LLNL-CONF-586816

MPI Runtime Error Detection with MUST: Advanced Error Reports

J. Protze, T. Hilbrich, B. de Supinski, M. Schulz, M. Mueller, W. Nagel

October 1, 2012

6th Parallel Tools Workshop
Stuttgart, Germany
September 25, 2012 through September 26, 2012

Chapter 1
MPI Runtime Error Detection with MUST: Advanced Error Reports

Joachim Protze, Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, Matthias S. Müller, and Wolfgang E. Nagel

Abstract The Message Passing Interface (MPI) is a widely used paradigm for distributed memory programming. Its API is primarily designed for good performance and less for usability; it provides only very limited abstractions that help enforce its correct use. As a result, application developers need tools that aid in the detection and removal of MPI usage errors. Our runtime error detection tool MUST addresses this issue and provides a wide range of automatic correctness checks. MUST uses state-of-the-art approaches to cope with complex MPI semantics like derived datatypes, collective operations, and wildcard receive operations. However, equally important to detecting correctness violations, is that such correctness tools present all details of the violating MPI call(s) required to pinpoint the problem in the source code and to remove the error. In this paper we focus on the error reports presented by MUST and propose a new set of error reports that present complex errors with fine-grained details of the error situation. This includes a deadlock view and a view for usage errors in complex MPI datatypes.

1.1 Introduction

The development of Message Passing Interface (MPI) [1] applications is a time consuming and complex task. One of the key challenges, aside from achieving high efficiency, is guaranteeing soundness of an application's use of MPI, i.e., its correct usage of the MPI API. While some MPI related errors may directly cause wrong re-

Joachim Protze · Tobias Hilbrich · Matthias S. Müller · Wolfgang E. Nagel
Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, D-01062 Dresden, Germany
e-mail: {joachim.protze, tobias.hilbrich, matthias.mueller, wolfgang.nagel}@tu-dresden.de

Bronis R. de Supinski · Martin Schulz
Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
e-mail: {bronis, schulzm}@llnl.gov

sults, application crashes, or hangs, some errors may only manifest on some systems or runs and then in some cases only long after their cause or simply by producing wrong results at the end of the execution. If done manually, finding such problems can be a long and difficult task and developers therefore require tool support that aids in the removal of these errors. Runtime error detection, i.e., detecting errors during an application run, is one tool class that provides this support. We develop the Marmot Umpire Scalable Tool (MUST), named after its predecessor tools Marmot [2] and Umpire [3], for this purpose.

Process 0	Process 1
MPI_Recv(from:1, tag:100)	MPI_Recv(from:0, tag:200)
MPI_Isend(to:1, tag:200, &req)	MPI_Isend(to:0, tag:100, &req)
MPI_Wait(&req)	MPI_Wait(&req)

(a) Recv-recv deadlock.

Process 0	Process 1
MPI_Isend(to:1, tag:200, &req)	MPI_Isend(to:0, tag:100, &req)
MPI_Recv(from:1, tag:200)	MPI_Recv(from:0, tag:100)
MPI_Wait(&req)	MPI_Wait(&req)

(b) Deadlock resulting from a tag mismatch.

Fig. 1.1 MPI usage error examples.

Recent advances in runtime deadlock detection [4] and datatype correctness checks [5] allow MUST to efficiently detect complex errors. However, detecting such errors is only half the solution to the overall problem. Any tool must also present all details about a detected error in a way that helps users understand the erroneous behavior of their codes and help them fix the problem. Consider the following examples that illustrate some potential complexities:

Figure 1.1 presents two deadlock scenarios with simplified MPI calls. Two processes attempt to send and receive a message from each other using blocking receive and non-blocking send calls. The example in Figure 1(a) results in a deadlock, as both processes issue the `MPI_Recv` call without issuing any send calls first. As a result, both processes wait in a cyclic fashion for each other's send call, which is never reached, and hence can't continue execution. MUST's graph-based deadlock detection catches this error and presents the user with a wait-for graph. As no process issued a send call before the receive call, this report includes the key items to understand the error, which in this case are the processes involved in the deadlock and their individual active MPI calls. The situation in Figure 1(b) represents a similar communication, which also results in a deadlock, due to a mismatch in the given message tags. MUST's wait-for graph shows the user that both processes are blocked in the `MPI_Recv` call. As both processes have active send calls, the simple criteria used in the example above doesn't hold and the tool user needs to investigate these calls manually in order to determine whether a tag or even a communicator mismatch exists. Different source files that contain active send calls or the use of variables as tag arguments can complicate this further.

In this paper, we present a set of novel output extensions of MUST that provide tool users with the necessary fine-grained and detailed information of such complex error situations, but without overwhelming them with additional unrelated data. In particular, we include:

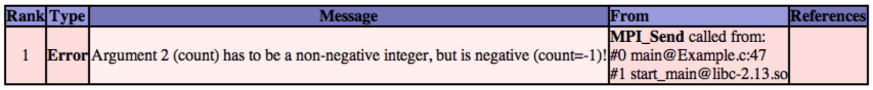
- A parallel call stack that highlights the processes that MUST determined as the root of a deadlock,
- A condensed message queue that only lists send and receive calls that are meaningful in a deadlock situation, and
- A call-stack based decomposition of the message queue graph to augment a regular message queue graph with source location information, and
- A datatype tree view that highlights error positions in derived datatypes.

We first present an overview of MUST, its correctness checks, and its basic error report in Section 1.2, followed by a summary of MUST's current deadlock view and datatype usage reports. Afterwards, we present our proposed deadlock view extensions in Section 1.4. Section 1.5 presents how we can efficiently pinpoint particular error positions in derived datatypes. Finally, we present related work in Section 1.6 and conclude in Section 1.7.

1.2 MUST

MUST detects MPI usage errors, i.e., usage of MPI calls that are not consistent with restrictions laid out in the MPI standard, during an application run and reports them to the user. Examples for such usage errors are illegal parameters to MPI calls, writes to a send buffer while an asynchronous message transfer is in progress, inconsistent orderings of collective operations, or deadlocks due to improper synchronization. MUST uses the MPI profiling interface to intercept and analyze all MPI calls that an application issues. The tool can be loaded into the application using the *LD_PRELOAD* mechanism. In this case, the usage of the tool becomes as easy as replacing the respective *mpiexec* command with a wrapper command called *mustrun*.

We distinguish two types of correctness checks: local correctness checks and non-local checks. Local checks only require information that is available on a single MPI process and hence don't require any communication for their execution. As a result, MUST is able to execute local checks inside each application process, or more precisely inside the MUST MPI wrappers used to intercept all MPI calls. Using local checks, we can, e.g., detect whether a datatype that is used in a communication call is committed or whether parameters to MPI calls are out of range. Non-local correctness checks require information from more than one process. Datatype signature matching between sending and receiving communication calls is one such example. The implementation of non-local correctness checks requires additional communication and hence a separate communication mechanism that can forward information about MPI calls to other processes or extra resources. MUST uses the



Rank	Type	Message	From	References
1	**Error**	Argument 2 (count) has to be a non-negative integer, but is negative (count=-1)!	**MPI_Send** called from: #0 main@Example.c:47 #1 start_main@libc-2.13.so	

Fig. 1.2 Example MUST error report.

Generic Tool Infrastructure (GTI) [6] for this purpose. Currently MUST provides the following classes of correctness checks covering a wide spectrum of possible error cases:

- Local:
  - Integer checks (e.g., restrictions on tags, counts, sizes, and offsets)
  - Integrity checks (e.g., Arrays allocated or communication buffer present)
  - MPI resource surveillance (e.g., use of requests, datatypes, reduce operations, groups, and communicators)
  - Resource leak checks
  - Communication buffer overlap checks
- Non-local:
  - Collective verification (e.g., matching roots and compatible reduce operations)
  - Lost message detection
  - Message type matching (for both point-to-point and collective operations)
  - Deadlock detection

Previous work [4] includes extensive performance results and has shown the feasibility of this approach, including its scalability using an application study on up to 512 processes.

In its initial form, the basic output of MUST is an HTML table that follows the format of Marmot [7]. In Marmot checks had to be implemented for each MPI call, even for the same error conditions, leading to significant code duplication of any error reporting. MUST avoids this redundancy with the use of so-called argument IDs. Figure 1.2 shows a basic MUST report with an integer usage error. The check that detects the negative count argument in the `MPI_Send` call is mapped to many different calls and argument types. MUST uses the argument IDs to identify the argument number and name, which increases the detail in its output reports. Further, MUST uses the Stackwalker API of the Dyninst project[1] to retrieve call stack information for each MPI call it intercepts.

[1] http://www.dyninst.org/

1.3 Shortcoming of Current Error Views

While the initial MUST implementation provided useful information about violated checks, the output format was not optimal and omitted several key pieces of information a user requires to identify the broken code location and to fix it. These shortcomings were introduced because the initial output format was driven by the implementation of the tool and what it naturally collects, without taking the user's needs into account. This is, unfortunately, common for many tools, which flood the user with raw data, but fail to provide some essential details. We illustrate two such problems in the following, using the examples of deadlock detection and problems with complex datatypes. We will first show (in this section) why the existing views are insufficient and (following in the next two sections) how we were able to work around it.

1.3.1 Example: Pinpointing deadlocks

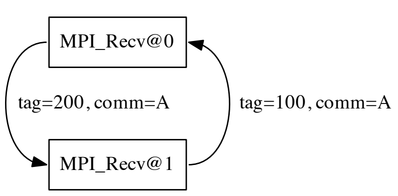
A key feature of MUST is its graph-based deadlock detection [8]. It creates a wait-for graph and then uses this graph to identify existing deadlock conditions. If such a condition is found, the tool provides the user with a list of processes that are in a deadlocked state as well as their wait-for dependencies that cause them to be deadlocked. This enables MUST to separate processes that cause the deadlock from processes that hang due to waiting for deadlocked processes directly or indirectly.

The graph based approach also has the additional advantage that we can use the graph itself to visualize the deadlock conditions and the wait-for dependencies to the user. As a result, MUST's previous deadlock view provides:

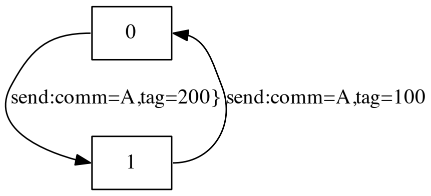
- A textual description of the deadlock situation,
- A wait-for graph of the deadlocked processes, and
- A source location list of the deadlock processes.

In the following we use the erroneous sequence of MPI calls in Figure 1(b) as an example to illustrate MUST's previous output. Figure 3(a) shows the wait-for graph (WFG) that MUST provides for this example. However, this graph along with the source location lists of the deadlocked processes alone is not sufficient to identify the root cause for this error. From our experience, a tool must provide answers to the following questions:

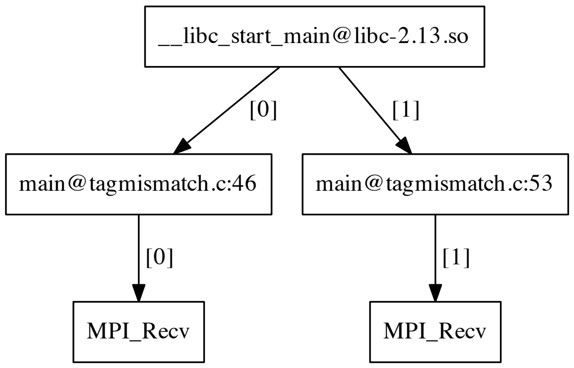
1. Which processes cause the deadlock?
2. What MPI calls are active on these processes?
3. Which control flow led to these active calls?
4. In the case of involved point-to-point operations, which other active communications exist?



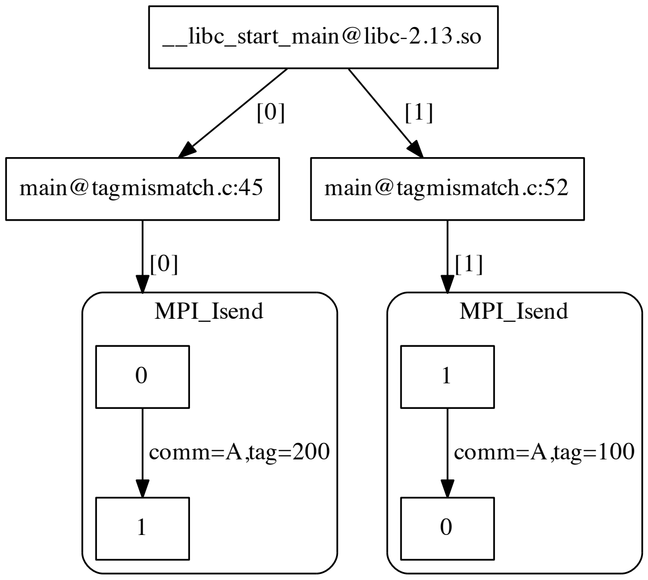
(a) Wait-for graph.



(b) Message queue graph.



(c) Parallel call stack graph.



(d) Call stack graph decomposition of the message queue graph.

Fig. 1.3 Deadlock view components for the example in Figure 1(b).

While MUST's previous output provides answers to the first two questions it does not provide information on the latter two. Also, the list of source locations is insufficient for deadlock reports that involve more than a few processes.

1.3.2 Example 2: Viewing datatype related problems

The MPI standard imposes constraints for communication operations. Erroneous usage of MPI datatypes may collide with three of such constraints. In the following we sketch these three referring to version 2.2 of the MPI standard [1]:

- For sending operations, the application may not modify the communication buffer, until the send completes.
- For receiving operations, the application must not access any part of the communication buffer, until the receive completes.
- The type signature of a communication must adhere to matching rules during the following three steps:

    1. MPI types must match programming language types for reads from the application memory (except for the MPI type `MPI_BYTE`),
    2. MPI types must match on receiver and sender sides during transport to receiver, and
    3. MPI types must match programming language types for writes to the application memory (except for the MPI type `MPI_BYTE`).

In MUST we provide checks for overlapping communication buffers handling a sub-set of clashes with the first two constraints, and for type matching in communication which meets step two of the latter constraint. These checks handle any (derived) datatypes that communication calls may use. We provide no checks for memory manipulation done in application context. Instead, we focus on simultaneous MPI communications that break any of these constraints. If MUST detects such an error, it is crucial that it provides precise information on its source. While the simplest solution would be to provide memory addresses, this provides unsatisfactory details on where the error resides in a communication buffer and its associated MPI datatype. We currently use a path expression approach [5] to pin-point these error locations. An example for this path expression can be found in Section 1.5. While these expressions provide an exact position of the error location within a datatype signature, they require a deep understanding of their format, while losing information about the overall structure of the involved datatype(s).

1.4 Deadlock View in MUST

As the last section illustrated, MUST's previous deadlock view lacked detail, especially for message mismatch situations, and scalability. To overcome this limitation, we propose a new, dedicated deadlock view that contains the following elements:

- A textual summary,
- A communicator overview,
- The WFG with a legend,
- A parallel call stack,
- A graph representation of the current message queue, and
- A decomposition of the message queue that uses a parallel call stack.

Our new output generator in MUST combines all of these elements in a single HTML page (for better readability, however, we present the individual elements in separate sub-figures). While the textual summary matches our previous outputs, we use the communicator overview to represent each communicator with an upper case letter. In the erroneous sequence of MPI calls in Figure 1(b), which we use as an example throughout this section, the application only uses `MPI_COMM_WORLD`, which we represent as *comm A*. If additional communicators are defined by the application, the communicator summary includes information on the MPI calls that created the communicator. The WFG (Figure 3(a)) matches our previous outputs, except that we now use the communicator symbols to also present information on the communicators in use. We also add a legend to this graph as it may contain intermediate nodes to represent complex MPI semantics. Additionally, the new view shows the parallel call stack to provide insights for Question 3 (introduced in Section 1.3) and the last two graphs to provide information for Question 4, which we describe in the following.

Figure 3(c) shows MUST's parallel call stack for our example. It helps to illustrate control flow decisions that lead to the deadlock condition. While it is challenging to represent information on the control flow of the individual processes in all details, this limited view provided by call stacks is in most cases sufficient. Additional static source analysis may reveal control flow relevant variables to enrich parallel call stack graphs with further information, as an extension [9] of the STAT [10] tool shows. Further, these graphs scale well with the number of application processes. For our purposes, we limit this call stack graph to only the application processes that are part of the deadlock in order to remove any unnecessary information and provided the most concise representation.

Question 4 addresses situations where point-to-point operations are involved in a deadlock. In this case the root-cause of the error may be a tag or communicator mismatch. In order to understand this situation, the application developer requires information about any active and meaningful point-to-point call, whether it is involved in the actual deadlock condition or not. MUST provides a message queue graph for this purpose. Since MUST detects which processes are part of the deadlock, while it also determines which processes are blocked in point-to-point calls, we can automatically reduce the full message queue graph to only present messages that:

- were started by a process that is part of the deadlock;
- have active send operations, which target a process that hangs in a receive operation or a completion that includes a non-blocking receive operation; or
- have active receive operations, which target a process that hangs in a send operation or a completion that includes a non-blocking send operation.

Using these conditions, we can condense our output to only present relevant point-to-point operations. Figure 3(b) shows this graph for our example. The graph includes an arc from node 0 (which represents process 0) to node 1 to represent the `MPI_Isend` call that was issued on process 0 before the deadlock manifested. The other arc represents the `MPI_Isend` operation that was started by process 1.

MUST's condensed message queue graph allows application developers to determine whether a potential mismatch exists. In our example, Figure 3(a) shows that process 0 waits for a matching send operation of process 1, which uses the tag 200, while Figure 3(b) shows that a send operation exits, but with tag 100. If a mismatch exists, the user needs to be able to identify the call and control flow origin of the mismatched operation. We use a parallel call stack to represent all MPI operations that started any operation within MUST's relevant message queue graph. This identifies the call stacks of these operations, but as each operation may use multiple targets, tags, and communicators, we need to highlight which individual parts of the message queue graph result from each leaf of the call stack graph. As a result, we decompose the message queue graph into sub-graphs that represent the components that each MPI operation creates. Figure 3(d) shows this call-graph-based decomposition for our example. This graph allows the tool user to determine which message might be mismatched, while it contains information about its source location along with limited control flow information.

1.5 Type Tree View

In this Section we will describe a new, more expressive graphical view for datatype related errors.

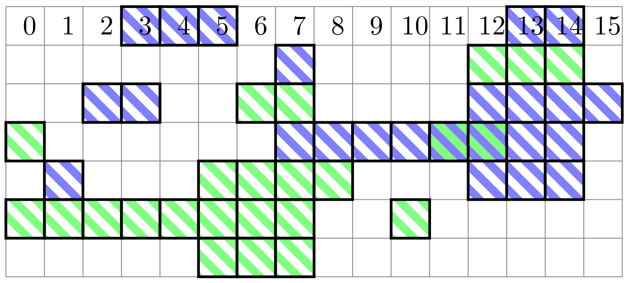
The code example in Listing 1.1 sketches a particle simulation where information about a subset of the particles needs to be transferred to a neighbor process. In the application a C struct holds the information about a particle. The set of particles is organized in an array of this struct. Using derived datatypes, MPI enables us to select the subset from the array and send it in a single contiguous operation to the neighbor. To create the fitting datatype, the example uses at first the `MPI_Type_struct` constructor to represent the C struct and then an `MPI_Type_indexed` constructor to select parts of an array of this struct. While the first constructor is correct with respect to type matching, the second one causes a communication buffer overlap when the example issues the `MPI_Sendrecv` call (performed as local

Listing 1.1 Example for a communication buffer overlap

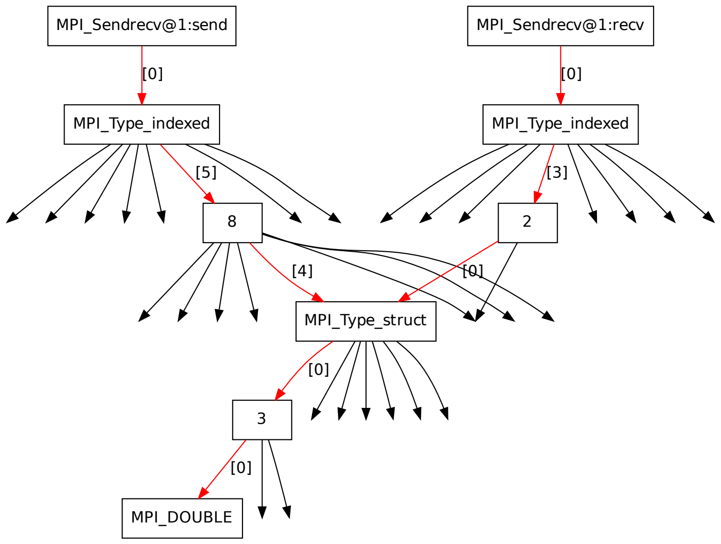
```
  double velocity[3]; double spin[3]; char charge;
  double radius; double mass; };

struct particle cloud[112];
MPI_Datatype structtype, indexedtype;

int blocklens[7] = {3, 3, 3, 3, 1, 1, 1};
MPI_Datatype types[7] = {MPI_DOUBLE, MPI_INT, MPI_DOUBLE,
    MPI_DOUBLE, MPI_CHAR, MPI_DOUBLE, MPI_DOUBLE};
// displs derived from c-struct by MPI_Get_address()
MPI_Aint displs[7] = {0, 24, 40, 64, 88, 96, 104, 112};
MPI_Type_struct (7, blocklens, displs, types,
    &structtype);

int array_of_blocklens[8] = {3, 2, 1, 2, 4, 8, 1, 3};
int array_of_displs[8] = {3, 13, 23, 34, 44, 55, 65, 76};
MPI_Type_indexed (8, array_of_blocklens, array_of_displs,
    structtype, &indexedtype);
MPI_Type_commit(&indexedtype);

MPI_Sendrecv(cloud, 1, indexedtype, 0, 42, cloud + 25, 1,
    indexedtype, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

operation in this simplified example). MUST's current path expressions calculate to `[0](INDEXED)[5][4](STRUCT)[0][0](DOUBLE)` for the sending part and `[0](INDEXED)[3][0](STRUCT)[0][0](DOUBLE)` for the receiving part of the `MPI_Sendrecv` call. Figure 4(a) sketches the overlap within the array (called *cloud*) of the C structure, i.e., the elements that the `MPI_Type_indexed` constructor selects from the array. While this representation highlights the overlap, this display loses information about the internal datatype structure. To combine the expressiveness of the path expression and the overview of such a memory map, we propose an overlap graph. This graph visualizes the two path expressions that cause the overlap along with a sketched structure of the datatypes in use. Figure 4(b) shows this graph for the example in Listing 1.1. We represent the path expressions of the overlap in red in this graph. For overlaps the trees of the colliding communication operations will either join at a node of the same basic MPI type and absolute offset, as in our example, or we use a compound node if the overlap occurs for two different types/offsets. We join further tree nodes if they compare to equal sub-types, as for the `MPI_Type_struct` in our example. We compute this by recursing the type trees from the leaf towards its root.

An example for a type mismatch can be derived from the above example by mixing up the struct entries for `charge` and `radius` at one of the neighbor processes. The current path expression for this situation calculates to `[0](INDEXED)[0][0](STRUCT)[4][0](CHAR)` and `[0](INDEXED)[0][0](STRUCT)[4][0]`



(a) Array indices of send / receive marked blue / green.



(b) Overlap graph.

Fig. 1.4 Overlap view for the example in Listing 1.1.

`(DOUBLE)`, indicating that an `MPI_CHAR` mismatches with an `MPI_DOUBLE`. To display the mismatch we create a tree for both involved datatypes where we skip nodes apart from the (red) error path while we keep a few basic MPI types near the mismatch position to have a more detailed context of the mismatch. To derive a smaller graph we merge similar nodes of both trees. Figure 1.5 provides the resulting view for the sketched mismatch situation.

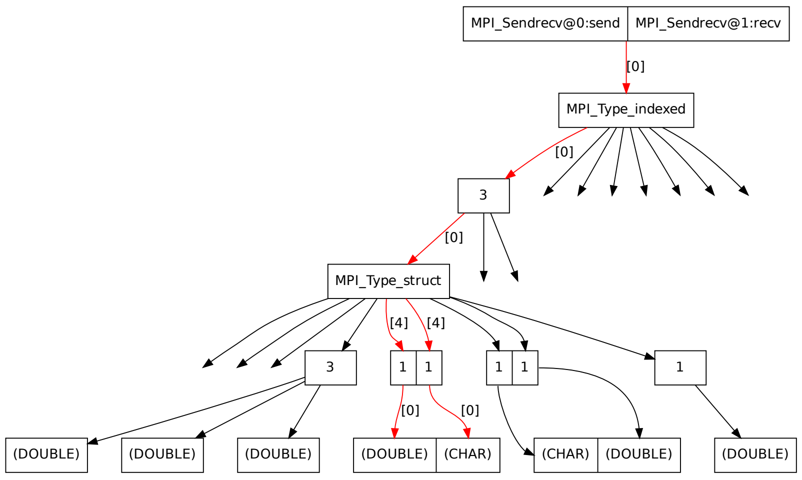


Fig. 1.5 Type mismatch view for a confusion in the type definition.

1.6 Related Work

This work directly relates to other runtime error detection approaches for MPI applications, which include Marmot [2], Umpire [3], ISP [11], MPI-Check [12], and Intel's approach [13]. While MUST is the successor of both Marmot and Umpire, the MPI-Check tool and Intel's approach use a timeout-based deadlock detection. As a result, these tools only provide a list of all active MPI calls when the presence of a deadlock is suspected. ISP runs a replay based investigation of all possible interleavings of an MPI application. As a result, this tool can detect some deadlocks that MUST would not detect in a certain application run. ISP's deadlock output includes a trace of all MPI calls that each process issued, as well as their matching decisions. While very detailed, this output will get overly complex, especially for longer application runs with more than a few processes. While our output contains no complete history of all issued MPI calls, we provide the user with a more scalable deadlock view that condenses relevant history information with the use of a reduced message queue graph.

The STAT [10] tool and debuggers like DDT and Totalview use parallel call stack graphs and/or message queue graphs. Debuggers use interfaces to the MPI library [14] to retrieve message queue information, whereas MUST tracks all MPI calls during the whole application run. Existing integrations of runtime error detection tools with debuggers, e.g. DDT and Marmot [15], could be extended to provide debuggers with information on which processes cause a deadlock. Debuggers could than condense message queue graphs as in our approach. Also, the representation of derived datatypes with trees is based on ideas of the flattening on the fly technique [16].

1.7 Conclusions

We present the MUST runtime error detection tool for MPI applications along with extensions of its error reports. Our previous output for deadlock situations failed to capture information on active point-to-point messages, which is crucial in the detection of message mismatch situations. We use message queue graphs to present these active operations. MUST's graph-based deadlock detection yields a set of processes that cause the deadlock, which allows us to condense parallel call stacks and message queues to only include relevant information. In order to add call location information to the message queue graph representation, we propose an extended parallel call stack graph that includes a decomposition of the message queue graphs in their leaves. While these representations allow us to present relevant information for the removal of deadlocks at moderate scale, we still need to investigate their practicability for thousands or more processes. While our approach allows us to visualize deadlocks that only involve a few processes, it may fail for complex deadlocks that involve all or most application processes. This especially affects the size of the WFG and the message queue graphs.

Our second error view provides a detailed output for errors that involve derived datatypes. This includes communication buffer overlaps, and type mismatches between point-to-point or collective MPI operations. The removal of these errors requires a precise understanding of which part in a derived datatype causes the error. As a result, we use a narrowed type tree representation that highlights the position in the datatype that causes the error, while it sketches the structure of the involved datatypes at the same time.

Acknowledgements Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-??????). This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. http://www.mpi-forum.org/docs/mpi22-report.pdf (April 2009)
2. Krammer, B., Müller, M.S.: MPI Application Development with MARMOT. In: PARCO. Volume 33., Central Institute for Applied Mathematics, Jülich, Germany (2005) 893–900
3. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. Supercomputing, ACM/IEEE 2000 Conference (04-10 Nov. 2000) 51–51

4. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: Mpi runtime error detection with must: Advances in deadlock detection. In: Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '12, New York, NY, USA, ACM (2012)
5. Protze, J., Hilbrich, T., Knüpfer, A., de Supinski, B.R., Müller, M.S.: Holistic Debugging of MPI Derived Datatypes. In: IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium. (2012)
6. Hilbrich, T., Müller, M.S., de Supinski, B.R., Schulz, M., Nagel, W.E.: GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems. In: IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium. (2012)
7. Krammer, B., Hilbrich, T., Himmler, V., Czink, B., Dichev, K., Müller, M.S.: MPI Correctness Checking with Marmot. In: Parallel Tools Workshop'08. (2008) 61–78
8. Hilbrich, T., de Supinski, B.R., Schulz, M., Müller, M.S.: A Graph Based Approach for MPI Deadlock Detection. In: ICS '09: Proceedings of the 23rd International Conference on Supercomputing, New York, NY, USA, ACM (2009) 296–305
9. Ahn, D.H., de Supinski, B.R., Laguna, I., Lee, G.L., Liblit, B., Miller, B.P., Schulz, M.: Scalable temporal order analysis for large scale debugging. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. SC '09, New York, NY, USA, ACM (2009) 44:1–44:11
10. Arnold, D., Ahn, D., de Supinski, B., Lee, G., Miller, B., Schulz, M.: Stack Trace Analysis for Large Scale Debugging. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. (march 2007) 1 –10
11. Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: ISP: A Tool for Model Checking MPI Programs. In: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2008) 285–286
12. Luecke, G.R., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., Zou, Y.: MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs. Concurrency and Computation: Practice and Experience **15**(2) (2003) 93–100
13. Desouza, J., Kuhn, B., Supinski, B.R.D.: Automated, Scalable Debugging of MPI Programs with Intel Message Checker. In: In Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS. (2005)
14. Cownie, J.: A standard interface for debugger access to message queue information in MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 1697 of Lecture Notes in Computer Science, Springer Verlag (1999) 51–58
15. Krammer, B., Himmler, V., Lecomber, D.: Coupling DDT and Marmot for debugging of MPI applications. In: PARCO'07. (2007) 653–660
16. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. In: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, Springer-Verlag (1999) 109–116