is chosen and added, as shown in Fig. 31b.



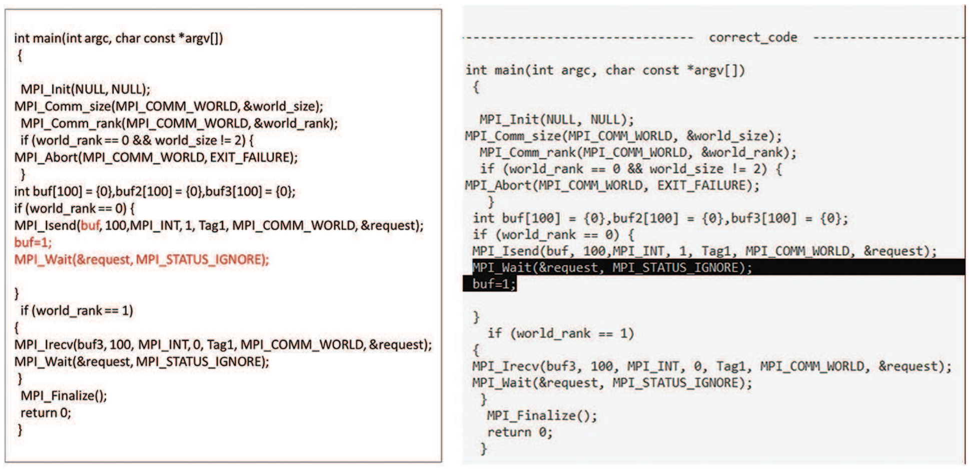
```
int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
buf=1;
}
  if (world_rank == 1)
{
MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
 }
```

```
------------------------------  correct_code  ------------------------------

int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }
 int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait( &request , MPI_STATUS_IGNORE); ///correct Isend with tag  Tag1
 buf=1;
 }
   if (world_rank == 1)
 {
 MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
   return 0;
  }
```

(a) (b)

Figure 31: (a) Sending or receiving call that does not have a wait call; (b) Corrected by add waiting call

Additionally, when waiting happens after changing a variable, such as when the value of the buffer changes, as shown in Fig. 32a, the waiting position changes to the line before the variable changes, as shown in Fig. 32b. If there is a wait function without an associated call, as in Fig. 33a, it is corrected by deleting the wait call, as shown in Fig. 33b. This approach helps avoid potential issues such as DL that can arise when NBPTP communication requests are not properly completed before subsequent program actions depend on their results.



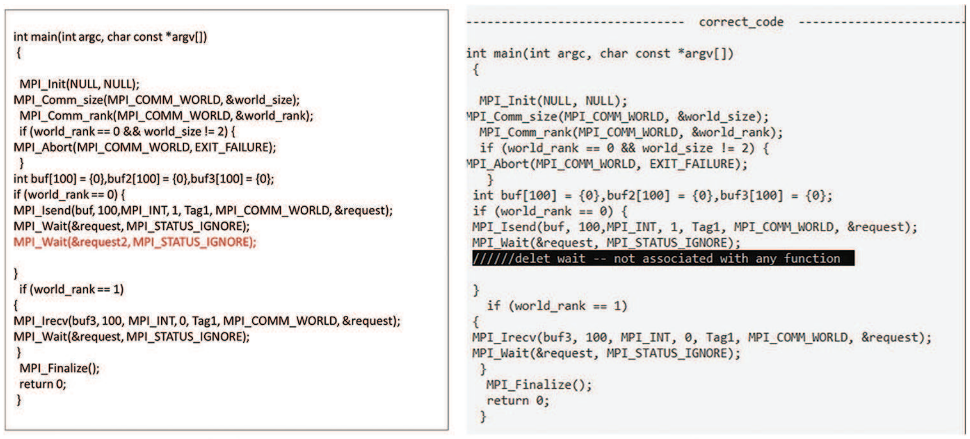
```
int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
 }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
buf=1;
MPI_Wait(&request, MPI_STATUS_IGNORE);

}
 if (world_rank == 1)
{
MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
}
 MPI_Finalize();
 return 0;
}
```

(a)

```
------------------------------- correct_code --------------------
int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }
 int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
 buf=1;

 }
   if (world_rank == 1)
 {
 MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
   return 0;
  }
```

(b)

Figure 32: (a) Change buffer value before waiting call; (b) Corrected by changing the position of waiting call



```
int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
 }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
MPI_Wait(&request2, MPI_STATUS_IGNORE);

}
 if (world_rank == 1)
{
MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
}
 MPI_Finalize();
 return 0;
}
```

(a)

```
----------------------------- correct_code ----------------------
int main(int argc, char const *argv[])
 {

  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }
 int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
 //////delet wait -- not associated with any function

 }
   if (world_rank == 1)
 {
 MPI_Irecv(buf3, 100, MPI_INT, 0, Tag1, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();
   return 0;
  }
```

(b)

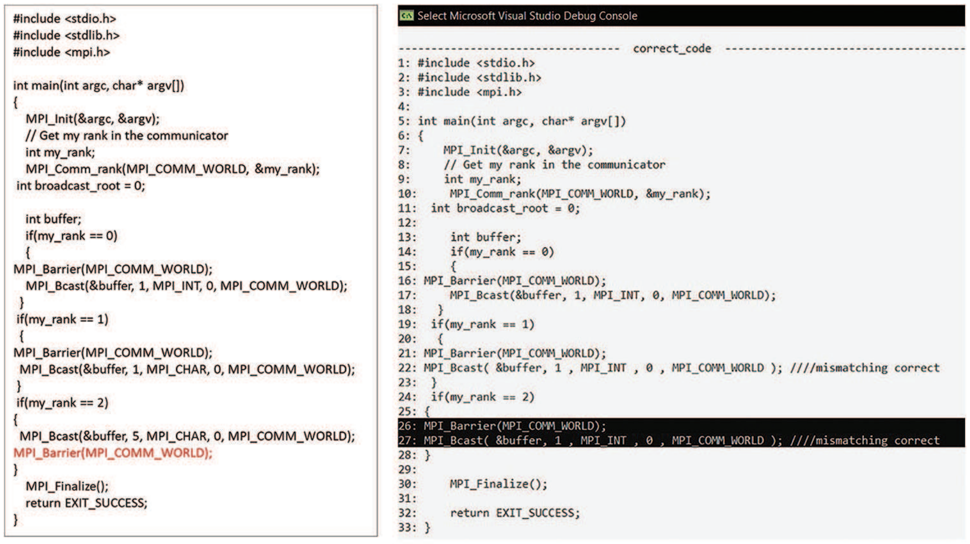
Figure 33: (a) Wait function without an associated call; (b) Corrected by deletion of waiting call

4.4.6 IOCO

In a collective communicator, incorrect ordering of collective operations may happen Fig. 34a. This was fixed by swapping MPI_Bcast and barrier Fig. 34b. DL can occur if not all processes call the MPI collective operation as seen in Fig. 35. This was resolved by adding the missing collective operation.

4.4.7 RWSB

Reading and writing same MPI buffer caused RC. Resolved by selecting most recently used buffer. In Fig. 36a, outmsg1 changed before being sent. Detected and corrected in DC_MPI by replacing buffer with inmsg Fig. 36b. Figs. 37a and 37b show NBPTP communication defect.



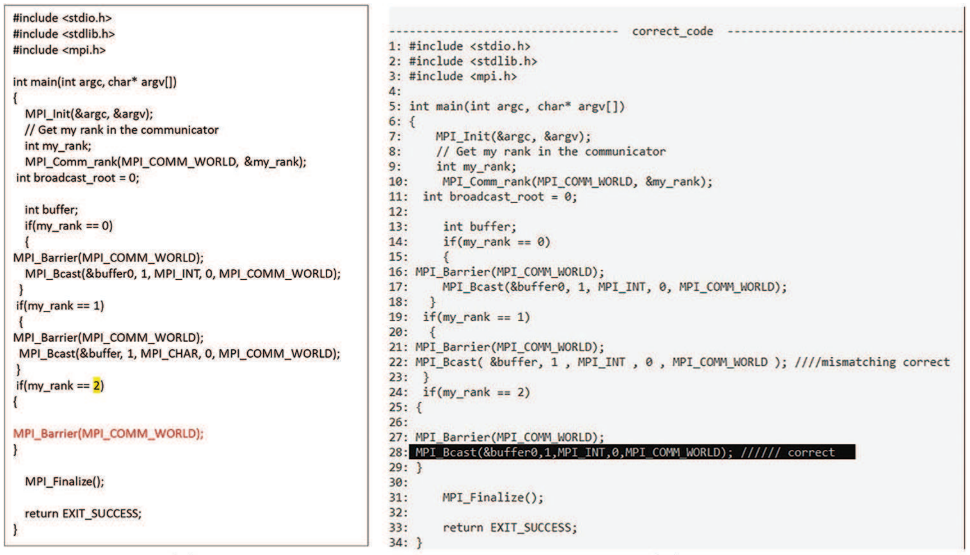
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
   // Get my rank in the communicator
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 int broadcast_root = 0;

   int buffer;
   if(my_rank == 0)
   {
MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
 }
 if(my_rank == 1)
 {
MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
}
 if(my_rank == 2)
{
 MPI_Bcast(&buffer, 5, MPI_CHAR, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
}
   MPI_Finalize();
   return EXIT_SUCCESS;
}
```

Select Microsoft Visual Studio Debug Console

```
---------------------------------  correct_code  ------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    // Get my rank in the communicator
    int my_rank;
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  int broadcast_root = 0;

     int buffer;
     if(my_rank == 0)
     {
MPI_Barrier(MPI_COMM_WORLD);
     MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
   }
  if(my_rank == 1)
   {
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast( &buffer, 1 , MPI_INT , 0 , MPI_COMM_WORLD ); ////mismatching correct
 }
  if(my_rank == 2)
{
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast( &buffer, 1 , MPI_INT , 0 , MPI_COMM_WORLD ); ////mismatching correct
}

     MPI_Finalize();

     return EXIT_SUCCESS;
}
```

(a) (b)

Figure 34: (a) Collective operations inside the same MPI communicator were arranged incorrectly; (b) Corrected by swapping the MPI_Bcast and barrier



```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
   // Get my rank in the communicator
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 int broadcast_root = 0;

   int buffer;
   if(my_rank == 0)
   {
MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&buffer0, 1, MPI_INT, 0, MPI_COMM_WORLD);
 }
 if(my_rank == 1)
 {
MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
}
 if(my_rank == 2)
{

MPI_Barrier(MPI_COMM_WORLD);
}

   MPI_Finalize();

   return EXIT_SUCCESS;
}
```

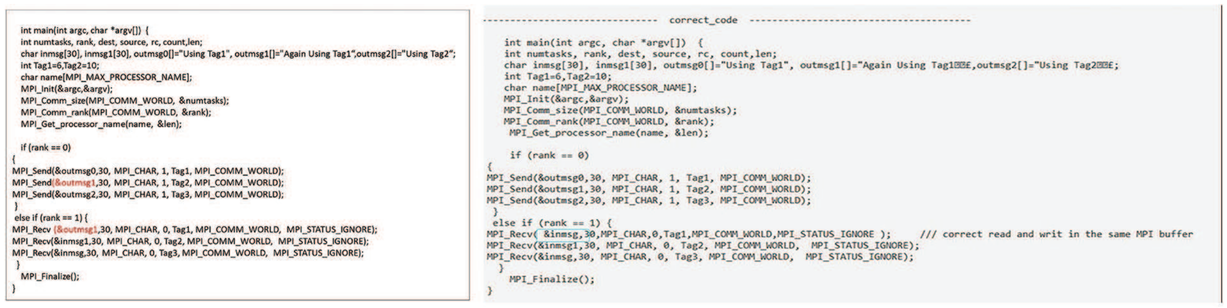
```
---------------------------------  correct_code  ------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    // Get my rank in the communicator
    int my_rank;
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  int broadcast_root = 0;

     int buffer;
     if(my_rank == 0)
     {
MPI_Barrier(MPI_COMM_WORLD);
     MPI_Bcast(&buffer0, 1, MPI_INT, 0, MPI_COMM_WORLD);
   }
  if(my_rank == 1)
   {
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast( &buffer, 1 , MPI_INT , 0 , MPI_COMM_WORLD ); ////mismatching correct
 }
  if(my_rank == 2)
{

MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer0,1,MPI_INT,0,MPI_COMM_WORLD); ////// correct
}

     MPI_Finalize();

     return EXIT_SUCCESS;
}
```

(a) (b)

Figure 35: (a) In situations where the MPI collective operation is not invoked by all processes, (b) Corrected by searching for it in all processes and adding it

4.5 DTM and MSM Defects

The DC_MPI tool corrects the defect of DTM and MSM by adjusting the size and type based on the size and type of the sender. Fig. 38a shows the BPTP communication code containing a defect discovered and corrected by DC_MPI. The first sender contains a DTM; the sender's data type is single character (MPI_CHAR), and the receiver's is integers (MPI_INT). In the correction phase, the receiver's type was changed to MPI_CHAR, as shown in Fig. 38b. The second sender also contains a MSM; the sender's data size is 60, and the receiver's size is 30. Fig. 38b illustrates the receiver's size adjusted to 60 during the correction phase.



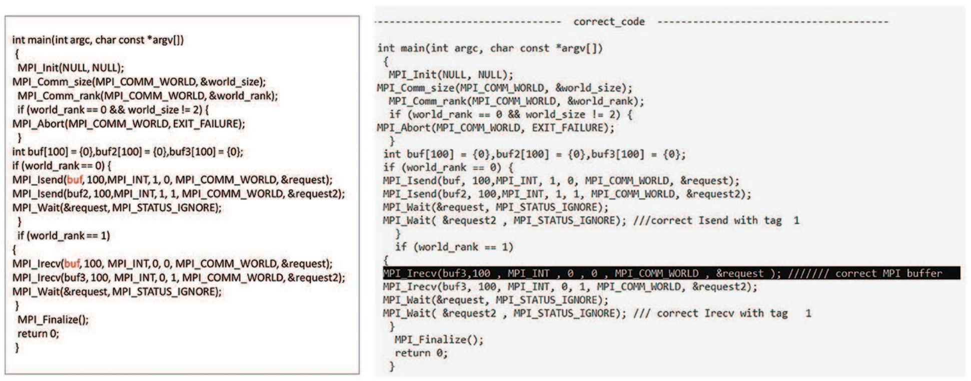
```
  int main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count,len;
  char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",outmsg2[]="Using Tag2";
  int Tag1=6,Tag2=10;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(name, &len);

  if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
MPI_Recv (&outmsg1,30, MPI_CHAR, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg1,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
}
```

(a)

```
------------------------------  correct_code  --------------------------------------

   int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30], inmsg1[30], outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1▯▯£,outmsg2[]="Using Tag2▯▯£;
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,30, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
MPI_Send(&outmsg2,30, MPI_CHAR, 1, Tag3, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
MPI_Recv( &inmsg,30,MPI_CHAR,0,Tag1,MPI_COMM_WORLD,MPI_STATUS_IGNORE );     /// correct read and writ in the same MPI buffer
MPI_Recv(&inmsg1,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag3, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
  }
    MPI_Finalize();
}
```

(b)

Figure 36: (a) The code uses the same MPI buffer for both reading and writing; (b) It corrected a reading and writing defect in an MPI buffer



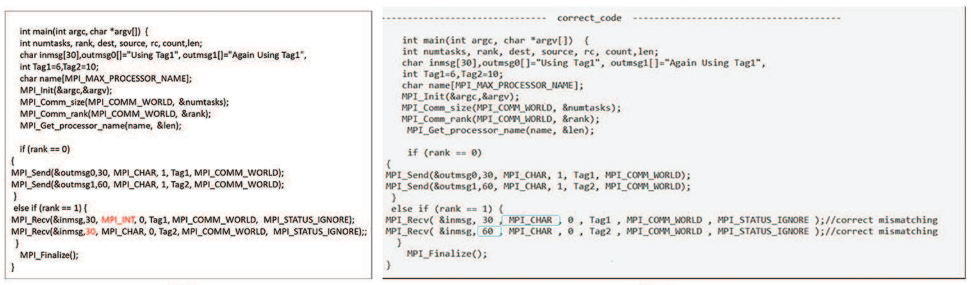
```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Isend(buf2, 100,MPI_INT, 1, 1, MPI_COMM_WORLD, &request2);
MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1)
{
MPI_Irecv(buf, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
MPI_Irecv(buf3, 100, MPI_INT, 0, 1, MPI_COMM_WORLD, &request2);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
 }
```

(a)

```
------------------------------  correct_code  --------------------------------------

int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
 int buf[100] = {0},buf2[100] = {0},buf3[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
 MPI_Isend(buf2, 100,MPI_INT, 1, 1, MPI_COMM_WORLD, &request2);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
 MPI_Wait( &request2 , MPI_STATUS_IGNORE); ///correct Isend with tag  1
   }
   if (world_rank == 1)
 {
MPI_Irecv(buf3,100 , MPI_INT , 0 , 0 , MPI_COMM_WORLD , &request ); ////// correct MPI buffer
 MPI_Irecv(buf3, 100, MPI_INT, 0, 1, MPI_COMM_WORLD, &request2);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
 MPI_Wait( &request2 , MPI_STATUS_IGNORE); /// correct Irecv with tag   1
  }
   MPI_Finalize();
   return 0;
  }
```

(b)

Figure 37: (a) The NBPTP communication code includes reading and writing to the same buffer; (b) Corrected code



```
  int main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count,len;
  char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
  int Tag1=6,Tag2=10;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(name, &len);

  if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,60, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
MPI_Recv(&inmsg,30, MPI_INT, 0, Tag1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
MPI_Recv(&inmsg,30, MPI_CHAR, 0, Tag2, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);;
 }
  MPI_Finalize();
}
```

(a)

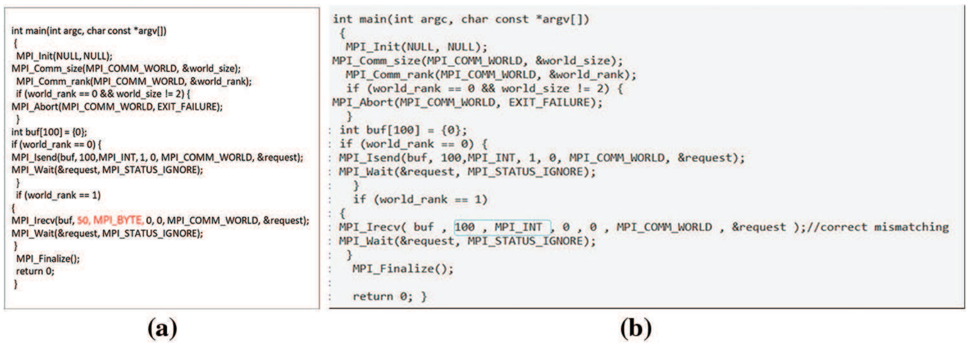
```
------------------------------  correct_code  --------------------------------------

   int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count,len;
   char inmsg[30],outmsg0[]="Using Tag1", outmsg1[]="Again Using Tag1",
   int Tag1=6,Tag2=10;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    if (rank == 0)
{
MPI_Send(&outmsg0,30, MPI_CHAR, 1, Tag1, MPI_COMM_WORLD);
MPI_Send(&outmsg1,60, MPI_CHAR, 1, Tag2, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
MPI_Recv( &inmsg, 30 , MPI_CHAR , 0 , Tag1 , MPI_COMM_WORLD , MPI_STATUS_IGNORE );//correct mismatching
MPI_Recv( &inmsg, 60 , MPI_CHAR , 0 , Tag2 , MPI_COMM_WORLD , MPI_STATUS_IGNORE );//correct mismatching
  }
    MPI_Finalize();
}
```

(b)

Figure 38: (a) The BPTP communication code contains MM; (b) Corrected MM

Fig. 39a shows the detection and correction of a defect in the NBPTP communication's code, where the sender has a DTM and MSM. The sender's data type is MPI_INT and the receiver's is

MPI_BYTE; the sender's data size is 100, and the receiver's is 50. In the correct phase, the receiver's data type was changed to MPI_INT and the size to 100, as shown in Fig. 39b.



(a)

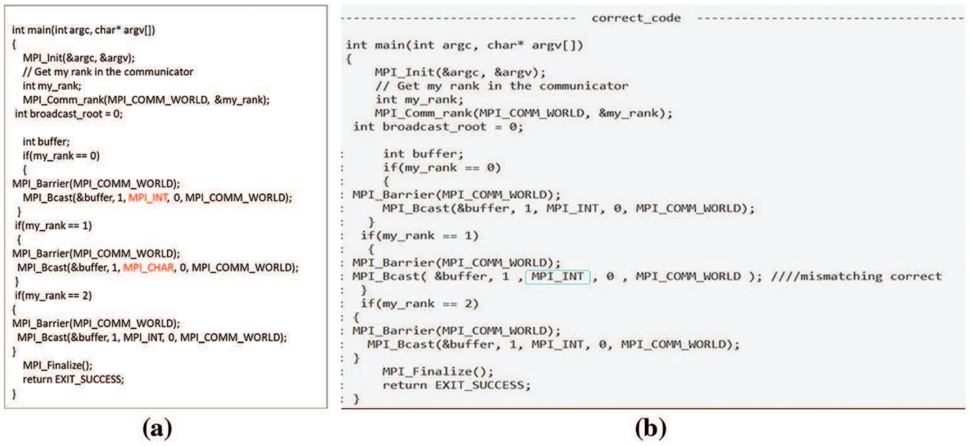
```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
int buf[100] = {0};
if (world_rank == 0) {
MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  if (world_rank == 1)
{
MPI_Irecv(buf, 50, MPI_BYTE, 0, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
  MPI_Finalize();
  return 0;
 }
```

(b)

```
int main(int argc, char const *argv[])
 {
  MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  if (world_rank == 0 && world_size != 2) {
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
 int buf[100] = {0};
 if (world_rank == 0) {
 MPI_Isend(buf, 100,MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait(&request, MPI_STATUS_IGNORE);
   }
   if (world_rank == 1)
 {
 MPI_Irecv( buf , 100 , MPI_INT , 0 , 0 , MPI_COMM_WORLD , &request );//correct mismatching
 MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
   MPI_Finalize();

   return 0; }
```

Figure 39: (a) The NBPTP communication code contains MM; (b) Corrected MM

In Fig. 40a, the code for CC has a DTM, where the root data type is MPI_INT, and the receiver's is MPI_CHAR; the receiver's type is changed to MPI_INT, as shown in Fig. 40b.



(a)

```
int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
   // Get my rank in the communicator
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 int broadcast_root = 0;

   int buffer;
   if(my_rank == 0)
   {
MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
 if(my_rank == 1)
  {
MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
 }
 if(my_rank == 2)
{
MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
   MPI_Finalize();
   return EXIT_SUCCESS;
}
```

(b)

```
-------------------------------  correct_code  ----------------------------------

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    // Get my rank in the communicator
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 int broadcast_root = 0;

     int buffer;
     if(my_rank == 0)
     {
 MPI_Barrier(MPI_COMM_WORLD);
     MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
   }
  if(my_rank == 1)
   {
 MPI_Barrier(MPI_COMM_WORLD);
 MPI_Bcast( &buffer, 1 , MPI_INT , 0 , MPI_COMM_WORLD ); ////mismatching correct
  }
  if(my_rank == 2)
 {
 MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
 }
     MPI_Finalize();
     return EXIT_SUCCESS;
 }
```

Figure 40: (a) The CC code contains MM; (b) Corrected MM

5 Discussion

Our model has been built and tested for verification and validation. To verify our suggested method and ensure DC_MPI's capacity for detecting illegal MPI calls, DL, RC, and MM defects in MPI programs, several tests have been carried out, encompassing a variety of test scenarios.

Due to the absence of MPI program codes, we used all MPI communication and gathered 40 MPI codes, some of which had defects, and some did not. The database incorporated both BPTP and NBPTP communications. BPTP communication refers to the situation where the process waits for the message data to satisfy a particular condition before it can continue. NBPTP calls start communication, and the programmer must use MPI_Wait to check the progress of data transmission and the success of the call afterward. Additionally, the database includes CC, which must include all procedures relevant to a communicator.

The DC_MPI model was introduced to detect and correct defects in both BPTP and NBPTP communication, along with CC. The defects include illegal MPI calls, DL, RC, and MM. The evaluation of the DC_MPI model demonstrates its effectiveness in defect detection and correction. The model successfully detected 37 out of 40 MPI codes tested, revealing its high accuracy. Furthermore, the program execution duration ranged between 0.81 and 1.36 s, indicating efficient and timely defect correction.

5.1 Detection Accuracy

The DC_MPI model displayed high accuracy in detecting defects, correctly identifying 37 out of the 40 MPI codes tested, translating to an overall detection accuracy of 92.5%. The model displayed a high detection rate, effectively pinpointing errors in the MPI-based code.

5.2 Correction Effectiveness

The DC_MPI model demonstrated remarkable capabilities in automatically correcting detected defects in MPI communication. The model accurately identified the defects and applied appropriate corrective actions, resulting in the resolution of a substantial number of defects without manual intervention.

5.3 Scalability

The scalability of the DC_MPI model refers to its ability to handle larger and more complex datasets while maintaining its performance. Although the evaluation was conducted on a dataset of 40 MPI code examples, the model's underlying architecture and design principles allow for scalability to larger datasets. As the dataset size grows, the model can be trained on more diverse examples, enabling it to generalize and detect defects in a wider range of scenarios.

5.4 Reduction in Manual Effort

The automatic detection and correction capabilities of the DC_MPI model significantly reduced the manual effort required by programmers for defect resolution. Programmers were relieved from the burden of manually identifying and fixing defects, leading to improved productivity and reduced development time.

5.5 Defect Coverage

The DC_MPI model effectively detected and corrected defects in various aspects of MPI communication, including BPTP, NBPTP and CC. Defect types such as illegal MPI calls, DL, RC, and MM were successfully addressed by the model, providing comprehensive coverage.

5.6 Program Execution Duration

The evaluation recorded program execution durations ranging between 0.81 and 1.36 s. These execution times indicate that the DC_MPI model is capable of handling defects efficiently within an acceptable timeframe. The model's ability to achieve timely defect detection and correction contributes to improved software development productivity.

The evaluation results highlight the effectiveness of the DC_MPI model in detecting and correcting defects. The model's high accuracy in detecting 37 out of 40 error codes indicates its robustness and reliability. Additionally, the model's efficient program execution durations demonstrate its ability to handle defects within reasonable time constraints.

Overall, the evaluation confirms that the DC_MPI model is a suitable tool for detecting and correcting defects in MPI-based systems. Its high detection accuracy, comprehensive error code coverage, and efficient defect handling make it a promising solution for enhancing code quality and reducing manual effort in defect resolution.

Table 1 compares the error coverage achieved by previous work with our DC_MPI system. Our system detects and corrects illegal MPI calls, DL, RC, and MM. Previous works discovered these defects but did not correct the defects.

Table 1: Summarizes the comparative study between previous work and our DC_MPI system

Ref.	Detected defects				Correct defect
	Illegal MPI call	DL	RC	MM	
In [8]	✓	✓	×	✓	×
In [9]	×	✓	✓	✓	×
In [10]	✓	✓	✓	×	×
In [11]	✓	✓	×	×	×
In [12]	✓	✓	×	×	×
In [13]	×	✓	✓	×	×
In [14]	×	✓	✓	✓	×
In [15]	×	✓	×	×	×
In [16]	×	✓	×	×	×
In [17]	✓	✓	✓	×	×
In [18]	✓	✓	✓	✓	×
In [19]	×	✓	×	×	×
DC_MPI	✓	✓	✓	✓	✓

From a programmer's perspective, integrating the DC_MPI model into existing MPI libraries offers several advantages in terms of defect detection and correction. The benefits of integration include:

- Enhanced Defect Detection: By integrating the DC_MPI model into the MPI library, programmers can benefit from advanced defect detection capabilities. The model can identify a wide range of defects, such as illegal MPI calls, DL, RC, and MM.
- Automated Correction Suggestions: The integrated model offers real-time feedback and insights for defect correction, enhancing debugging and code quality.
- Seamless Workflow Integration: Integrating the DC_MPI model into the MPI library ensures a seamless workflow for programmers. They can leverage the model's capabilities directly within their existing programming environment, enabling efficient defect detection and correction without the need for separate tools or external processes.
- Reduced Debugging Efforts: The integrated model assists programmers in identifying and addressing defects early in the development process. This leads to a reduction in debugging

efforts by catching issues before they can cause more complex problems or impact the program's execution.
- Enhanced Code Quality and Reliability: DC_MPI improves MPI code's quality and reliability. It detects and corrects defects, prevents errors, promotes best practices, and ensures proper MPI function usage.
- Time and Resource Savings: Integrating the DC_MPI model into the MPI library streamlines the defect detection and correction process. By automating these tasks, programmers save time and resources that would otherwise be spent manually identifying and resolving defects.

Overall, integrating the DC_MPI model into existing MPI libraries empowers programmers with advanced defect detection, automated correction suggestions, and improved code quality. It simplifies the development workflow, reduces debugging efforts, and enhances the reliability and efficiency of MPI-based applications.

6 Limitations of the DC_MPI Model

The DC_MPI model has shown promising results in reducing the time and effort required for defect correction. However, it is important to acknowledge certain limitations associated with the model. These limitations can provide insights into areas that require attention and further improvement. The following are some limitations of the DC_MPI model.

6.1 Generalization to Unseen Defects

While the DC_MPI model has demonstrated effectiveness in detecting and correcting defects in the 40 MPI code examples we tested, its ability to generalize to defects is not guaranteed. It may encounter challenges when facing new or complex defect patterns that were not encountered during the training phase.

6.2 Limited Scope of Defects

The DC_MPI model primarily focuses on defects related to MPI communication: Illegal MPI calls, DL, RC, and MM. The model may not effectively detect or correct other types of defects that are not directly related to MPI communication, such as algorithmic errors or memory leaks.

While the DC_MPI model has demonstrated effectiveness in detecting defects and reducing the time and effort required for correction, it is important to consider these limitations. Further research and development efforts should focus on addressing these limitations to enhance the model's accuracy and generalization capabilities.

7 Conclusions and Future Work

In this research, we proposed the DC_MPI model, which is a defect detection and correction model specifically designed for scenarios involving MPI-based communication. The model underwent evaluation to assess its performance in detecting and correcting defects in MPI code examples. The evaluation results indicate that the DC_MPI model achieved a detection rate of 37 out of 40 MPI codes, resulting in an overall detection accuracy of 92.5%. This demonstrates the model's effectiveness in identifying defects present in MPI-based communication patterns. By accurately pinpointing these errors, the DC_MPI model provides valuable support to programmers in identifying and rectifying issues that could potentially impact the performance and reliability of their MPI applications. Furthermore, the program execution duration of the DC_MPI model ranged between

0.81 and 1.36 s. This indicates that the model operates efficiently and delivers relatively fast results, enabling programmers to promptly identify and address defects within their MPI codebases. The model's ability to provide timely feedback contributes to an expedited defect correction process, saving valuable time and effort for software development teams.

While the DC_MPI model has demonstrated strong performance in defect detection and correction, there are areas that warrant further improvement. In the future, we have planned to enhance the DC_MPI model in several ways. First, we aim to expand the variety and quantity of training data used to train the model. By incorporating a wider range of defect scenarios, including rare and complex cases, we can improve the model's ability to handle unforeseen defects more accurately. Additionally, we intend to incorporate techniques that can identify algorithm errors, memory leaks, and other types of defects that extend beyond the scope of MPI communication. This expansion will make the model more comprehensive and capable of addressing a broader range of software issues. Furthermore, as part of our future work, we plan to implement a method specifically designed to detect and correct DL, RC, and MM defects in OpenMP. By extending the capabilities of the model to cover OpenMP-related defects, we can provide programmers with a more holistic solution for identifying and rectifying synchronization issues and other problems arising from parallel programming.

These future developments will contribute to the overall effectiveness of the DC_MPI model, enabling it to handle a wider range of defects in both MPI and OpenMP contexts. By incorporating additional training data, expanding defect detection capabilities, and addressing OpenMP-related issues, we aim to improve code quality and reduce the effort required for defect identification and correction.

Acknowledgement: The Deanship of Scientific Research (DSR) at King Abdulaziz University (KAU), Jeddah, Saudi Arabia has funded this project.

Funding Statement: This research is funded by the Deanship of Scientific Research at King Abdulaziz University, Jeddah, Saudi Arabia under the Grant No. RG-12-611-43.

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: F. Eassa, S. Sharaf; data collection: N. Al-Johany; analysis and interpretation of results: N. Al-Johany, F. Eassa, S. Sharaf, R. Alnanih; draft manuscript preparation: N. Al-Johany; paper reviewing and editing R. Alnanih; paper proof reading: F. Eassa, S. Sharaf, R. Alnanih. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data and materials used in this study are available to interested researchers upon request.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] L. Clarke, I. Glendinning, and R. Hempel, "The MPI message passing interface standard," presented at the Programming Environments for Massively Parallel Distributed Systems: Working Conf. of the IFIP WG 10.3, Monte Verita, Switzerland, Apr. 25–29, 1994, pp. 213–218.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "Communicators", in *MPI-The Complete Reference: The MPI Core*, 2nd ed. London, England: MIT Press, 1998, vol. 1, pp. 201–252.

[3] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *Proc. Eighth Annual Conf. on Objectoriented Programming Systems, Languages, and Applications*, New York, NY, USA, 1993, pp. 91–108.

[4] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996. doi: 10.1109/2.546611.

[5] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proc. Sixth Workshop on I/O in Parallel and Distributed Systems*, New York, NY, USA, 1999, pp. 23–32.

[6] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *Proc. Eighth ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, New York, NY, USA, 2001, pp. 123–132.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs," in *Proc. of the Ninth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, New York, NY, USA, 2003, pp. 84–94.

[8] A. Droste, M. Kuhn, and T. Ludwig, "MPI-checker: Static analysis for MPI," in *Proc. Second Workshop on the LLVM Compiler Infrastructure in HPC*, New York, NY, USA, 2015, pp. 1–10.

[9] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with umpire," in *SC'00: Proc. 2000 ACM/IEEE Conf. on Supercomputing*, Dallas, TX, USA, IEEE, 2000, pp. 51.

[10] V. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "PARCOACH extension for static MPI nonblocking and persistent communication validation," in *2020 IEEE/ACM 4th Int. Workshop on Software Correctness for HPC Applications (Correctness)*, GA, USA, IEEE, 2020, pp. 31–39. doi: 10.1109/Correctness51934.2020.00009.

[11] R. A. Alnemari, M. A. Fadel, and F. Eassa, "Integrating static and dynamic analysis techniques for detecting dynamic errors in MPI programs," *Int. J. Comput. Sci. Mob. Comput.*, vol. 7, no. 4, pp. 141–147, 2018. doi: 10.47760/ijcsmc.

[12] E. Saillard, P. Carribault, and D. Barthou, "Combining static and dynamic validation of MPI collective communications," in *Proc. 20th European MPI Users' Group Meeting*, New York, NY, USA, 2013, pp. 117–122.

[13] E. Saillard, M. Sergent, C. T. A. Kaci, and D. Barthou, "Static local concurrency errors detection in MPI-RMA programs," in *2022 IEEE/ACM Sixth Int. Workshop on Software Correctness for HPC Applications (Correctness)*, Dallas, TX, USA, IEEE, 2022, pp. 18–26.

[14] J. Protze, T. Hilbrich, B. R. de Supinski, M. Schulz, M. S. Müller and W. E. Nagel, "MPI runtime error detection with must: Advanced error reports," in *Tools for High Performance Computing 2012*, Berlin, Heidelberg: Springer, 2013, pp. 25–38.

[15] H. M. Wei, J. Gao, P. Qing, K. Yu, Y. F. Fang and M. L. Li, "MPI-RCDD: A framework for MPI Runtime Communication Deadlock Detection," *J. Comput. Sci. Technol.*, vol. 35, pp. 395–411, 2020. doi: 10.1007/s11390-020-9701-4.

[16] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A tool for model checking MPI programs," in *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, New York, NY, USA, 2008, pp. 285–286.

[17] A. S. Alghamdi, A. M. Alghamdi, F. E. Eassa, and M. A. Khemakhem, "ACC_TEST: Hybrid testing techniques for MPI-based programs," *IEEE Access*, vol. 8, pp. 91488–91500, 2020. doi: 10.1109/ACCESS.2020.2994172.

[18] S. M. Altalhi *et al.*, "An architecture for a tri-programming model-based parallel hybrid testing tool," *Appl. Sci.*, vol. 13, no. 21, pp. 11960, 2023. doi: 10.3390/app132111960.

[19] S. Li, M. Wang, and H. Zhang, "Deadlock detection for MPI programs based on refined match-sets," in *2022 IEEE Int. Conf. on Cluster Computing (CLUSTER)*, Heidelberg, Germany, IEEE, 2022, pp. 82–93.

[20] N. A. Al-johany, F. E. Eassa, S. A. Sharaf, A. Y. Noaman, and A. Ahmed, "Prediction and correction of software defects in message-passing interfaces using a static analysis tool and machine learning," *IEEE Access*, vol. 11, pp. 60668–60680, 2023. doi: 10.1109/ACCESS.2023.3285598.

[21] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "Deadlock detection in MPI programs," *Concurr. Comput. Pract. Exp.*, vol. 14, no. 11, pp. 911–932, 2002. doi: 10.1002/cpe.701.