Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications

Roger Kowalewski and Karl Fürlinger

MNM-Team
Ludwig-Maximilians-Universität München
Oettingenstr. 67, Munich, Germany
{kowalewski, fuerling}@mnm-team.org

Abstract. The Message Passing Interface (MPI) specifies a one-sided interface for Remote Memory Access (RMA), which allows one process to specify all communication parameters for both the sending and receiving side by providing support for asynchronous reads and updates of distributed shared data. While MPI RMA communication can be highly efficient, proper synchronization of possibly conflicting accesses to shared data is a challenging task.
This paper presents a novel debugging tool that supports developers in finding latent synchronization errors. It dynamically intercepts RMA calls and reschedules them into pessimistic executions which are valid in terms of the MPI-3 standard. Given an application with a latent synchronization error, we force a manifestation of this error which can easily be detected with the help of program invariants. An experimental evaluation shows that the tool can uncover synchronization errors which would otherwise likely go unnoticed for a wide range of scenarios.

Keywords: Bug Detection, MPI, One-Sided Communication

1 Introduction

Modern remote direct memory access (RDMA) network interconnects leverage efficient MPI one-sided communication, also known as MPI RMA (remote memory access), as an important communication paradigm. In contrast to traditional message passing, RMA conceptually decouples data transfer and synchronization, enabling superior performance potential [2]. Furthermore, while message passing is natural for some problems, it can be cumbersome to use for applications with irregular communication patterns. However, the non-blocking nature of MPI RMA poses several challenges. Programmers must understand the complex synchronization model to maintain memory consistency between possibly conflicting asynchronous data accesses. Latent synchronization bugs may lead to an erroneous state manifested during one execution that may not be triggered during another execution due to the underlying MPI library, network interconnect facilities, thread interleaving, etc. Often, errors remain unnoticed for a long period of time and only occur in large-scale scenarios [1] or after porting the application to a different HPC platform [3].

As an example, MPI RMA provides only weak ordering guarantees. In Fig. 1b, a `MPI_Put` modifies a remote memory buffer `x` which is subsequently accessed by a `MPI_Get` call. Due to the non-blocking semantics, both RMA calls may complete in any order. Furthermore, they are non-atomic allowing the `MPI_Get` to fetch a partially written value by the preceding `MPI_Put`. While this semantic flexibility is a major strength of MPI and necessary to achieve high performance, it complicates the task to write portable and correct programs.

For this purpose, we present Nasty-MPI, a runtime debugging tool to support programmers in finding latent synchronization bugs within their applications. Like many other tools, it is based on the MPI profiling interface (PMPI) and can therefore be used with any MPI application. The approach takes the RMA semantics into account to schedule *pessimistic executions* of the issued RMA communications which force a manifestation of latent synchronization bugs. Because each application may have numerous of such pessimistic executions, we provide external configuration parameters to control the scheduling process of Nasty-MPI, enabling easy integration into any environment. Since Nasty-MPI has no correctness model of the target applications, the only requirement on programmers is to supply program invariants (e.g., `assert` statements) which uncover possibly forced synchronization errors.

Such a tool supports programmers already during early development stages. We have integrated the concept into the DASH library distribution [6]. DASH is a C++ template library which adopts hierarchical PGAS[1] concepts to important standard containers (arrays, matrices, etc.) and is published along with extensive unit test suites to validate the implemented containers and algorithms. Applying Nasty-MPI improves these unit tests as it significantly increases the chance to uncover latent synchronization bugs in the low-level MPI RMA communications.

In the reminder of this paper, we first summarize the MPI RMA synchronization semantics and present a formalism to model memory consistency in Sect. 2. Section 3 elaborates the concept and strategies of Nasty-MPI to uncover synchronization bugs. An experimental evaluation in Sect. 4 with small test cases compares the behavior of applications with latent synchronization bugs on different HPC platforms. It reveals that the presented approach can manifest these synchronization bugs which would otherwise likely go unnoticed. Finally, Sect. 5 summarizes related work and Sect. 6 concludes the results of this paper.

2 MPI-3 one-sided communication

MPI RMA can be applied only on a point-to-point basis, i.e., an origin process remotely accesses memory on a target process. All communication actions (puts, gets, accumulates) operate in the context of a *window*, abstracting the distributed memory between MPI processes, and are grouped into synchronization phases, called *access epochs*. No RMA operation may be issued before opening an access epoch and no completion guarantees, neither local nor remote, are available before closing an access epoch.

[1] partitioned global address space

```
MPI_Win_lock(target);
MPI_Get(&buf, ..., target);

if (buf % 2 == 0)
   buf++;
MPI_Win_unlock(target);
```

(a) Conflicting native load and `MPI_Get`.

```
MPI_Win_lock(target);
int s = 10, r = 0;
MPI_Put(&s, ..., x, target);
MPI_Get(&r, ..., x, target);
assert(r == 10);
MPI_Win_unlock(target);
```

(b) Unsynchronized Put-Get sequence.

```
int buf[100];
/* init buf */

MPI_Win_lock(target);
MPI_Put(&buf, 100 ..., target);
MPI_Win_unlock(target);
```

(c) Non-atomic Put.

Fig. 1: Application samples with synchronization bugs.

```
MPI_Win_lock(target);
...
if (buf % 2 == 0)
   buf++;


/* Defer Get*/
MPI_Get(&buf, ..., target);

MPI_Win_unlock(target);
```

(a) Deferred `MPI_Get`.

```
MPI_Win_lock(target);
int s = 10, r = 0;

/* Reverse order */
MPI_Get(&r, ..., x, target);
MPI_Win_flush(target);
MPI_Put(&s, ..., x, target);

assert(r == 10); /* Fails */
MPI_Win_unlock(target);
```

(b) Reordered Put-Get sequence.

```
/* init buf[100] */
MPI_Win_lock(target);

/* Splitting */
MPI_Put,(&buf, ..., target);
MPI_Put,(&(buf + 1), ..., target);
...
MPI_Put,(&(buf + 99), ..., target);

MPI_Win_unlock(target);
```

(c) Split non-atomic Put.

Fig. 2: Exemplified Modifications by Nasty-MPI.

MPI RMA offers two synchronization modes which are called the *active target* and *passive target* mode. In the scope of this paper, we focus on passive target mode as only the origin is involved in synchronization, which closely matches the requirements of one-sided communication. Passive target synchronization relies on a single *lock*/*unlock* model to open and close an access epoch, respectively. Nevertheless, we can adopt the proposed techniques to the active target mode, as well as to other one-sided programming models.

2.1 Challenges in MPI RMA

In MPI RMA, all communication primitives and the *lock* routine to open an access epoch are, in fact, non-blocking. Thus, memory operations within an access epoch, whether RMA or native loads/stores, may conflict with each other. In particular, we have to consider three critical properties:

Completion: RMA communication operations are not guaranteed to complete before the surrounding access epoch is explicitly synchronized. For example in Fig. 1a, the receive buffer (`buf`) for the `MPI_Get` is subsequently accessed by a native load. Both memory accesses conflict, resulting in undefined behavior.

Ordering: In general, MPI provides no ordering guarantees for RMA operations, i.e., the order in which they are applied within an access epoch is unspecified. An exception is made for *accumulates* directed to the same target, and with the same operation and basic data type. In Fig. 1b, two RMA calls

read (`MPI_Put`) and write (`MPI_Get`) on a single memory buffer, respectively. Since the operations may complete in any order, they conflict with each other.

Atomicity: In general, RMA operations are non-atomic, except *accumulates*, which guarantee element-wise atomic reads and writes to a single target if they use the same basic data type. Figure 1c shows an example where an origin copies an array, consisting of 100 integers, to a target memory. This `MPI_Put` is non-atomic and may conflict with any memory accesses, operating concurrently on the target memory location.

These guarantees are crucial in even simple concurrent protocols. Thus, writing portable and well-defined programs requires properly synchronized memory accesses on overlapping locations. MPI RMA specifies dedicated synchronization primitives for this purpose. Beside the common approach of synchronizing by distinct access epochs, MPI additionally provides *local_flush* and *flush* primitives to locally or remotely complete pending RMA operations within an access epoch. While local completion guarantees consistent memory buffers only on the origin process, remote completion guarantees memory consistency of the target memory as well [16].

2.2 Modeling Memory Consistency in MPI RMA

To model and analyze the RMA operations issued by an application, we use a formalism based on a paper written by the MPI RMA Working Group [10].

Two memory accesses a and b conflict if they target overlapping memory and are not synchronized by both a happens-before ($\xrightarrow{hb}$) [11] and a consistency edge ($\xrightarrow{co}$) [10]. The happens-before order may either be the program order, if both operations occur in a single process, or the synchronization order between two MPI processes, such as blocking send-receive pairs. A consistency edge between two operations (i.e., $a \xrightarrow{co} b$) implies that the memory effects of a may be observed by b. Consistency edges are established by the RMA synchronization primitives, as described earlier.

Utilizing this notation, we derive an execution model of all issued RMA communications in an MPI program P. All executions E over the set of RMA calls in P may be modeled as a partially ordered *happens-before graph*, formed by the transitive closure of $\xrightarrow{hb}$ and $\xrightarrow{co}$ edges. Two executions e_1 and e_2 in E are semantically equivalent if they result in the same happens-before graph. If a and b are not synchronized, they are contained in a parallel region. For example, Fig. 3 represents a happens-before graph, derived from the program in Fig. 1b. Since both RMA operations operate on overlapping memory and are within a parallel region, the program includes a synchronization bug. If we want to guarantee that both operations remotely complete in program order, one valid solution is to synchronize by an additional *flush*, which establishes the required $\xrightarrow{cohb}$ edge, as depicted in Fig. 4.

The next section explains how we exploit this formalism to uncover latent synchronization bugs.

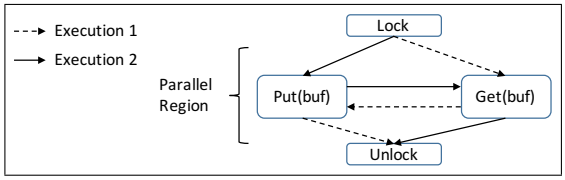

Fig. 3: Unsynchronized (two executions).

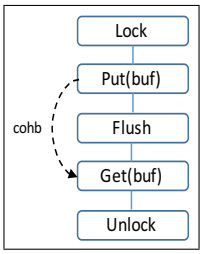

Fig. 4: Synchronized execution.

3 Forcing Synchronization Errors with Nasty-MPI

This section describes an effective approach to support programmers in debugging MPI programs with improperly synchronized RMA communications. Suppose an MPI program P contains a latent synchronization bug. Given further that P has a predefined correctness model in the form of included program invariants, as illustrated by the `assert` statements in Fig. 1b. Based on the presented memory consistency model we are able to explore different execution paths in the happens-before graph of P with the objective of finding at least one execution which forces a manifestation of this bug.

3.1 Conceptual Overview

By exploiting the PMPI interface we intercept all RMA communication actions at runtime and initially buffer them, instead of handing them over to the MPI library. This enables to dynamically construct a happens-before graph and, in particular, tracking all its parallel regions. The approach relies on the RMA completion semantics, allowing to defer the execution of communication actions to a matching synchronization call. When the application issues a synchronization action, it triggers a three-stage rescheduling process:

1) Completion Stage: We consider only those communication actions which are necessarily required to complete, as specified by the synchronization action.
2) Atomicity Stage: We break non-atomic communication actions into a set of smaller requests in such a way that the memory semantics are identical.
3) Reordering Stage: We reorder communication actions which do not conceptually give any ordering guarantees within the synchronized access epoch.

Figure 2 illustrates the rescheduling techniques when applying Nasty-MPI to the programs in Fig. 1 in the form of source code modifications that are equivalent to the effects achieved by the dynamic interception and rescheduling process.

In Fig. 2a, Nasty-MPI exploits the completion semantics and defers communication actions to a matching synchronization. Thus, the `MPI_Get` will be issued to the MPI library after the native load.

Figure 2b demonstrates the reordering technique. Suppose both RMA calls in Fig. 1b are required to complete as encountered. Since there is no synchronization to guarantee program order, we may reverse the order. Note the additional flush, issued by Nasty-MPI to force the reverse order.

The last example depicts how we utilize the atomicity semantics. In Fig. 2c, we split one single `MPI_Put` into 100 separate `MPI_Put` calls. While both variants have identical semantics, splitting RMA operations can effectively force errors which result from non-atomic memory access on overlapping locations.

In the next section, we explain the rescheduling process in more detail and discuss how the tool uses the full semantic flexibility, given by the MPI standard, to schedule pessimistic executions.

3.2 Nasty-MPI Rescheduling Process

Suppose Nasty-MPI receives a synchronization action, triggering the rescheduling process on buffered communication actions. The three stages of this rescheduling process are described in the following.

Completion Stage. Nasty-MPI first distinguishes between local and remote completion. If the issued synchronization action has remote completion semantics (i.e., *unlock* or *flush*), we filter all buffered RMA calls which are necessarily required to complete. A synchronization action can complete either all pending RMA calls within a window or to a specific target rank [16].

In the case of local completion (i.e., *flush_local*), however, all `MPI_Put` calls remain in the buffer and are not issued to the MPI library. This approach is allowed, because local completion only guarantees memory consistency of local buffers. However, because local completion creates a consistency edge between two consecutive memory access (i.e., $a \xrightarrow{co} b$), we have to copy the source buffer of a to keep it internally until remote completion is forced. This approach is applicable to RMA *accumulates* as well. However, because accumulates are conceptually ordered under certain conditions [16], we have to make sure that there are no subsequent correlated accumulates which atomically fetch data from remote memory. In this case, we are not allowed to further postpone the first accumulate operation. This strategy is useful because several experiments revealed that some MPI libraries do not distinguish between local and remote completion, i.e., they always apply remote completion. Table 1 lists two parameters for the completion stage to control, whether Nasty-MPI should apply local completion semantics (`NASTY_LOCAL_COMPLETION_ENABLED`) or even bypass the completion stage (`NASTY_SKIP_COMPLETION_STAGE`).

Atomicity Stage. While fast RMA data transfers (i.e., *put*, *get*) are non-atomic, accumulates guarantee it only on a per element granularity. Thus, we apply a splitting technique to break a single RMA call into a set of many smaller RMA calls which have identical memory semantics. We first analyze the `count` and `datatype` parameters which are contained in the signature of each RMA

call. If the `count` parameter is specified with at least 2 elements (i.e., `count >= 2`), we further determine the *extent* of a single `datatype` element. Based on these two parameters, one RMA call can be split into many single-element calls. For example, in Fig. 1c, `count` is 100 and the extent of `MPI_INT` is 4 bytes. This results in 100 `MPI_Put` calls, each having a source buffer which starts at increasing 4 bytes offsets relative to the original buffer address (see Fig. 2c).

RMA *put* and *get* calls can be even split into 1-byte RMA operations. However, we are restricted by the *displacement unit* in MPI *windows* which defines the minimum size of a single element. Thus, this approach applies only if the displacement unit is specified with a size of `MPI_BYTE` at window creation.

The atomicity stage may skipped by setting the `NASTY_SKIP_ATOMICITY_STAGE` parameter to `1`, as listed in Table 1.

Reordering Stage. Passing the first two stages gives a set of RMA calls which are *a*) required to remotely complete; and *b*) split into many small RMA calls in order to explore the minimal completion and atomicity semantics. Before we hand over these RMA calls to the native MPI library, they are finally reordered. The only restriction applies to accumulates. We can interleave them with any other communication action, however, their syntactic order has to be preserved. The default reordering approach is to randomly shuffle buffered communication actions. More fine-grained control is provided by the configuration parameter `NASTY_SUBMIT_ORDER` which can be set to any of the options in Table 2. However, simply reorder RMA operations does not guarantee that the native MPI library obeys the scheduled order. Similar to Nasty-MPI, MPI libraries are free to reorder or even apply additional optimizations, such as merging of RMA calls [7]. Thus, we must explicitly force the scheduled ordering. One option is to simulate communication latency between consecutive communication actions, giving the MPI library a chance to asynchronously process an RMA operation before the next call is issued. However, if the MPI library does not facilitate asynchronous progress mechanisms or applies lazy execution, this approach has no effect. An effective solution is to issue additional *flush* operations which is semantically valid, as we modify only parallel regions in the original happens-before graph.

The reordering stage can be further controlled by two parameters in order to configure the simulation of communication latency (`NASTY_ADD_LATENCY`) and to configure whether Nasty-MPI is allowed to inject additional *flush* synchronizations (`NASTY_ADD_FLUSH_ENABLED`).

4 Experimental Evaluation

The experiments were conducted on two HPC platforms: The NERSC Edison Cray XC 30 supercomputer [17] and SuperMUC [12] at the Leibniz Supercomputing Centre. The Cray machine is interconnected by an Aries network and provides its own MPI library and compiler, included in Cray's Message Passing Toolkit. SuperMUC facilitates a fully non-blocking Infiniband network and supports three MPI libraries: IBM (v9.1.4), Intel(v5.0) and open MPI(v1.8). The

Table 1: Nasty-MPI configuration parameters.

Parameter	Options
NASTY_SKIP_COMPLETION_STAGE	0*, 1
NASTY_LOCAL_COMPLETION_ENABLED	0, 1*
NASTY_SKIP_ATOMICITY_STAGE	0, 1*
NASTY_SUBMIT_ORDER	see Tbl. 2
NASTY_ADD_FLUSH_ENABLED	0, 1*
NASTY_ADD_LATENCY	unit32 range**

*default value **default value: 0

Table 2: Options for NASTY_SUBMIT_ORDER.

Option	Description
random	Random (default).
reverse_po	Reverse program order.
put_before_get	Schedule *put* before *get* calls.
get_before_put	Schedule *get* before *put* calls.

corresponding compiler is Intel `icc` (v15.0.4). A prototypical implementation of Nasty-MPI is publicly accessible on Github[2].

4.1 Methodology

All experiments include at least two MPI processes which communicate by improperly synchronized RMA operations. The correctness model of these applications is defined by included `assert` statements in the source code to uncover the synchronization errors.

Each experiment is evaluated with all MPI libraries in 4 scenarios, which are based on two parameters. The first parameter determines process locality, i.e., the origin and target process reside either on a single node or on two distant nodes. Process locality is an important property, because MPI libraries may hide communication latency in MPI RMA calls by utilizing shared memory semantics. The second parameter depends on whether Nasty-MPI is linked to the target application. If Nasty-MPI is linked, all applications are repeatedly executed with distinct combinations of the Nasty-MPI configuration parameters, listed in Table 1. The assumption is that, if Nasty-MPI is not linked, the MPI libraries can successfully execute the applications, i.e., the assert statements manifest no errors. In this case, there has to be at least one configuration for Nasty-MPI which forces a pessimistic execution to uncover the synchronization bug.

4.2 Effectiveness of Nasty-MPI

The first test case is a binary tree broadcast algorithm which was described by Luecke et al. [13]. The code relies on `MPI_Get` being a blocking MPI call because there is no synchronization action which actually completes it. The relevant snippet is shown in Fig. 5. Executing this program setup leads to different results, depending on the test setup. If the communicating processes, involved in the `MPI_Get`, reside on distant nodes no MPI library can successfully terminate this

[2] `https://github.com/rkowalewski/nasty-MPI`

program due to an infinite loop. But the situation changes, if both processes reside on the same node. While IBM MPI and open MPI again cannot exit from the polling loop, the implementations of Intel (SuperMUC) and Cray (NERSC Edison) can complete the RMA call. This demonstrates that process locality may impact the behavior of RMA communications, depending on the underlying MPI library. If Nasty-MPI is linked and the completion stage is not skipped (i.e., `NASTY_SKIP_COMPLETION_STAGE = 0`), the MPI library does never receive the `MPI_Get` request, because no synchronization action completes the buffered RMA call.

```
MPI_Win_lock(target);
double check = 0;
...
while (check == 0)
{
  MPI_Get(&check, ..., target);
  /* Missing Synchronization */
}
...

MPI_Win_unlock(target);
```

Fig. 5: Non-completed `MPI_Get`.

```
MPI_Win_lock_all(win);

MPI_Accumulate(...,
                predecessor, ..., win);
do {
  MPI_Fetch_and_op(..., self, ...,win);

  MPI_Win_flush(self);
} while (flag);

MPI_Win_unlock_all(win);
```

Fig. 6: Improperly synchronized Acc.

The second test case is an implementation of the MCS lock [15] which can be implemented using MPI RMA primitives [10]. In the code for acquiring the lock (Fig. 6), a requesting process issues two RMA calls which are directed to different targets. For test purposes, we have injected a synchronization error in such a way that only one target is synchronized. As listed in Table 3, all MPI libraries, except Intel, can successfully execute this program. This observation confirms that some MPI libraries always complete all pending RMA calls, regardless of the synchronization target. In Nasty-MPI, however, only the second RMA call reaches the native MPI library, while the first `MPI_Accumulate` is rejected in the completion stage, causing a manifestation of the synchronization bug.

The third test case is a slight modification from the example in Fig. 1b. The `MPI_Put` modifies a remote memory location `x` and is only locally completed by a *local_flush*. All MPI libraries pass the assert statement, i.e., the `MPI_Get` fetches the modified value by the `MPI_Put`. If Nasty-MPI is linked and the parameter `NASTY_LOCAL_COMPLETION_ENABLED` is set to `1`, it defers the `MPI_Get` to the *unlock* call, leading to a manifestation of the synchronization error.

Program 4 tests the ordering guarantees of the MPI libraries. It requires that two consecutive remote writes, one `MPI_Put` followed by an `MPI_Accumulate`, are completed in target memory as encountered by the program order. Still, there is no synchronization action to ensure this order. If the origin and target processes reside on a single node, all MPI libraries, except Intel, complete both RMA calls in program order. Nasty-MPI can easily force the synchronization bug by setting `NASTY_SUBMIT_ORDER` to `reverse_po`.

Finally, Nasty-MPI helped to detect a synchronization bug in the DASH library, while it was applied to a large test suite. The root cause was to pass a memory buffer, located on the stack frame, to a `MPI_Put`. However, the matching synchronization call was outside of the method scope, causing the memory buffer to be invalid if the RMA call is deferred to this synchronization call.

Table 3: Results of the experiments without linking Nasty-MPI.

		Edison	SuperMUC		
No.	Test Program	Cray	IBM	Intel	oMPI
1	Binary Broadcast [13]	✗	✓	✗	✓
2	MCS lock [15]	✗	✗	✓	✗
3	Local completion	✗	✗	✗	✗
4	Unordered Put calls	✗	✗	✓	✗

✓ Synchronization error manifested.
✗ Synchronization error not manifested.

5 Related Work

There is a large number of approaches for automatic bug detection in *two-sided* MPI [21,5,20,4], however, they cannot be applied to one-sided MPI due to the contrary synchronization model.

A tool, called MC-Checker [3], is closely related to this paper. It can detect memory consistency errors by profiling both MPI RMA and native memory accesses, i.e., loads and stores. Based on the MPI semantics, it effectively finds potential data races even across different origins which concurrently access overlapping target memory. However, MC-Checker only covers the MPI-2 standard which follows different synchronization semantics compared to MPI-3. Moreover, Nasty-MPI follows a different approach, since it forces synchronization errors, rather than detecting them. MUST [9] focuses on deadlocks and semantic parameter checking, which is not the scope of Nasty-MPI. However, both tools can complement each other to debug memory consistency and semantic parameter errors. Scalasca [8] detects inefficient wait states in MPI RMA applications. Another approach applies model checking [19] for deadlock detection in MPI RMA programs. Furthermore, there are tools from other PGAS languages. UPC-Thrill [18] uncovers data races in UPC programs. Significant semantic differences between UPC and MPI RMA distinguish this tool from Nasty-MPI.

6 Conclusion and Future Work

This paper discusses the semantic challenges of MPI-3 RMA and presents Nasty-MPI, a novel approach to support the detection of latent synchronization bugs in MPI applications. Based on the complex RMA semantics, we apply a systematic

strategy to force latent errors, which may be easily manifested with the help of program invariants. An experimental evaluation has demonstrated that we can uncover synchronization errors which would be otherwise go unnoticed for a wide range of synchronization scenarios. Furthermore, the tool detected a synchronization bug in the DASH library [6].

We currently evaluate to track native memory accesses by using tools, such as Pin [14]. This enables to more effectively force synchronization errors between MPI RMA and native memory accesses. Another challenge are RMA communications which use complex MPI data types (e.g., structs). Currently, we cannot apply the full potential of Nasty-MPI to such RMA operations, as it requires to understand the memory layout of complex MPI data types.

Finally, Nasty-MPI may be used by any programmer who wants to verify the semantic correctness of a given MPI RMA program.

Acknowledgments

We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

1. 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. pp. 1–10 (March 2007)
2. Bell, C., Bonachea, D., Nishtala, R., Yelick, K.: Optimizing Bandwidth Limited Problems Using One-sided Communication and Overlap. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing. pp. 84–84. IPDPS'06, IEEE Computer Society, Washington, DC, USA (2006)
3. Chen, Z., Dinan, J., Tang, Z., Balaji, P., Zhong, H., Wei, J., Huang, T., Qin, F.: MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 499–510. IEEE Press (2014)
4. Chen, Z., Li, X., Chen, J.Y., Zhong, H., Qin, F.: SyncChecker: Detecting Synchronization Errors Between MPI Applications and Libraries. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium. pp. 342–353. IPDPS '12, IEEE Computer Society, Washington, DC, USA (2012)
5. 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: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications. pp. 78–82. SE-HPCS '05, ACM, New York, NY, USA (2005)
6. Fürlinger, K., Glass, C., Knüpfer, A., Tao, J., Hünich, D., Idrees, K., Maiterth, M., Mhedheb, Y., Zhou, H.: Dash: Data structures and algorithms with support for hierarchical locality. In: Euro-Par 2014 Workshops (Porto, Portugal) (2014)
7. Gropp, W., Thakur, R.: An Evaluation of Implementation Options for MPI One-Sided Communication. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 415–424. Springer (2005)

8. Hermanns, M.A., Miklosch, M., Böhme, D., Wolf, F.: Understanding the formation of wait states in applications with one-sided communication. In: Proceedings of the 20th European MPI Users' Group Meeting. pp. 73–78. ACM (2013)
9. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 30:1–30:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
10. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., Underwood, K.: Remote Memory Access Programming in MPI-3. ACM Trans. Parallel Comput. 2(2), 9:1–9:26 (Jun 2015)
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7), 558–565 (Jul 1978)
12. Leibniz Supercomputing Centre, Munich, Germany: SuperMUC Petascale System. https://www.lrz.de/services/compute/supermuc/systemdescription/
13. Luecke, G.R., Spanoyannis, S., Kraeva, M.: The Performance and Scalability of SHMEM and MPI-2 One-sided Routines on a SGI Origin 2000 and a Cray T3E-600: Performances. Concurr. Comput. : Pract. Exper. 16(10), 1037–1060 (Aug 2004)
14. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Janapa, V., Hazelwood, R.K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. ACM Press (2005)
15. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (Feb 1991)
16. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 2012), available at: http://www.mpi-forum.org
17. National Energy Research Center, United States: Edison System Configuration. https://www.nersc.gov/users/computational-systems/edison/configuration/
18. Park, C.S., Sen, K., Hargrove, P., Iancu, C.: Efficient Data Race Detection for Distributed Memory Parallel Programs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 51:1–51:12. SC '11, ACM, New York, NY, USA (2011)
19. Pervez, S., Gopalakrishnan, G., Kirby, R., Thakur, R., Gropp, W.: Formal Verification of Programs That Use MPI One-Sided Communication. In: Mohr, B., Träff, J., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, vol. 4192, pp. 30–39. Springer Berlin Heidelberg (2006)
20. Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: ISP: A Tool for Model Checking MPI Programs. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 285–286. PPoPP '08, ACM, New York, NY, USA (2008)
21. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. SC '00, IEEE Computer Society, Washington, DC, USA (2000)