MC-VCChecker: A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Communication

Thanh-Dang Diep
High Performance Computing Laboratory
Faculty of Computer Science and Engineering
HCMC University of Technology, VNUHCM, Vietnam
Email: dang@hcmut.edu.vn

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678-2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—The abstract goes here.**

1. Introduction

Contemporary scientific computing applications have an increasing demand for computational power in order to understand and solve complex problems by means of the development of models and simulations. High performance computing hardware platforms are making every effort to keep pace with the demand because myriads of challenges are posed in terms of hardware and software advances. With the dramatic development of the high performance computing field, enormous new supercomputer systems in the world are built up. To date, many systems peaked at the speed of petaFLOPS [1] and bordering on the speed of exaFLOPS. These exascale systems are anticipated to come in the near future, which poses challenging tasks for scientists, developers and programmers in the development as well as the maintenance of the scientific applications.

With the advent of exascale computing [2], [3], a large number of programming models, algorithms, scientific applications utilized efficiently on the existing systems will become unfeasible as they are close on unable to be leveraged efficiently on the exascale systems. Hence, it is inevitable that there is one realistic demand for developing new programming models, algorithms, scientific applications that can be run well on such systems.

MPI (Message-Passing Interface) [4] is a de facto programming interface specification used widely to facilitate the development of scientific computing applications run on high performance computing systems. Most existing MPI programs are written by the means of two common message-passing programming mechanisms. One is MPI point-to-point communication and the other is MPI collective communication. Both mechanisms have the same characteristic of both the origin process and the target process participating in the communication and requiring the synchronization from two sides. In these mechanisms, the memory is dedicated for each process. Each time the origin process invokes the send operation and the the target process calls the receive operation, data will be copied from the memory to the system buffer and then, sent to the network, and to the memory of the target process. The major limitation of two those mechanisms is that the origin process must wait until the target process is ready to receive data before it can send data to. This causes the transferred data to be possibly postponed, which results in the drastic reduction in the program performance.

In order to overcome the aforementioned limitation, MPI developers provide a novel mechanism, called RMA (Remote Memory Access) or MPI one-sided communication [5], [6] since it requires only one process to take part in the data movement. Moreover, this mechanism is increasingly used to write scientific applications [7], [8], [9], [10] because it lets programmers take advantage of RDMA (Remote Direct Memory Access) facilities. Unlike traditional two-sided and collective communication models, MPI one-sided communication decouples data movement from synchronization, eliminating overhead from unneeded synchronization and allowing for greater concurrency. Furthermore, MPI one-sided communication helps enhance significantly program performance by dint of removing message matching and buffering overheads that are required for MPI two-sided communication, which leads to a significant reduction in communication costs. Hence, MPI one-sided communication is a promising mechanism for the exascale computing.

On the one hand the separation between data movement and synchronization is the great strength of MPI one-sided communication, but on the other, it cause programs more prone to error in comparison with MPI two-sided communication. One of notorious synchronization bugs occurring in MPI one-sided programs is memory consistency error [11]. Few debugging tools are able to detect effectively this kind of error. MC-Checker [11] is an effective debugger to solve this bug. However, MC-Checker can only capture direct process-to-process synchronization; however, for indirect synchronization, for example through send and receive operations by several different processes, result in a transitive ordering, MC-Checker cannot capture. The lack of such synchronization is a potential source of false positive.

In this paper, we present another approach called MC-VCChecker so as to deal with the limitation of MC-Checker. Generally, the design of MC-VCChecker is analogous to MC-Checker and consists of three components:

ST-Analyzer, Profiler and DN-Analyzer. However, MC-VCChecker use vector clocks similar to [12], [13] rather than happens-before relation [14] to check the concurrency between two given events. By dint of taking advantage of the vector clocks, MC-VCChecker can capture the indirect synchronization, which preserves transitive ordering, thereby eliminating the potential source of false positives.

The rest of the paper is organized as follows. The next section defines formally memory consistency errors. Section 3 reviews briefly the insight and the design of MC-Checker while Section 4 demonstrates a synchronization clock algorithm for MPI one-sided communication. The implementation of MC-VCChecker is elaborated in Section 5 along with its evaluation being depicted in Section 6. Section 7 surveys the state of the art and related work. Finally, Section 8 gives some conclusions and the further enhancement of the study.

2. Memory Consistency Errors

Memory consistency errors were specified clearly in [11]. Formally, if there are two events accessing concurrently on the same memory area and there is at least one of them that is an update operation (local or remote), there exists one memory consistency error in a MPI one-sided program execution. Two events a and b are concurrent ($\|$) when they are not ordered by consistency happens-before order ($\xrightarrow{cohb}$) [5].

$$a \parallel b \equiv a \not\xrightarrow{cohb} b \wedge b \not\xrightarrow{cohb} a \quad (1)$$

$\xrightarrow{cohb}$ between a and b is the transitive closure of the intersection of happens-before relation ($\xrightarrow{hb}$) [14] with consistency order ($\xrightarrow{co}$) [5].

$$a \xrightarrow{cohb} b \equiv a \xrightarrow{hb} b \wedge a \xrightarrow{co} b \quad (2)$$

$\xrightarrow{hb}$ between a and b is the transitive closure of the union of program order ($\xrightarrow{po}$) [5] with synchronization order ($\xrightarrow{so}$) [5].

$$a \xrightarrow{hb} b \equiv a \xrightarrow{po} b \vee a \xrightarrow{so} b \quad (3)$$

$\xrightarrow{po}$ specifies the program order of actions at the same process while $\xrightarrow{so}$ is a total order of synchronization relations between synchronization actions including external synchronizations, such as matching send/recv pairs and collective operations. $\xrightarrow{co}$ defines a partial order of the memory actions. $a \xrightarrow{co}$ guarantees that the memory effects of action a are visible before b.

In general, there are two kinds of memory consistency errors. One occurs within an epoch in same process and the other happens across processes.

P0 (Origin Process)	P1 (Target Process) **window buffer X**
MPI_Win_fence	MPI_Win_fence
MPI_Put(buf, X, …)	…
buf[0] = 8	…
…	…
MPI_Win_fence	MPI_Win_fence

Figure 1. Memory consistency error within an epoch

P0 (Origin Process)	P1 (Target Process) **window buffer X**
MPI_Barrier	MPI_Barrier
MPI_Win_lock(SHARED, P1)	MPI_Win_lock(SHARED, P1)
…	…
MPI_Put(P1, X)	**store X**
…	…
MPI_Win_unlock(P1)	MPI_Win_unlock(P1)
MPI_Barrier	MPI_Barrier

Figure 2. Memory consistency error across processes

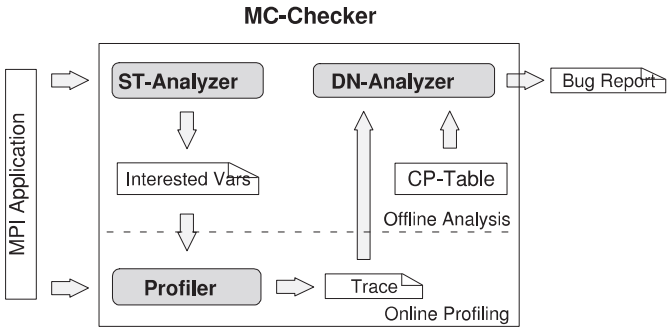

Figure 3. Design overview of MC-Checker

3. MC-Checker

There are three components in MC-Checker:

- ST-Analyzer
- Profiler
- DN-Analyzer

4. Synchronization Clock for MPI One-Sided Communication

Based on the happened-before relation [14] and consistency ordering [5], we propose a time-stamping system for MPI one-sided communication. The system preserves the consistency happens-before order between synchronization events by maintaining a vector of clocks $V_{p,i}$ in each process P_p and associates each event $e_{p,i}$ with a vector clock value $V_{p,i}[j]$. $V_{p,0}[j]$ is initialized with 0 for each component j being in the range of 0 to $n-1$. Since

the consistency happens-before order describes the ordering for synchronization events within and across processes, the vector clock algorithm needs to provide such coverage. The time-stamping system uses the following set of clock updating rules:

- *R*1. For any two consecutive synchronization events $e_{p,i}$ and $e_{p,j}$ with $j = i + 1$, $V_{p,j}[k] = V_{p,i}[k]$ for each component k; if $e_{p,i} \xrightarrow{cohb} e_{p,j}$ then $V_{p,j}[p] = V_{p,i}[p] + 1$. Assume that every first event $e_{p,i}$ occurring in all processes always issues before some *virtual event* $e_{p,0}[k]$ completes.
- *R*2. If event $e_{p,i}$ is *fence* (or barrier) and event $e_{q,j}$ is *fence* (or barrier) corresponding with $e_{p,i}$, then a message m sent from $e_{p,i}$ to $e_{q,j}$ contains a time-stamp $T_m = V_{p,i}$. Upon receiving the message m, $V_{q,j}[k] = max(V_{q,j}[k], T_m[k])$ for each component k.
- *R*3. If event $e_{p,i}$ is *post* and event $e_{q,j}$ is *start* corresponding with $e_{p,i}$, then a message m sent from $e_{p,i}$ to $e_{q,j}$ contains a time-stamp $T_m = V_{p,i}$. Upon receiving the message m, $V_{q,j}[k] = max(V_{q,j}[k], T_m[k])$ for each component k.
- *R*4. If event $e_{p,i}$ is *complete* and event $e_{q,j}$ is *wait* corresponding with $e_{p,i}$, then a message m sent from $e_{p,i}$ to $e_{q,j}$ contains a time-stamp $T_m = V_{p,i}$. Upon receiving the message m, $V_{q,j}[k] = max(V_{q,j}[k], T_m[k])$ for each component k.
- *R*5. If event $e_{p,i}$ is *send* and event $e_{q,j}$ is *recv* corresponding with $e_{p,i}$, then a message m sent from $e_{p,i}$ to $e_{q,j}$ contains a time-stamp $T_m = V_{p,i}$. Upon receiving the message m, $V_{q,j}[k] = max(V_{q,j}[k], T_m[k])$ for each component k.

From the aforementioned clock updating rules, we have the following theorem:

Theorem 1.

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \xrightarrow{cohb} e_{q,j} \iff V_{p,i}[p] \leq V_{q,j}[p]$$

Proof: Based on the consistency happens-before order [5]: **Under construction** □

Negation of Theorem 1, we have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \not\xrightarrow{cohb} e_{q,j} \iff V_{p,i}[p] > V_{q,j}[p] \quad (4)$$

Similarly, we also have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{q,j} \not\xrightarrow{cohb} e_{p,i} \iff V_{q,j}[q] > V_{p,i}[q] \quad (5)$$

Besides, in the consistency happens-before order [5], two concurrent events are defined as follows:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \parallel e_{q,j} \iff (e_{p,i} \not\xrightarrow{cohb} e_{q,j}) \wedge (e_{q,j} \not\xrightarrow{cohb} e_{p,i}) \quad (6)$$

From (4), (5), (6), we attain:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \parallel e_{q,j} \iff (V_{p,i}[p] > V_{q,j}[p]) \wedge (V_{q,j}[q] > V_{p,i}[q]) \quad (7)$$

Show an example: **Under construction**

```
0: check = 0;
1: tracels(check…);
```

Figure 4. Insertion into program's source code for assignment

```
0: if (check == 0)
1: {
2:    /* something */
3: }
4: tracels(check…);
```

Figure 5. Insertion into program's source code for branch

```
0: while (check == 0)
1: {
2:    tracels(check…);
3:    /* something */
4: }
5: tracels(check…);
```

Figure 6. Insertion into program's source code for loop

5. MC-VCChecker

Like the design of MC-Checker, MC-VCChecker also consists of there components: ST-Analyzer, Profiler and AN-Analyzer.

5.1. ST-Analyzer

Like MC-Checker, the objective of ST-Analyzer in MC-VCChecker is to identify and mark only relevant memory accesses in order to reduce unnecessary overhead in trace files and complexity in error detection. However, the implementation of ST-Analyzer in MC-VCChecker is different from MC-Checker. In particular, ST-Analyzer in MC-VCChecker performs a static analysis by means of scanning through the source code of MPI one-sided applications. In the analysis, ST-Analyzer identifies interesting variables, including MPI windows in MPI_Win_create and local buffers in MPI_Put, MPI_Get, MPI_Accumulate and stores their names in a list. When ST-Analyzer encounters a load or store operation containing a name in the list, it will insert a function call containing that name into an appropriate position in the program's source code which depends on the type of the statement containing those names. There are three types of the statement in MPI one-sided program: assignment, branch and loop. Fig. 4, Fig. 5 and Fig. 6 show how to insert the function call named "tracels" into a program's source code for each kind of corresponding statement.

```
0: int MPI_prefix (…)
1: {
2:     /* insert something here */
3:     int result = PMPI_prefix (…)
4:     /* insert something here */
5:     return result
6: }
```

Figure 7. Implementation template for instrumenting MPI calls

5.2. Profiler

The goal of Profilers in MC-Checker and MC-VCChecker are the same, which means Profiler encounters all interesting operations including MPI calls and memory accesses in an program execution and logs them in trace files. To instrument MPI calls, Profiler is implemented by making use of PMPI (MPI Profiling) [4]. With PMPI, every MPI standard call can be invoked via either MPI_prefix or PMPI_prefix. For instance, both MPI_Put and PMPI_Put can be called in order to transfer data from the origin process to the target process. By means of PMPI, a MPI call can be refined in the way of encompassing its initial behavior along with other user-defined behaviors of MPI calls for their specific purposes. An implementation template is depicted in Fig. 7. We can evidently insert additional code segments in order to implement the time-stamping system described in Section 4. Particularly, we implement MPI calls as follows:

- For MPI_Win_fence, MPI_Barrier: using PMPI_Allgather to update vector clocks
- For MPI_Win_post/complete: using PMPI_Send to update vector clocks
- For MPI_Win_start/wait: using PMPI_Recv to update vector clocks
- For MPI_Send: using PMPI_Send and PMPI_Pack to update vector clocks
- For MPI_Recv: using PMPI_Recv and PMPI_Unpack to update vector clocks

5.3. DN-Analyzer

5.3.1. Matching synchronization calls and constructing concurrent regions. In MC-Checker, DN-Analyzer reads the trace files and then constructs a Directed Acyclic Graph (DAG). A DAG is a set of concurrent regions. A concurrent region is separated from another one by MPI synchronization calls encompassing all collective calls, blocking send/receive and non-blocking send with its corresponding wait or test in the nonblocking receive process. Each concurrent region consists of a set of vertices and a set of edges. The vertices are operations in MPI one-sided programs and the edges are happens-before relations. DN-Analyzer identifies concurrent regions and detects memory consistency errors in each concurrent region by taking two given operations being not ordered by happens-before relation.

Unlike MC-Checker, MC-VCChecker does not construct DAG, but still preserve concurrent regions. Concurrent regions in MC-VCChecker are different from MC-Checker. In particular, A concurrent region in MC-Checker is separated from another one by only MPI collective calls, such as MPI_Win_fence, MPI_Barrier, and so on. Matching synchronization calls in MC-VCChecker is easier in MC-Checker in that corresponding synchronization calls in MC-VCChecker have the same value of the vector clocks.

5.3.2. Detecting conflicting operations within an epoch. A concurrent region is defined as a group of program regions across multiple processes that can be executed concurrently. Each program region consists of one or many epochs. An epoch is formed by a pair of MPI synchronization calls (MPI_Win_fence and MPI_Win_fence, MPI_Win_post and MPI_Win_wait, MPI_Win_lock and MPI_Win_unlock, etc.). Each time DN-Analyzer read a given epoch from trace files, DN-Analyzer will check whether there exist memory consistency errors withing an epoch or not by dint of the operation compatibility rules [11] for operations within an epoch: (a) if the MPI_Get is followed by any other operation, only non-overlapping accesses are permitted; (b) if MPI_Put/Acc is followed by local store or MPI_Get, only non-overlapping accesses are permitted; and (c) for all other cases, both overlapping and non-overlapping accesses are permitted. We can detect conflicting operations based on these rules.

5.3.3. Detecting conflicting operations across processes.

6. Evaluation

6.1. Experimental Setup

We conduct our experiments on two HPC platforms. One is a node belonging to the cluster which is available in our laboratory with 24 cores (48 threads) and 128 GB RAM. The other is x nodes with 128 cores and x GB RAM in SuperMUC [15] at the Leibniz Supercomputing Center. We test MC-VCChecker on three criteria: correctness, slowdown and memory usage (RAM). In order to evaluate the effectiveness, we use three MPI one-sided applications: (1) Our example program which shows the advantage of MC-VCChecker over MC-Checkereliminate in preserving the transitive ordering; (2) BT-broadcast [16] which is a binary tree broadcast algorithm using one-sided MPI discussed in a paper's appendix; (3) lockopts [17] which is an RMA test case in the MPICH library package. The last two applications consists of real-world bug cases of memory consistency errors within an epoch or across processes. We change exclusive lock to shared lock for the lockopts bug.

6.2. Results and Analysis

6.2.1. Correctness. Under construction

6.2.2. Slowdown. Under construction

6.2.3. Memory Usage. Under construction

7. Related Work

The following are some debugging tools or techniques to detect bugs in MPI one-sided communication:

- Marmot [18]
- [19]
- "Mirror Memory" [20]
- SyncChecker [21]
- MC-Checker [11]
- Nasty-MPI [22], [23]

8. Conclusions and Future Work

- The size of vector clocks

Acknowledgments

The authors would like to thank...

References

[1] "Top500," https://www.top500.org/, accessed: 2017-10-03.

[2] W. Gropp and M. Snir, "Programming for exascale computers," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 27–35, 2013.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[4] *MPI: A Message-Passing Interface Standard*, 3rd ed., Message Passing Interface Forum, June 2015.

[5] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 9, 2015.

[6] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the mpi 3.0 one-sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[7] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[8] C. Oehmen and J. Nieplocha, "Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 740–749, 2006.

[9] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia *et al.*, "Scalable earthquake simulation on petascale supercomputers," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–20.

[10] R. Thacker, G. Pringle, H. Couchman, and S. Booth, "Hydra-mpi: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures," in *High Performance Computing Systems and Applications*. NRC Research Press, 2003, p. 23.

[11] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "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*. IEEE Press, 2014, pp. 499–510.

[12] F. Mattern *et al.*, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[13] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1987.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] "Leibniz supercomputing center, munich, germany: Supermuc petascale system," https://www.lrz.de/services/compute/supermuc/systemdescription, accessed: 2018-03-06.

[16] G. R. Luecke, S. Spanoyannis, and M. Kraeva, "The performance and scalability of shmem and mpi-2 one-sided routines on a sgi origin 2000 and a cray t3e-600," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 10, pp. 1037–1060, 2004.

[17] "Mpich2: A high-performance and widely portable implementation of the message passing interface (mpi) standard," http://www.mpich.org, accessed: 2018-03-06.

[18] B. Krammer and M. M. Resch, "Correctness checking of mpi one-sided communication using marmot," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2006, pp. 105–114.

[19] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Formal verification of programs that use mpi one-sided communication," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2006, pp. 30–39.

[20] M.-Y. Park and S.-H. Chung, "Detecting race conditions in one-sided communication of mpi programs," in *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*. IEEE, 2009, pp. 867–872.

[21] Z. Chen, X. Li, J.-Y. Chen, H. Zhong, and F. Qin, "Syncchecker: Detecting synchronization errors between mpi applications and libraries," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 342–353.

[22] R. Kowalewski and K. Fürlinger, "Nasty-mpi: Debugging synchronization errors in mpi-3 one-sided applications," in *European Conference on Parallel Processing*. Springer, 2016, pp. 51–62.

[23] ——, "Debugging latent synchronization errors in mpi-3 one-sided communication," pp. 83–96, 2017.