Debugging Latent Synchronization Errors in MPI-3 One-Sided Communication

Roger Kowalewski and Karl Fürlinger

Abstract The Message Passing Interface (MPI-3) provides a one-sided communication interface, also known as MPI Remote Memory Access (RMA), which enables one process to specify all required communication parameters for both the sending and receiving side. While this communication interface enables superior performance potential developers have to deal with a complex memory consistency model. Proper synchronization of asynchronous remote memory accesses to shared data structures is a challenging task. More importantly, it is difficult to pinpoint such synchronization bugs as they do not necessarily manifest in an error or occur for example only after porting the application to a different HPC environment.

We introduce a debugging tool to support the detection of latent synchronization bugs. Based on the semantic flexibility of the MPI-3 specification we dynamically modify executions of improperly synchronized MPI remote memory accesses to force a manifestation of an error. An experimental evaluation with small applications and the usage in a library which heavily relies on MPI RMA reveal that this approach can uncover synchronization bugs which would otherwise likely go unnoticed.

1 Introduction

MPI, as the de-facto standard for programming scientific applications, specifies RMA as an alternative communication approach where processes communicate shared data by one-sided *put* and *get* primitives. In contrast to traditional message-passing the target process (receiver) does not necessarily need to synchronize with the origin (sender) to complete the communication. This significantly reduces the required synchronization overhead and enables new programming models such as Partitioned Global Address Space (PGAS). PGAS provides shared memory abstractions on distributed machines to boost programmer productivity. An example is DASH [4] which is a C++ template library to specify distributed generic data structures (e.g. arrays, lists) and algorithms. It supports among other options MPI-3 RMA as the low-level communication backend.

R. Kowalewski (✉) • K. Fürlinger
Ludwig-Maximilians-Universität München, Munich, Germany
e-mail: kowalewski@nm.ifi.lmu.de

© Springer International Publishing AG 2017

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2016*,
DOI 10.1007/978-3-319-56702-0_5

(a)
```
int buf = 0;
MPI_Win_lock(target);
MPI_Get(&buf, ..., target);
buf = 1;
MPI_Win_unlock(target);
assert(buf == 1)
```

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

(c)
```
int buf[100];
/* init buf */

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

Fig. 1 Application samples with synchronization bugs. (**a**) Data race condition between native load and `MPI_Get`. (**b**) Unsynchronized Put-Get sequence. (**c**) Non-atomic Put

(a)
```
int buf = 0;
MPI_Win_lock(target);

buf = 1
/* Defer Get*/
MPI_Get(&buf, ..., target);

MPI_Win_unlock(target);
/* Assertion fails */
assert(buf == 1);
```

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

MPI_Get(&r, ..., target);
MPI_Win_flush(target);
MPI_Put(&s, ..., target);
MPI_Win_unlock(target);

/* Assertion fails */
assert(r == 10);
```

(c)
```
/* 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);
```

Fig. 2 Exemplified modifications by Nasty-MPI. (**a**) Deferred `MPI_Get`. (**b**) Reordered Put-Get sequence. (**c**) Split non-atomic Put

However, MPI RMA comes with a complex memory model which is often poorly understood and makes it difficult to precisely reason about the semantics of RMA applications, especially when changing the underlying network fabrics or MPI library. To illustrate the semantic challenges, consider the code in Fig. 1a. If we reason about the outcome based on a sequentially consistent execution the value in the local variable `buf` is `1`. However, MPI RMA provides only weak ordering guarantees meaning that the final value of `buf` may be `0, 1` or even undefined because the *get* action may happen concurrently with the local write (`buf = 1`). Figure 2a illustrates a semantically equal execution if we reason in terms of the MPI-3 specification. In order to avoid such data race conditions program developers have to properly synchronize RMA and native memory accesses. Debugging these synchronization bugs can be very time-consuming as the execution depends on the underlying hardware and scheduling interleavings at runtime.

We propose Nasty-MPI, a debugging tool to support the detection of latent synchronization errors in any MPI-3 RMA application at runtime. We apply a heuristic approach which takes the semantic flexibility given by the MPI-3 standard into account and forces *pessimistic executions* to manifest synchronization bugs. Because each application may have numerous of such pessimistic executions we provide external configuration parameters to refine the Nasty-MPI heuristic. Utilizing the PMPI interface enables easy integration into any MPI application. Since we have no semantic model of the target application we rely on supplied

program invariants (e.g. `assert` statements) raising an error if the application's semantics are not satisfied.

The remainder is organized as follows. We first explain the MPI-3 RMA synchronization semantics and present a formalism to model memory consistency in Sect. 2 to set the stage of this contribution. Section 3 elaborates the concept and strategies of Nasty-MPI to uncover synchronization errors. An experimental evaluation in Sect. 4 with small test cases compares the behavior of applications with latent synchronization bugs on different HPC platforms. We further show that applying Nasty-MPI to the extensive DASH unit test suite uncovered a latent synchronization error in the underlying MPI-3 RMA communication. Finally, Sect. 5 summarizes related work and Sect. 6 concludes.

2 MPI-3 One-Sided Communication Semantics

RMA communication can be applied only on a point-to-point basis. 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 consistency guarantees, neither local nor remote, are available before closing an access epoch.

MPI RMA offers two synchronization modes which are called the *active target* and *passive target* mode. In contrast to *passive target*, the *active target* mode requires target processes to actively synchronize with the origin to complete the communication. For this reason we focus only on *passive target* which closely matches the semantic requirements of PGAS models. The origin issues *lock*/*unlock* operations to open and close an access epoch on the target window, respectively. We can, however, adopt the concept to active target synchronization as well.

2.1 Modeling Memory Consistency

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 [8].

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 ($\xrightarrow{co}$) [8]. 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.

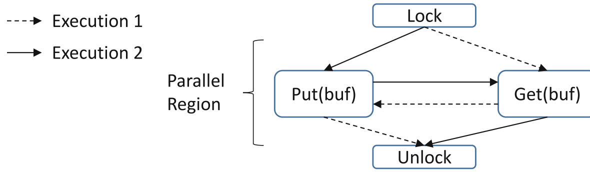

Fig. 3 Unsynchronized (two executions)

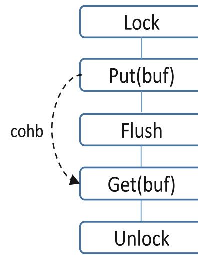

Fig. 4 Synchronized execution

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 error. 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.

2.2 Consistency Properties

After formalizing the memory consistency model of MPI-3 RMA we discuss essential semantic properties of one-sided communication actions. These properties are fundamental to satisfy correctness in even simple concurrent programs:

Atomicity Fast *put* and *get* communications are non-atomic. Only *accumulates* 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 can result in a race condition with any memory accesses operating concurrently on the target memory location.

Ordering MPI-3 provides no ordering guarantees for RMA calls in a single epoch. An exception is made for a sequence of *accumulates* directed to the same target. In addition, both the reduction operator and basic data type have to be identical among subsequent *accumulates*. In Fig. 1b, two RMA calls read (`MPI_Put`) and write (`MPI_Get`) a local memory buffer, respectively. Since the operations may complete in any order they conflict with each other.

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 store. Both memory accesses conflict, resulting in a data race condition.

In order to prevent memory consistency issues as illustrated in Fig. 1, MPI specifies dedicated primitives to synchronize pending RMA communications [15]. One approach is to synchronize by distinct access epochs. This concept fits well into the structure of many scientific applications which consist of communication and computation phases. For more fined-grained control in irregular communication patterns, such as graph problems, MPI additionally provides *flush_local* and *flush* primitives to locally or remotely complete pending RMA operation during 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.

3 Uncovering Latent Synchronization Errors

After elaborating the semantic challenges of MPI RMA we describe an effective approach to support programmers in debugging MPI programs with improperly synchronized RMA communications. Suppose an MPI program P contains a latent synchronization error. Assume further that P has a predefined correctness model in the form of included program invariants, as illustrated by the `assert` statements in Fig. 1. 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 error.

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 us to dynamically construct a happens-before graph and, in

particular, track 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 store.

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

When Nasty-MPI receives a synchronization operation it triggers the rescheduling process on buffered communication actions. The three stages of this rescheduling process are described in the following.

3.2.1 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 [15].

Table 1 Nasty-MPI configuration parameters

	Parameter	Value type	Default
1	NASTY_SKIP_COMPLETION_STAGE	bool	false
2	NASTY_LOCAL_COMPLETION_ENABLED	bool	true
3	NASTY_SKIP_ATOMICITY_STAGE	bool	false
4	NASTY_SUBMIT_ORDER	string (see Table 2)	random
5	NASTY_ADD_FLUSH_ENABLED	bool	true
6	NASTY_ADD_LATENCY	unit32_t	0

In the case of local completion (i.e. *flush_local*) 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 [15], 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. Several experiments revealed that some MPI libraries do not necessarily 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 (Table 1, line 2) or even bypass the completion stage (Table 1, line 1).

3.2.2 Atomicity Stage

While fast RMA data transfers (i.e. *put*, *get*) are non-atomic, accumulates guarantee this only on a per element granularity. 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 two elements, we further determine the *extent* of a single `datatype` element. Based on these two parameters we split a single RMA call into many single-element operations. 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. This approach applies only if the displacement unit is specified with a size of `MPI_BYTE` at window creation. Dynamic MPI

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

windows always satisfy this condition. The atomicity stage may skipped by setting the corresponding parameter (Table 1, line 3) to `true`.

3.2.3 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 reordering RMA operations does not guarantee that the native MPI library obeys the scheduled order. MPI libraries are free to reorder or even apply additional optimizations, such as merging of RMA calls [5]. 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 are 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 (Table 1, line 6) and whether Nasty-MPI is allowed to inject additional *flush* synchronizations (Table 1, line 5).

4 Experimental Evaluation

The experiments were conducted on two HPC platforms: The NERSC Edison Cray XC 30 supercomputer [16] and SuperMUC Petascale System [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 corresponding compiler is Intel `icc` (v15.0.4). A prototypical implementation of Nasty-MPI is publicly available on Github.[1]

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 four scenarios which are based on two settings. First, we have to consider 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. And second, we run each test with and without linking Nasty-MPI. If Nasty-MPI is linked, all applications are repeatedly executed with distinct combinations of the Nasty-MPI configuration parameters, listed in Table 1.

Our assumption is that without linking Nasty-MPI some, if not all, MPI libraries can successfully execute the test cases, i.e. the assert statements manifest no errors. For these cases there has to be at least one configuration for Nasty-MPI which forces a pessimistic execution to uncover the synchronization error.

4.2 Nasty-MPI Test Cases

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 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 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, the MPI library does never receive the `MPI_Get` request, because no synchronization action completes the buffered RMA call.

[1] https://github.com/dash-project/nasty-MPI.

Fig. 5 Non-completed `MPI_Get`

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

MPI_Win_unlock(target);
```

Fig. 6 Improperly synchronized Acc

```
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);
```

Table 3 Results of the experiments without linking Nasty-MPI

		NERSC Edison	LRZ SuperMUC		
	Test program	Cray	IBM	Intel	Open MPI
1	Binary broadcast [13]	✗	✓	✗	✓
2	MCS lock [14]	✗	✗	✓	✗
3	Local completion	✗	✗	✗	✗
4	Put-Put sequence	✗	✗	✓	✗

✓ Synchronization error manifested
✗ Synchronization error not manifested

The second test case is an implementation of the MCS lock [14] which can be implemented using MPI RMA primitives [8]. In the code for acquiring the lock (Fig. 6), a requesting process issues two RMA calls which are directed to different targets, namely *self* and *predecessor*. For test purposes, we have injected a synchronization error in such a way that only MPI calls to one target are 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 specified target process. 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 error.

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 *flush_local*. 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 given ordering properties of MPI libraries. It requires that two consecutive MPI_Put calls, as illustrated in Fig. 1a, are completed in target memory as encountered by the program order. However, 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 manifest the synchronization error by setting NASTY_SUBMIT_ORDER to reverse_po.

Finally, Nasty-MPI helped to detect a synchronization error in the DASH library, while it was applied to a large test suite. In dash::copy_async we asynchronously copy a strided memory block from a distant node to a local memory buffer. The aggressive splitting described in Sect. 3.2 forced a situation where the initiator of the copy operation accessed an element in the local memory buffer before the communication was completed. After fixing this issue the error is not present anymore.

4.3 Discussion

The observations show that consistency properties differ among the examined MPI libraries. Some of them provide even stronger consistency properties than required by the MPI-3 specification. However, we cannot explain all results only by the libraries themselves but have to consider the underlying network fabrics. Cray MPT uses DMAPP as communication backend and provides strong in-order guarantees based on the DMAPP_ROUTING_DETERMINISTIC attribute [3]. This attribute is a default setting on the NERSC Edison and guarantees ordering of two subsequent RMA calls if and only if both calls are directed to the same target process. Test cases 3 and 4 satisfy this condition which confirms the results, however, it does not explain the behavior in test cases 1 and 2.

On the other hand, Infiniband does not provide parametric in-order guarantees but specifies implicit ordering between two subsequent RDMA reads or writes [9]. This may explain some observations with test case 4, however, does not apply to the remaining applications on the SuperMUC system.

Summarizing the results we have shown that the concept of Nasty-MPI can effectively force various kinds of synchronization errors. While the presented test cases are no real world applications, it is a useful tool during development and can be easily integrated into any test environment. We use Nasty-MPI on a daily basis in the extensive unit test suite of the DASH library.

Regarding the additional overhead with Nasty-MPI we still have to evaluate larger scientific applications. Depending on the configuration parameters it drastically increases the number of communication and synchronization actions. In particular, additional *flush* operations which specify very expensive semantics cause significant runtime overhead. Linking the tool to the DASH unit test suite roughly

increases the execution time by 20–30%. We expect that it may get worse with more complex applications.

5 Related Work

We discuss related research focused on MPI RMA as well as other RMA programming languages.

MC-Checker [1] can dynamically 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 between different MPI processes 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, the approach is different from this work because we do not actually detect synchronization errors but rather force a manifestation based on given program invariants. UPC-Thrill [17] has similar functionality to detect data races in UPC programs. Significant semantic differences between UPC and MPI RMA distinguish the work presented here.

Another approach applies model checking [18] for deadlock and synchronization bug detection in MPI RMA programs. While it can effectively uncover latent synchronization bugs it requires to model the target application with a dedicated language.

MUST [7] is another runtime debugging tool focusing on semantic parameter checking. It detects errors which are caused by an erroneous sequence of MPI RMA calls, for example mismatched lock/unlock calls. However, it cannot uncover memory consistency errors caused by improperly synchronization RMA calls at runtime. MUST may complement with Nasty-MPI to debug both memory consistency and semantic parameter errors.

Scalasca [6] which is a well-known tool for performance optimization in two-sided MPI can detect inefficient wait states to pinpoint performance bottlenecks in MPI RMA applications.

Finally, we have related research which focuses on RMA programming models in general. Dan et al. provide a formal abstraction to model RMA languages and analyze semantic corner cases based on the specification of the hardware vendors [2]. It confirms the observations of this work that semantic guarantees heavily depend on the capabilities and configuration of the network fabrics.

6 Conclusion and Future Work

This work points out the major challenges of MPI-3 RMA communication which specifies only weak consistency guarantees. An experimental evaluation reveals that MPI libraries exploit implicit guarantees of underlying network fabrics which

may result in stronger consistency than specified by the MPI standard. This makes it challenging to write well-defined applications since a latent synchronization bug does not necessarily manifest in an error. It is even more crucial for library developers which have to provide correct semantics on any HPC platform.

For this purpose Nasty-MPI effectively supports programmers as it exploits the weak MPI RMA semantics to force pessimistic corner case executions. The observations in Sect. 4 show that this approach uncovers synchronization bugs which would otherwise only occur either after porting to an HPC platform with a different network interconnect or in large-scale scenarios. Examples include both small applications and the DASH library which supports MPI RMA as its communication backend.

Future work addresses the question whether we can guarantee to detect synchronization bugs based on formally proven scenarios. We will refine the semantic model of Nasty-MPI and verify the strategies with more productive use cases.

Acknowledgements We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA). We further want to inform that this work is an extended revision from an originally published paper [10].

References

1. Chen, Z., Dinan, J., Tang, Z., Balaji, P., Zhong, H., Wei, J., Huang, T., Qin, F.: MC-Checker: detecting memory consistency errors in MPI one-sided applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 499–510. IEEE Press, Piscataway (2014)
2. Dan, A.M., Lam, P., Hoefler, T., Vechev, M.: Modeling and analysis of remote memory access programming. In: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Amsterdam, pp. 129–144 (2016)
3. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J.: Cray cascade: a scalable HPC system based on a dragonfly network. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–9. IEEE, Washington, DC (2012)
4. Fürlinger, K., Fuchs, T., Kowalewski, R.: DASH: a C++ PGAS library for distributed data structures and parallel algorithms. In: Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications HPCC (2016)
5. 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, Berlin (2005)
6. 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, New York (2013)
7. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI runtime error detection with MUST: advances in deadlock detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp. 30:1–30:11. IEEE Computer Society Press, Los Alamitos, CA (2012)

8. 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 (2015). doi:10.1145/2780584
9. Infiniband Trade Association: InfiniBand Architecture Specification Volume 2. https://cw.infinibandta.org/document/dl/7155 (2006)
10. Kowalewski, R., Fürlinger, K.: Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications. Lecture Notes in Computer Science, pp. 51–62. Springer, Cham (2016). doi:10.1007/978-3-319-43659-3_4. http://dx.doi.org/10.1007/978-3-319-43659-3_4
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). doi:10.1145/359545.359563
12. Leibniz Supercomputing Centre, Munich, Germany: SuperMUC Petascale System. https://www.lrz.de/services/compute/supermuc/systemdescription/. Last accessed 2016
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 (2004). doi:10.1002/cpe.v16:10
14. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991). doi:10.1145/103727.103729
15. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (2012). Available at: http://www.mpi-forum.org
16. National Energy Research Center, United States: Edison System Configuration. https://www.nersc.gov/users/computational-systems/edison/configuration/. Last accessed 2016
17. Park, C.S., Sen, K., Hargrove, P., Iancu, C.: Efficient data race detection for distributed memory parallel programs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 51:1–51:12. ACM, New York (2011). doi:10.1145/2063384.2063452
18. 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). doi:10.1007/11846802_13