Detection of violations to the MPI Standard in hybrid OpenMP/MPI Applications

Tobias Hilbrich[1], Matthias S. Müller[1], and Bettina Krammer[2]

[1] TU Dresden, Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany
[2] High Performance Computing Center (HLRS), Universität Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany

Abstract. The MPI standard allows the usage of multiple threads per process. The main idea was that an MPI call executed at one thread should not block other threads. In the MPI-2 standard this was refined by introducing the so called *level of thread support* which describes how threads may interact with MPI. The multi-threaded usage is restricted by several rules stated in the MPI standard. In this paper we describe the work on an MPI checker called MARMOT[1] to enhance its capabilities towards a verification that ensures that these rules are not violated. A first implementation is capable of detecting violations if they actually occur in a run made with MARMOT. As most of these violations occur due to missing thread synchronization it is likely that they don't appear in every run of the application. To detect whether there is a run that violates one of the MPI restrictions it is necessary to analyze the OpenMP usage. Thus we introduced artificial data races that only occur if the application violates one of the MPI rules. By this design all tools capable of detecting data races can also detect violations to some of the MPI rules. To confirm this idea we used the Intel® Thread Checker.

1 Introduction

1.1 Hybrid MPI Applications

The MPI standard states that MPI calls should only block the calling thread. This was refined in the MPI-2 standard by introducing the so called *level of thread support* (*thread level*). Each MPI implementation supports the lowest level and may support a higher one. The levels are:

MPI_THREAD_SINGLE:	only one thread exists
MPI_THREAD_FUNNELED:	multiple threads may exist but only the main[3] thread performs MPI calls
MPI_THREAD_SERIALIZED:	multiple threads exist and each thread may perform MPI calls as long as no other thread is calling MPI

[3] The thread that initialized MPI.

MPI_THREAD_MULTIPLE: multiple threads may call MPI simultaneously

The application specifies a desired *thread level* and passes it to the MPI implementation, which might return a lower level. The level returned is referred to as provided *thread level* and must not be violated.

1.2 Restrictions for Hybrid MPI Applications

Besides the definition of the *thread level* there are further restrictions mentioned in the MPI standard. Here we present the restrictions that are currently checked by our MPI checker. This list is incomplete, e.g. most MPI-2 calls are currently unsupported. The currently checked restrictions are:

(I) *The call to MPI_FINALIZE should occur on the same thread that initialized MPI. We call this thread the main thread. The call should occur only after all the process threads have completed their MPI calls, and have no pending communications or I/O operations.* [2, page 194 lines 24-29]

(II) *A program where two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI_WAIT{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test which violates this rule is erroneous.* [2, page 194 lines 32-36]

(III) *A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.* [2, page 194f line 46ff]

(IV) *Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the usere's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.* [3, page 130 lines 37-41]

We will refer to these restrictions by using their respective number. The additional restriction that the provided *thread level* may not be violated will be referred to as (V). To our knowledge none of the available MPI Checkers does check any of these restrictions. The MPI checkers we looked at are: Umpire[4], MPI-Check[5] and Intel® Message Checker[6].

2 Example

For restriction (II) we give an example violation in Table 1. Violations to this restriction are only possible if the *thread level* MPI_THREAD_MULTIPLE is

Process 1	
Thread 1	Thread 2
MPI_Isend(msg,&request)	
#pragma omp barrier	#pragma omp barrier
MPI_Wait(request)	MPI_Test(request)

Table 1. Violation to restriction (II), assuming that *request* is a shared variable.

available. This applies to most of the restrictions. In the example, process 1 has two threads that are simultaneously performing an MPI_Wait and an MPI_Test call. Both calls are using the same request. This violates restriction (II). As one can see if the MPI_Test call is performed before the MPI_Wait call the error won't occur whereas simultaneous execution yields an error.

3 Constraints for the Restrictions

In order to create checks for restrictions (I) to (V) we want to define constraints that implicate whether one of the restrictions is violated. These constraints can be used to implement checks and to verify whether the implemented checks can detect every instance of the problem. If a run of an application matches any of these constraints this implies that there is a violation to the respective restriction. The constraints are:

violation to (I):
If and only if one of the following holds:
(A) for a thread calling MPI_Finalize holds: "MPI_Is_thread_main() == False"
(B) one thread is performing an MPI_Finalize call while another thread is also calling MPI
(C) an MPI call is issued after the call to MPI_Finalize is finished

violation to (II):
If and only if a thread performs a *Wait*[4] or *Test*[5] call using Request X while another thread is also using Request X in a *Wait* or *Test* call.

violation to (III):
If and only if one of the following holds:
(A) one thread is performing a *Probe*[6] call that will return the value X as *source* and the value Y as *tag* and another thread is performing a *Recieve*[7] call with *source* and *tag* values that match X and Y at the same time
(B) a thread is performing a *Receive* call with source and tag values returned by a *Probe* call of another thread that did not receive that message yet

[4] An MPI_Wait{ANY|ALL|SOME} call.
[5] An MPI_Test{ANY|ALL|SOME} call.
[6] An MPI_Iprobe or an MPI_Probe call.
[7] An MPI_Recv, MPI_IRecv, MPI_Sendrecv, MPI_Sendrecv_replace or (MPI_Start) call.

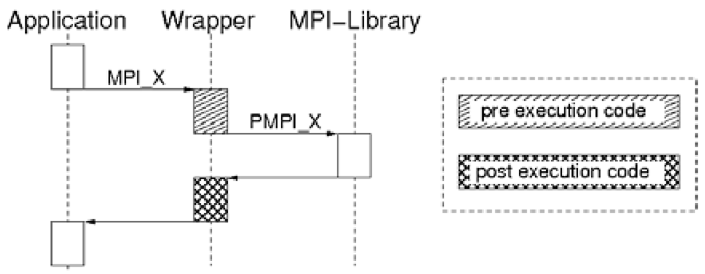

Fig. 1. Execution of an MPI call when using a wrapper.

(C) one thread is receiving a message with *source* and *tag* values returned by a *Probe* it called and another thread is receiving a message with matching *source* and *tag* values at the same time (assumed that the probed message was not yet received)

violation to (IV):

If and only if a thread performs a collective call using communicator X while another thread is within a collective call using communicator X

violation to (V):

If and only if one of the following holds:

(A) provided is MPI_THREAD_SINGLE, omp_in_parallel() returns true and omp_get_num_threads() returns a value greater than 1 while the application is in an MPI call

(B) provided is MPI_THREAD_FUNNELED and MPI_Is_thread_main() returns false while application is in an MPI call

(C) provided is MPI_THREAD_SERIALIZED and two threads are calling MPI simultaneously

One might have compressed constraints (A)-(C) of restriction (III) into one constraint which detects whether a *Receive* call is issued between a *Probe* and a *Receive* call of another thread (all using the same *source* and *tag* values). But we will see later that this decomposition is useful when creating artificial data races.

4 Applying an MPI Checker to a Hybrid Application

4.1 Instrumenting the Application

Runtime MPI checkers have to monitor all performed MPI calls and check their parameters and results. This is usually achieved by intercepting the MPI calls and executing additional code before and after execution of the MPI call. For this purpose the MPI standard specifies the so called Profiling Interface. For each MPI call "MPI_X" there is a second function "PMPI_X" that executes the same call. Thus it is possible to create wrappers that catch each MPI call and execute additional code before and after calling the appropriate PMPI call. This is illustrated in Fig. 1. We will refer to the code executed before the PMPI Call as the *pre execution code* and to the code after the PMPI call as the *post execution code*.

4.2 Propositions on Correctnes

Using *pre* and *post execution code* in an environment in which multiple threads are calling MPI simultaneously might change the order in which MPI calls are issued by the threads. Imagine a *pre execution code* that synchronizes the threads such that only one thread is calling MPI at a time. That would prohibit violations to most of the above mentioned restrictions. It is desirable that the *pre* and *post execution code* is designed such that it does not change the semantics of the application. This especially requires that if there is a run of the application in which the commands are executed in a certain order between the threads then there is a run with the wrapper attached in which the commands of the application are executed in the same order. We propose several rules for the *pre* and *post execution code* that should enforce preservation of semantics:

- the code must not enforce serialization or a certain ordering of the MPI calls made on the threads of each process
- the code must not enforce an ordering of the MPI calls of different processes
- execution may only take a finite time
- parameters passed to the MPI call must not be modified
- no data that is used by the application or the MPI library must be changed
- no input files used by the application or the MPI library must be changed

We assume that as long as the code within the wrappers does not violate any of these restrictions all errors that appear in the application may still appear when the wrapper is used. However, the probabilities of errors to occur during a run with or without wrapper might differ.

4.3 Synchronization

Our MPI checker uses process local data to store whether certain MPI events have occurred and to track the usage of MPI resources. In a multi threaded environment it is necessary to protect these data against unsynchronized access. To do so we introduced a mutual exclusion mechanism that is executed in the *pre* and *post execution code*. The scheme used is shown in Fig. 2. Before any process global data is used in the *pre execution code* the synchronization is started with a call to “enterMARMOT”. It is stopped at the end of the *pre execution code* by calling “enterPMPI”. The same scheme is used for the *post execution code*. To implement the mutual exclusion we use OpenMP locks. This synchronization should not violate any of the above propositions thus all MPI errors of the Application can still appear when our MPI wrappers are used.

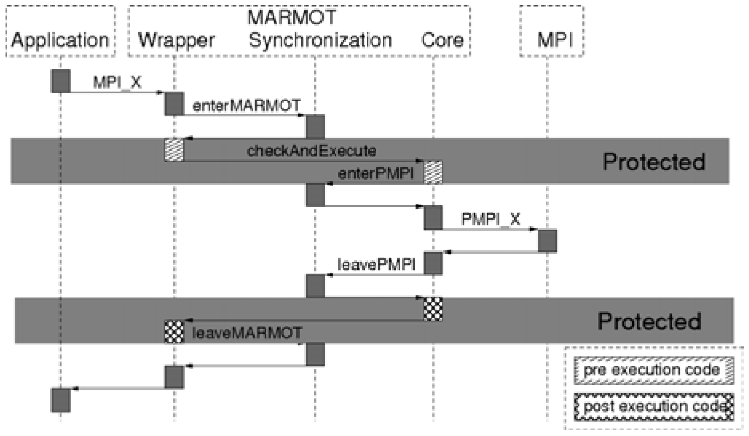

Fig. 2. Scheme of the applied synchronization within MARMOT.

5 First Implementation of Checks

As a first approach we implemented checks that detect violations if they actually occur in a run with MARMOT. Therefore we added *pre* and *post execution code*. Note that some of the code might be executed before the mutual exclusion starts and might thus require additional synchronization. Implementation of most checks is straightforward and requires only a few lines of code. To implement checks for the constraints of restrictions (II) and (IV) we register used requests and communicators. We illustrate the used technique for the case of the communicators. In the *pre execution code* of all collective calls we check whether the given communicator is already registered as used. If that's the case we return an error, otherwise we register the communicator as used. In the *post execution code* we remove the communicator from the registration. In order to create a check for the constraints of restriction (III) we use a mechanism that registers *source* and *tag* values returned by a probe. These values are registered until the message is received. If the receiving thread was not the thread that probed for the message we issue a warning. With these checks we were able to detect violations to the restrictions in small test applications.

6 Detection with Artificial Data Races

6.1 General Design

The problem of the first detection is that it can only detect violations if they actually appear in a run made with MARMOT. It is desirable to also detect whether it is possible that a violation might ever appear. For the MPI calls of a hybrid application different runs may have a different execution order of the MPI calls and it might happen that certain MPI calls are issued by different threads.

The execution order is important for constraint (B) and (C) of restriction (I), for all constraints of restrictions (II), (III), (IV) and for constraint (C) of restriction (V). For all other constraints it is only of interest which thread is calling MPI or how many threads are used. Note that for the constraints of restriction (III) it is also important which threads are performing the MPI calls.

Assume that each thread writes one and the same variable in the *pre execution code* of the Wrapper. Then there is a data race[7] on this variable if it is possible that two threads are issuing an MPI call simultaneously. This idea can be used to create artificial data races that only exist if a violation to one of the constraints is given. As there are tools that are capable to detect data races it is possible to determine whether a violation of one of the constraints occurs by detecting the associated data race. This technique can be applied to all the constraints that require an unsynchronized execution of MPI calls to happen. As detecting the violations relies on detecting a data race with a third party tool, results will only be as good as the tool applied.

Another aspect of this design is that it is necessary to create these races either before the synchronization starts or to stop and restart the synchronisation for the race.

6.2 Detection of *thred level* Violation

For constraint (C) of this restriction it is simply necessary to write a variable in the *pre execution code* of each MPI call. Note that most tools that detect data races have a fixed output format and will thus only issue an error that there is a data race on a certain variable. To influence this output we can only change the name of the variable that caused the data race. For this constraint we use the variable "App_Needs_MPI_THREAD_MULTIPLE".

6.3 Detection of Wrong Communicator Usage

To detect a violation to constraint (IV) it is necessary to detect whether it is possible that two collective calls use the same communicator simultaneously. Thus we have to design a data race that only occurs when this restriction is violated. We achieved this by mapping each communicator to an index. The index is used to write the corresponding entry of an array. In this way we only get a data race if two collective calls use the same communicator simultaneously. The *pre execution code* code used has the following structure (this code is executed before the synchronisazion starts):

```
beginCritical()
id = communicator2id(comm)
endCritical()
writeCommVarIndexed(id)
```

It is necessary to create a critical section here as mapping a communicator to an index requires the usage of an internal list that stores all the known communicators and their respective indizes. This list must only be used by one thread at a time to avoid unintended data races.

Case 1	
Thread 1	Thread 2
MPI_Probe	MPI_Recv
MPI_Recv	

Case 2	
Thread 1	Thread 2
MPI_Probe	
omp barrier	omp barrier
MPI_Recv	MPI_Recv

Case 3	
Thread 1	Thread 2
MPI_Probe	MPI_Recv
omp barrier	omp barrier
MPI_Recv	

Case 4	
Thread 1	Thread 2
MPI_Probe	
omp barrier	omp barrier
–	MPI_Recv
omp barrier	omp barrier
MPI_Recv	

Table 2. Different instances of violations to restriction (III)

6.4 Detection of MPI_Probe Invalidation

To detect violations to this restriction it is necessary to create checks for the associated constraints (A),(B) and (C). To illustrate the need to check all three constraints we present four different instances of this problem in table 2. Assume that all calls are using the same values for *source*, *tag* and *comm*. For each of these instances one of the three constraints is violated. Especially note that in all cases except the fourth either constraint (A) or constraint (C) is violated. Due to the synchronization present in the fourth case constraint (B) will be violated for all runs of this case. Thus the already implemented detection for constraint (B) is sufficient. So it is only necessary to create data races for constraints (A) and (C). Detecting violations to (A) is hard as the result of the *Probe* call might differ between runs. Thus it would be necessary to know all the possible return values of the *Probe* call. We avoided this as restriction (III) only yields a warning. Thus we only created data races that do not cover all instances of the problem and that might be hard to detect by a tool. Better data races could be created by using a more complicated scheme. The simple data races use the following *pre* and *post execution codes*:

```
Post execution code for all Probe calls:
conflict_index = getNextIndex()
registerNewProbe(returned_source,returned_tag,comm,conflict_index)
endSynchronization()
writeProbeVarIndexed(conflict_index)
beginSynchronization()

Pre execution code for all Receive calls:
FORALL probes B in registered probes {
   if (matchesProbe(B,my_source,my_tag,my_comm)) {
      endSynchronization()
      writeProbeVarIndexed(B->conflict_index)
```

```
        beginSynchronization()
        unregisterProbe(B)
    }
}
```

With this design there is a data race between two simultaneously executed Receive calls if one of them is invalidating a *Probe* call. There is also a data race if a *Receive* call is called in parallel to a *Probe* call that returned *source* and *tag* values matched by the receive. This race is hard to detect as not all runs will enter the code in the if statement.

6.5 Detection of Wrong Request Usage

A data race for the constraint of restriction (II) can be constructed as in the case of restriction (IV). Again it is necessary to map each request to an index in an array of variables. Some of the *Wait* and *Test* calls can use an array of Requests. In this case one has to map each request to an index and write the respective variable.

6.6 Detection of Erroneus MPI_Finalize

In order to create checks for the constraints of restriction (I) it is only necessary to create a data race for constraint (B). This is achieved by reading a variable for each MPI call and writing it in the *pre execution code* of MPI_Finalize. Thus if the application has the possibility to execute another MPI call in parallel to MPI_Finalize then there exists a data race.

7 Results with Intel® Thread Checker

The Intel® Thread Checker is a tool capable of detecting data races and thus should be able to detect the artificial races. We used small test codes that violate one of restrictions (I)-(V) to test this approach. In almost all the tests Thread Checker detected the data race. Only some violations to restriction (III) could not be detected. We want to present the gained output for a violation to restriction (IV). The output is shown in Table 3. As the data race is caused by code within our MPI checker the output points to source code of MARMOT. In order to find out where the error occurred in the application one has to use the stack trace feature of Thread Checker.

But usage of this tool also has disadvantes. OpenMP applications using source code instrumentation of Thread Checker are only executed on one thread. Thus for our MPI checker only one thread is visible which makes it impossible to directly detect violations to the restrictions. When using binary instrumentation of Thread Checker, execution uses multiple threads but the results of the tool are less precise. We could still detect the data races in this case but the results do not contain the variable names used within MARMOT which makes it hard to interpret the results.

Intel(R) Thread Checker 3.0 command line instrumentation driver (23479) Copyright (c) 2006 Intel Corporation. All rights reserved.

ID	...	Severity Name	Count	...	Description	1st Access[Best]	...
			...				
2	...	Error	1	...	Memory write of this→App_Comm_In_Two_Collective_Calls[] at "mpo_TCheck_Races.cc":105 conflicts with a prior memory write of this→App_Comm_In_Two_Collective_Calls[] at "mpo_TCheck_Races.cc":105 (output dependence)	"mpo_TCheck_Races.cc":105	...
			...				

Table 3. Thread Checker output for a violation to restriction (IV).

8 Conclusion

We presented an approach to find bugs in hybrid OpenMP/MPI applications that violate restrictions in the MPI standard. A first implementation in the MPI checker MARMOT is able to detect violations that actually occur in a run. In order to detect race conditions and thus potential violations we introduced a technique with artifical data races. Construction of these races is only necessary for a subset of the constraints. The data races are detectable by appropriate shared memory tools. We demonstrated our approach with the Intel® Thread Checker.

References

1. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: An MPI Analysis and Checking Tool. In Joubert, G.R., Nagel, W.E., Peters, F.J., Walter, W.V., eds.: PARCO. Volume 13 of Advances in Parallel Computing., Elsevier (2003) 493–500
2. Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface.* http://www.mpi-forum.org/docs/mpi-20.ps (1997)
3. Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard.* http://www.mpi-forum.org/docs/mpi-10.ps (1995)
4. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with umpire. In: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, IEEE Computer Society (2000) 51
5. Luecke, G.R., Zou, Y., Coyle, J., Hoekstra, J., Kraeva, M.: Deadlock detection in MPI programs. Concurrency and Computation: Practice and Experience **14**(11) (2002) 911–932
6. DeSouza, J., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S.: Automated, scalable debugging of MPI programs with Intel® Message Checker. In: SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications, New York, NY, USA, ACM (2005) 78–82
7. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. **1**(1) (1992) 74–88