Parallel Computing 86 (2019) 36–44

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A time-stamping system to detect memory consistency errors in MPI one-sided applications☆

Thanh-Dang Diep[a,*], Kien Trung Pham[a], Karl Fürlinger[b], Nam Thoai[a]

[a] High Performance Computing Laboratory, Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, VNUHCM, Vietnam
[b] Ludwig-Maximilians-Universität (LMU) Munich, Computer Science Department, MNM Team, Oettingenstr. 67, Munich, 80538, Germany

ARTICLE INFO

Article history:
Received 22 October 2018
Revised 27 December 2018
Accepted 28 April 2019

Keywords:
Memory consistency error
MPI
One-sided communication
Encoded vector clock
MC-CChecker

ABSTRACT

Many high performance computing applications have been developed by using MPI one-sided communication. The separation between data movement and synchronization poses enormous challenges for programmers in preserving the reliability of programs. One of those challenges is the detection of memory consistency errors, which are a notorious bug, degrading the reliability and performance of programs. Even an MPI expert can easily make these mistakes. The lockopts bug, which occurred in an RMA test case of the MPICH MPI implementation, is an example for this situation. MC-Checker is the most effective debugger in solving the memory consistency errors. MC-Checker did ignore the transitive ordering of the happened-before relation to ensure the acceptable overheads in terms of time complexity. Consequently, MC-Checker is prone to error due to the source of false positives attributable to the ignorance of the transitive ordering of the happened-before relation. To address this issue, we propose a time-stamping system based on the encoded vector clock to help preserve the full happened-before relation with reasonable overhead. The system is implemented in MC-CChecker, which is an enhancement of MC-Checker. The experimental findings prove that MC-CChecker not only effectively detects memory consistency errors like MC-Checker did, but also completely eliminates the potential source of false positives, which is a major limitation of MC-Checker while still retaining acceptable overheads of execution time and memory usage. Especially, MC-CChecker is fairly scalable when processing a large number of trace files generated from running the lockopts up to 8192 processes.

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 peak at the speed of petaFLOPS [1] and border on the speed of exaFLOPS. These exascale systems are anticipated to come in the near future, which poses challenging tasks for scientists and developers 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 unable to efficiently leverage the new exascale hardware. Hence, it is inevitable that there is a strong demand for developing new programming models, algorithms, scientific applications that can be well run on such systems.

MPI (Message Passing Interface) [4] is a programming interface specification widely used to facilitate the development of scientific computing applications run on high performance computing systems. Most existing MPI programs are written by 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 characteristics of both the sender and the receiver participating in the communication and requiring the synchronization from two sides. In these mechanisms, memory is only used locally in each process. Each time the sender invokes the send operation and the receiver calls

☆ **Conflict of interest.** None.

* Corresponding author.
E-mail addresses: dang@hcmut.edu.vn (T.-D. Diep), 51301941@hcmut.edu.vn (K.T. Pham), fuerling@nm.ifi.lmu.de (K. Fürlinger), namthoai@hcmut.edu.vn (N. Thoai).

https://doi.org/10.1016/j.parco.2019.04.013
0167-8191/

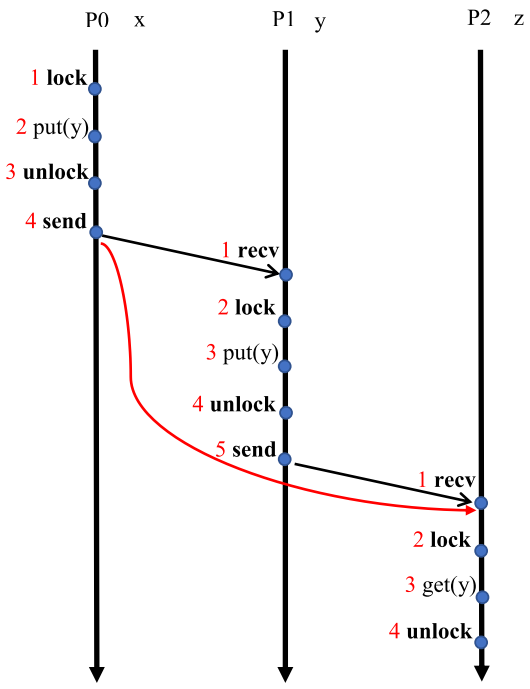


Fig. 1. fatalBug.

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 receiver. The major limitation of those two mechanisms is that the receiver must incur overhead from message matching and buffering. This results in the drastic reduction in program performance.

In order to overcome the aforementioned limitation, the MPI standard provides another mechanism, called RMA (Remote Memory Access) or MPI one-sided communication [5,6], which requires only one process to take part in the data movement. Moreover, this mechanism is increasingly used to write scientific applications [7–10], 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 greater concurrency. Hence, MPI one-sided communication is a promising mechanism for the exascale computing era.

On the one hand the separation between data movement and synchronization is a great strength of MPI one-sided communication, but on the other, it makes programs prone to errors in comparison with MPI two-sided communication. One class of notorious synchronization bugs occurring in MPI one-sided programs is memory consistency errors [11]. Few debugging tools are able to effectively detect this kind of error. MC-Checker [11] is the most effective debugger to detect this bug. However, MC-Checker can only capture direct process-to-process synchronization; Therefore, for the indirect synchronization like through send and receive operations by several different processes, which results in a transitive ordering, MC-Checker cannot capture them. The lack of such synchronization is a potential source of false positives [11]. Fig. 1 uses an event graph [12] to illustrate an example for this situation. There are three processes P_0, P_1, P_2 in the MPI one-sided program named fatalBug and *x, y, z* are window memory for each process respectively. Because MC-Checker cannot capture the transitive ordering between *put* (the second event) on P_0 and *get* (the third event) on P_2, MC-Checker reports a memory consistency error in the program. Programmers are unable to figure out the root cause in such situation while the program does not contain errors in nature.

In this paper, we present a time-stamping system to deal with the limitation of MC-Checker. The system takes advantage of the insight of the encoded vector clock [13] rather than the happened-before relation [14] to check the concurrency between two given events. By dint of making use of the encoded vector clocks, it can capture the indirect synchronization, which preserves transitive ordering, thereby eliminating the potential source of false positives. The system is implemented in MC-CChecker [15], which is an enhancement of MC-Checker.

This paper is an extended version of our earlier publication appeared at EuroMPI'18 [15]. This journal version extends the earlier paper by making the following contributions:

- Our time-stamping system is able to detect memory consistency errors in SKaMPI [16] with a fault injection.
- We also conduct several experiments on SKaMPI to investigate the performance of the proposed time-stamping system in terms of traces' size, memory usage and execution time.
- We present a variant of the encoded vector clock [13] which can mitigate overhead of the time-stamping system.

The rest of the paper is organized as follows. The next section formally defines memory consistency errors while a time-stamping system for MPI one-sided communication is described in Section 3. The application of the time-stamping system to MC-CChecker to detect the memory consistency errors is elaborated in Section 4 along with its evaluation being depicted in Section 5. Section 6 presents a variant of the encoded vector clock and discusses the extension of the time-stamping system in MPI-3 one-sided communication, and Section 7 surveys the state of the art and related work. Finally, Section 8 draws some conclusions.

2. Memory consistency errors

Memory consistency errors were clearly specified in the work of Chen et al. [11]. Formally, if there are two concurrent events accessing the same memory area and at least one of them is an update operation (local or remote), there exists a memory consistency error in an MPI one-sided program execution. An exception to this definition is made for accumulate operations that use the same operation and basic datatype. Two events *a* and *b* are concurrent ($\|_{cohb}$) when they are not ordered by consistency happens-before order ($\xrightarrow{cohb}$) [5].

$$a \|_{cohb} b \iff a \not\xrightarrow{cohb} b \wedge b \not\xrightarrow{cohb} a \tag{1}$$

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

$$a \xrightarrow{cohb} b \iff a \xrightarrow{hb} b \wedge a \xrightarrow{co} b \tag{2}$$

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

$$a \xrightarrow{hb} b \iff a \xrightarrow{po} b \vee a \xrightarrow{so} b \tag{3}$$

$\xrightarrow{po}$ 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. $a \xrightarrow{co} b$ guarantees that the memory effects of action *a* are visible before *b*.

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

Fig. 2. 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

Fig. 3. Memory consistency error across processes.

In general, there are two kinds of memory consistency errors. One occurs within an epoch in same process and the other happens across processes. Fig. 2 shows a typical example of a memory consistency error within an epoch. There are two processes in the MPI program. *MPI_Put* has the relation $\xrightarrow{po}$, but not $\xrightarrow{co}$ with the assignment (store) occurring right after. Therefore, *MPI_Put* does not have the relation $\xrightarrow{cohb}$ with the assignment and vice versa. Since both operations access variable *buf* and happen on the same process, there exists one memory consistency error within an epoch in the MPI program. Fig. 3 depicts an example of a memory consistency error across processes. Both *MPI_Put* on P_0 and the store on P_1 access the window buffer X belonging to P_1. In addition, both of the operations are concurrent and happen on different processes. Hence, there exists a memory consistency error across the processes in the MPI program.

3. Time-stamping system for MPI one-sided communication

Based on the consistency happens-before order [5] and the encoded vector clock [13], we propose a novel time-stamping system for MPI one-sided communication. The system preserves the consistency happens-before order between synchronization events by maintaining an encoded vector clock t_p in each process P_p and associates each ith event $e_{p,i}$ with an encoded vector clock value $t_{p,i}$. $t_{p,0}$ is initialized with 1 and each process P_p holds a unique prime number k_p in the whole system. Since the consistency happens-before order describes the ordering only for synchronization events within and across pairs of processes, the time-stamping algorithm needs to provide further 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 $j = i+1$, if $e_{p,i} \xrightarrow{cohb} e_{p,j}$ then $t_{p,j} = t_{p,i} * k_p$ (local tick). Assume that every first event $e_{p,i}$ occurring in all processes always issues after some *virtual event* $e_{p,0}$.
- *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 $t_m = t_{p,i}$. Upon receiving the message m, $t_{q,j}$ is set to $LCM(t_{q,j}, t_m)$. In other words, it implies that all clocks' value on every process are assigned to the least common multiple of all current clocks' value.
- *R*3/*R*4/*R*5. If event $e_{p,i}$ is *post*/*complete*/*send* and event $e_{q,j}$ is *start*/*wait*/*recv* corresponding with $e_{p,i}$, then a message m sent from $e_{p,i}$ to $e_{q,j}$ contains a time-stamp $t_m = t_{p,i}$ before executing the local tick. Upon receiving the message m, $t_{q,j}$ is set to $LCM(t_{q,j}, t_m)$.

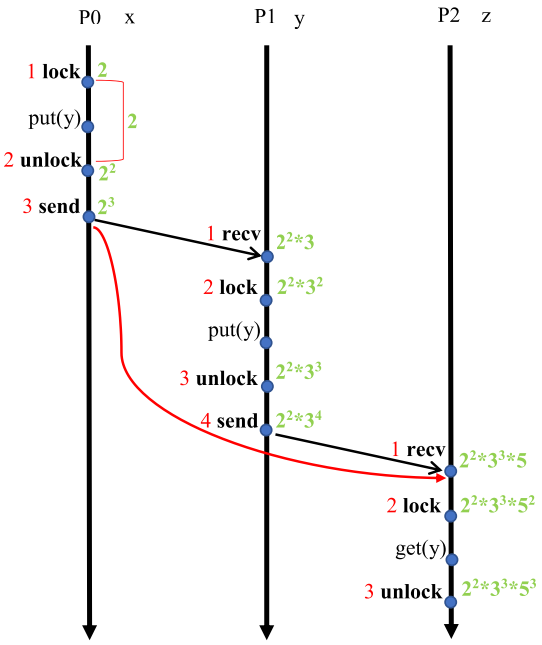


Fig. 4. fatalBug with time-stamped events.

For the sake of simplicity, we deem that *post, start, complete* and *wait* follow **strong synchronization** [4], which means *post* $\xrightarrow{cohb}$ *start* and *complete* $\xrightarrow{cohb}$ *wait*. This fact is the reason why there are rules 3 and 4. Assigning clocks' value to only synchronization operations makes sense in that its aim is to represent an area formed between two consecutive synchronization operations (including the former and excluding the later). Such an area is called *separate region*. All events' clocks within the separate region are equal to the representing synchronization event's clock. We call the set of synchronization events observed in a program execution E. From the aforementioned clock updating rules, we have the following theorem:

Theorem 1.

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \xrightarrow{cohb} e_{q,j} \iff t_{p,i} \prec t_{q,j}$$

Proof. Theorem 1 was proven in [15]. □

Negation of Theorem 1, we have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \not\xrightarrow{cohb} e_{q,j} \iff t_{p,i} \not\prec t_{q,j} \tag{4}$$

Similarly, we also have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{q,j} \not\xrightarrow{cohb} e_{p,i} \iff t_{q,j} \not\prec t_{p,i} \tag{5}$$

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

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

From (4)–(6), we attain:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \parallel_{cohb} e_{q,j} \iff (t_{p,i} \not\prec t_{q,j}) \wedge (t_{q,j} \not\prec t_{p,i}) \tag{7}$$

Fig. 4 shows fatalBug with time-stamped events. Initially, P_0, P_1, P_2 initialize t_0, t_1, t_2 with 1 and k_1, k_2, k_3 with 2, 3, 5 respectively. When the 1st event (*lock*) on P_0 occurs, a local tick is executed, therefore $t_{0,1} = t_{0,0} * 2 = 2$ (R1). In the same way, $t_{0,2} = 4$. Since $e_{0,3}$ is a *send* event, $t_{0,2}$ is piggybacked on the message sent to $e_{1,1}$ (R5). At the point $e_{1,1}$ happens, it executes a merge (R5)

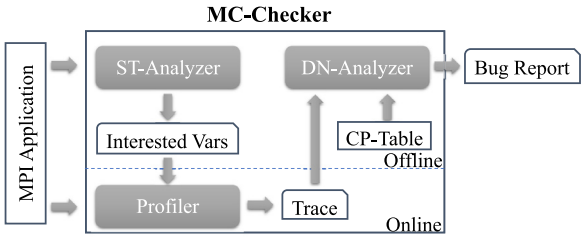


Fig. 5. Design overview of MC-Checker.

and a local tick afterward (R1), $t_{1,1} = LCM(4, 1) * 3 = 12$. Similarly, $t_{0,3} = 8$, $t_{1,2} = 36$, $t_{1,3} = 108$, $t_{1,4} = 324$, $t_{2,1} = 540$, $t_{2,2} = 2700$ and $t_{2,3} = 13500$.

4. Application of the time-stamping system to MC-CChecker in detecting memory consistency errors

Like the design of MC-Checker [11] (described in Fig. 5), MC-CChecker [15] also consists of three components: ST-Analyzer, Profiler and DN-Analyzer. MC-CChecker inherits distinguishing features of MC-Checker by reusing ST-Analyzer and Profiler of MC-Checker, but not DN-Analyzer. Hence, the main aim of MC-CChecker is to optimize DN-Analyzer by utilizing the time-stamping system for MPI one-sided communication depicted in the previous section rather than the incomplete happened-before relation used in MC-Checker.

Unlike MC-Checker, MC-CChecker does not construct a DAG. However, MC-CChecker still preserves the concurrent regions like MC-Checker. MC-CChecker loads concurrent regions one by one from trace files. Each time MC-CChecker loads one epoch totally in the concurrent region, it starts to detect memory consistency errors within that epoch in the same way as MC-Checker. However, it is different from MC-Checker in detecting memory consistency errors across processes. In particular, each time MC-CChecker loads one concurrent region totally, it starts to detect memory consistency errors across processes by examining the concurrency of each pair of separate regions for each concurrent region based on (7) in Section 3. If two separate regions are executed concurrently, MC-CChecker starts to check the accessed memory of each pair of operations belonging to the two separate regions. If the two events access the same memory, a memory consistency error across process is reported along with corresponding diagnostic information. The time-stamping system for MPI one-sided communication depicted includes 5 rules for general purpose. Nonetheless, MC-CChecker only leverages rule 1, 3, 4, 5. The reason that rule 2 is not used is to reduce unneeded overhead of clock size. In the scope of this paper, we just consider programs which use communicators being MPI_COMM_WORLD.

Fig. 6 elaborates an example for time-stamping system in MC-CChecker. There are 3 processes P_0, P_1 and P_2 running in the MPI one-sided application. *X, Y, Z* are window memory and *a, b, c* are local buffers of P_0, P_1, P_2 respectively. There are three concurrent regions in the application. The first starts when P_0, P_1 and P_2 reach the first fence in the program. Similarly, the second starts when the three processes reach the second fence. However, in the concurrent region, each time a process gets an epoch completely, it searches for memory consistency errors within the epoch. It is obvious that the second event and the third event on P_1 cause one memory consistency error within an epoch because the events both access same local buffer *b*. In addition, one memory consistency error across processes is attributable to the conflict of the second event of P_0 and the second event of P_2 since the events both access the same window memory *Y* on P_1. After obtaining the third concurrent region starting from right after the

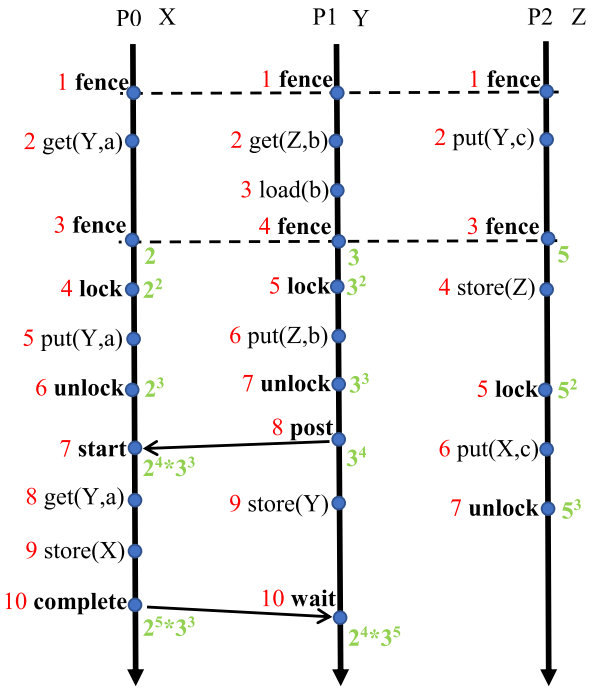


Fig. 6. The way MC-CChecker time-stamps events.

second fence of three processes to the end of the application, MC-CChecker checks each pair of separate regions. It is evident that the fifth event and the eight event of P_0 both conflict with the ninth event of P_1. Additionally, the ninth event of P_0 conflicts with the sixth event of P_2. The sixth event of P_1 conflicts with the fourth event of P_2.

5. Evaluation

5.1. Experimental setup

We conducted our experiments on two HPC platforms. One is SuperNode-XP [17] at Ho Chi Minh City University of Technology comprising 24 nodes, each equipped with two Intel Xeon E5-2680v3 CPUs with a total 24 physical cores and at least 128GB RAM per node. The other is Phase II of SuperMUC [18] at the Leibniz Supercomputing Centre consisting of 3072 nodes, each equipped with two Intel Xeon E5-2697v3 CPUs with a total of 28 physical cores and 64GB RAM per node. We tested MC-CChecker on five criteria: correctness, trace size, execution time, memory usage (RAM) and scalability. In order to implement the encoded vector clock algorithm, we used the GNU multiple precision arithmetic library (GMP), version 6.1.2 [19]. The MPI implementation used on SuperNode-XP is Intel® MPI Library for Linux* OS, Version 2017 Update 3 Build 20170405 (build id: 17193) while it is Intel® MPI Library for Linux* OS, Version 5.1.3 Build 20160120 (build id: 14053) on Phase II of SuperMUC.

In order to evaluate the correctness, we used four MPI one-sided applications: (1) fatalBug which shows the advantage of MC-CChecker over MC-Checker in preserving the transitive ordering, which eliminates the source of false positives; (2) BT-broadcast [20] which is a binary tree broadcast algorithm using one-sided MPI communication; (3) lockopts [21] which is an RMA test case in the MPICH library package with svn revision number 10308; (4) SKaMPI [16] which is a benchmark for MPI implementations with version number 5.0.4. Both BT-broadcast and lockopts consist of real-world cases of memory consistency errors within an epoch and across processes, respectively. We changed the exclusive

Table 1
Correctness of MC-CChecker.

MPI apps	Detect?	Pinpoint root cause?	Error locations	Mode	Conflicting operations	Failure symptoms	Num. of proc.
fatalBug	No						
BTbroadcast	Yes	Yes	Within an epoch	Active	Get and load	Program hang	2
lockopts-rev	Yes	Yes	Across processes	Passive	Put/get and load/store	Incorrect result	64
SKaMPI-inj	Yes	Yes	Across processes	Passive	Put and put	Incorrect result	4

lock to shared lock for the lockopts case and all communicators to MPI_COMM_WORLD for SKaMPI.

In order to evaluate the overhead of execution time and memory usage, we first ran BTbroadcast, lockopts and SKaMPI on SuperNode-XP to generate their trace files. The experiments were conducted with a various number of processes ranging from 8 to 128. After that, we used a computer with 2.4 GHz Intel® Core™ i5-2430M CPU and 4GB RAM to analyze the trace files. Furthermore, we carried out experiments to evaluate the scalability of MC-CChecker. Trace files were produced by running lockopts only on Phase II of SuperMUC. The justification is that BTbroadcast is hanged when running at least two processes on the supercomputer as on the Glenn cluster [22] at the Ohio Supercomputer Center while it does not when running on SuperNode-XP. Afterwards, the trace files were also analyzed on the computer with aforementioned hardware specification. The experiments were performed with a range of processes running from 512 to 8192. Each of the experiments in this paper was carried out five times to calculate the average value.

5.2. Results and analysis

5.2.1. Correctness

After running and analyzing four MPI applications encompassing fatalBug, BTbroadcast, lockopts and SKaMPI with MC-CChecker, we obtained the results shown in Table 1. The experimental findings show that MC-CChecker not only precisely detects all the memory consistency errors and describes bug information including error location, root cause, failure symptoms, synchronous mode, conflicting operations and the number of processes in BTbroadcast and lockopts in detail like MC-Checker but also does not cause the source of false positives when analyzing fatalBug which is a limitation of MC-Checker [11]. Hence, MC-CChecker outperforms MC-Checker in terms of the correctness. Furthermore, we changed a variable of P_0 from array data type to scalar data type in order to generate memory consistency errors across processes in the package Unlock of SKaMPI. Fig. 7 shows the bug case in SKaMPI. Processes P_1, P_2 and P_3 put data to same memory area in P_0 simultaneously, which leads to memory consistency errors across processes. MC-CChecker can detect these errors.

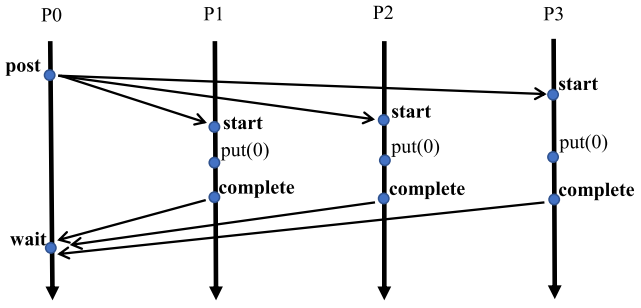


Fig. 7. Memory consistency errors across processes in SKaMPI.

5.2.2. Trace size

Because MC-CChecker reuses ST-Analyzer and Profiler of MC-Checker, trace files produced from MC-Checker and MC-CChecker have the same sizes. However, before taking other criteria into account, we first re-evaluated the size of trace files generated by MC-Checker. Fig. 8 describes the generated trace sizes after running BTbroadcast, lockopts and SKaMPI instrumented by Profiler of MC-CChecker with a number of processes ranging from 8 to 128. The trace sizes vary from 1.24 to 21.8 kB for BTbroadcast, from 15.6 to 235 kB for lockopts and from 3.2 to 761.7 MB for SKaMPI. The experimental results prove that both ST-Analyzer and Profiler in MC-Checker are efficient since it takes a small amount of disk space to store necessary information for analysis purpose and the increasing trend keeps close on linear. The reason for the low overhead is that MC-Checker logs only the necessary load/store and MPI function-level events [11].

5.2.3. Execution time

Fig. 9 depicts the execution time of BTbroadcast, lockopts and SKaMPI when running them with the number of processes varying from 8 to 128. The execution time of BTbroadcast changes from 1.49 to 16.342 ms and one of lockopts varies from 3.0708 to 32.4544 ms while it is from 735.45 to 11,276.31 ms for SKaMPI. The results prove that MC-CChecker takes a small amount of time so as to check whether MPI applications contain memory consistency errors.

5.2.4. Memory usage

Fig. 10 shows the memory usage of DN-Analyzer in MC-CChecker in order to check memory consistency errors in BTbroadcast, lockopts and SKaMPI when running them within 8 and 128 processes. The memory usage varies from 608 to 2644 kB for BTbroadcast and 603.2 to 3233.6 kB for lockopts while it is from 9.35 to 1,134.07 MB for SKaMPI. Nevertheless, there is a dramatic increase when running BTbroadcast and lockopts with 16 and 32 processes. Their memory usage when running with 32 processes takes more than three times as long as when running with 16 processes. The justification for this situation is the overhead from processing big numbers in the encoded vector clock algorithm starts to become higher in comparison with the overhead from detecting memory consistency errors. The findings show that DN-Analyzer in MC-CChecker takes a small amount of memory to check whether memory consistency errors manifested.

5.2.5. Scalability

It is noticeable that MC-CChecker is non-scalable when processing trace files generated from SKaMPI, regardless of the manifestation of anomalies in the measurement data when running SKaMPI with 16 or 32 processes. The reason is that SKaMPI contains a large number of program segments, each comprises k pairs of send-recv as shown in Fig. 12 and k is not less than 100. A previous work [23] showed that typically 21 to 25 point-to-point events were executed at some process before the encoded vector clock size exceeded the constant size used by traditional vector clocks. Because k is big enough, the values of the encoded vector clocks get quite large, which makes MC-CChecker

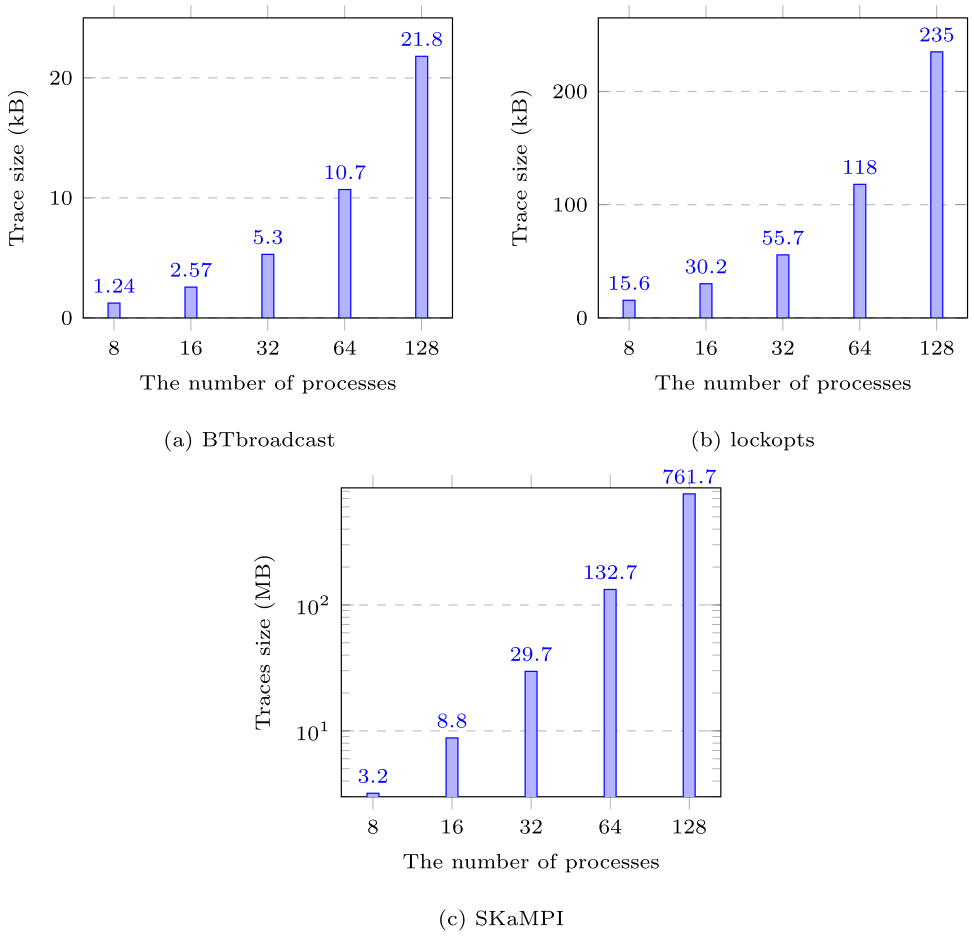


Fig. 8. Trace size.

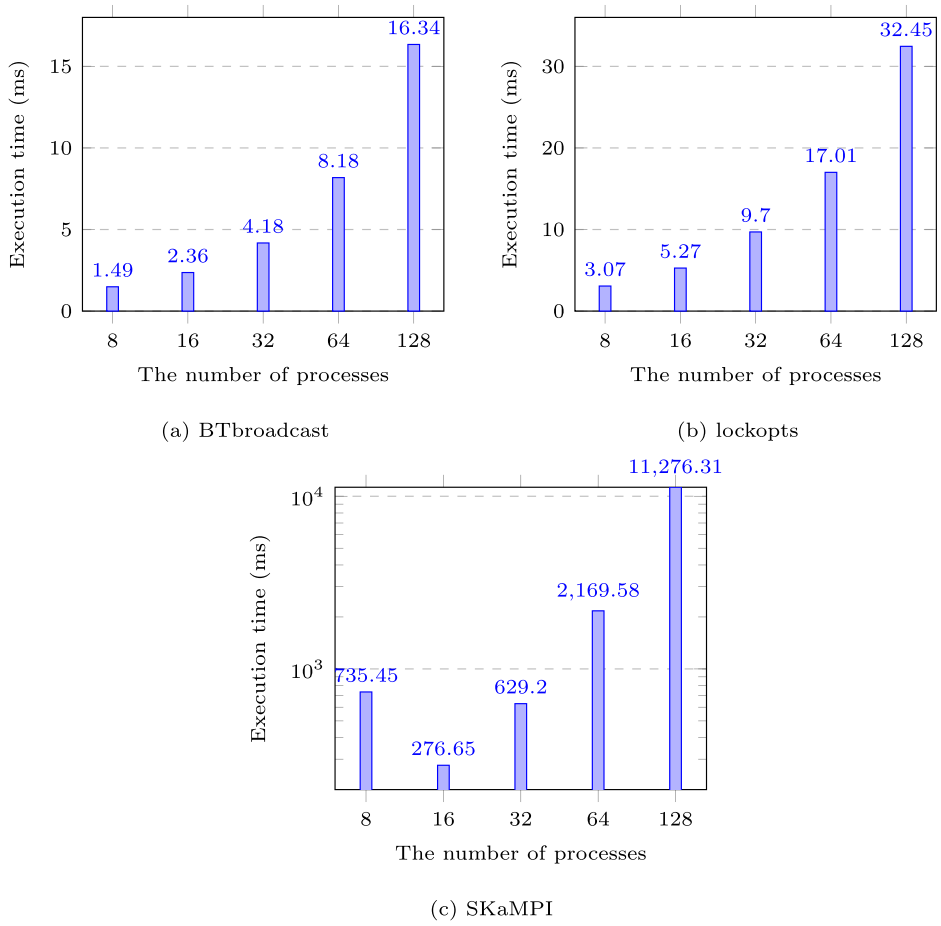
non-scalable. By contrast, it is scalable when processing trace files produced from lockopts. In order to show that MC-CChecker is scalable in this situation, we need to create a larger number of trace files by running lockopts with a larger number of processes. In particular, we ran lockopts with a number of processes ranging from 512 to 8192 to create an amount of corresponding trace files. Fig. 11 shows experimental results. In this figure, we ran two implementations of DN-Analyzer. One is DN-Analyzer using the vector clock algorithm (VC) while the other is DN-Analyzer with using the encoded vector clock (EVC). It is evident that DN-Analyzer implemented with the encoded vector clock almost matches the linear line while DN-Analyzer using the traditional vector clock goes up exponentially. The linear line means when the number of trace files increases n times, the execution time and the memory usage both increase n times as well since DN-Analyzer is a single-threaded program. Hence, DN-Analyzer of MC-CChecker is scalable when processing the large number of trace files.

6. Discussion

To alleviate the overhead of DN-Analyzer when processing trace files generated from MPI one-sided programs using a large number of point-to-point events like SKaMPI, we can use a variant of the Encoded Vector Clock as shown in Fig. 13. This version changes (1) from $t_i = 1$ to p_i while removing a local tick at (4) in comparison with the original version [13]. With k pairs of send-recv as shown in Fig. 12, $k-1$ local ticks are reduced.

For the sake of simplicity, we only focus on solving the memory consistency errors in MPI-2 one-sided communication which is a subset of MPI-3 so far. Nonetheless, we find it feasible to extend the research scope to MPI-3 calls. In particular, we take *MPI_Win_flush* and *MPI_Win_flush_local* into consideration. *MPI_Win_flush* completes both at the origin and at the target all outstanding MPI one-sided communication calls initiated by the origin process to the target process while *MPI_Win_flush_local* completes only at the origin. For that reason, *MPI_Win_flush* may be used to synchronize between two MPI one-sided communication calls while *MPI_Win_flush_local* may be used to synchronize between an MPI one-sided communication call and a local memory access. Thanks to the two synchronization calls, operations happening before and after them are ordered by $\xrightarrow{cohb}$. Fig. 14 shows an example of the situation. There are two processes P_0 and P_1 communicating each other in the program. X and Y are two memory window while a and b are local buffers on P_0 and P_1 respectively. It is obvious that *put* $\xrightarrow{cohb}$ *store* on P_0 by *flush_local* and *get* $\xrightarrow{cohb}$ *put* on P_1 by *flush*.

Based on the aforementioned observation, we realize that both *MPI_Win_flush* and *MPI_Win_flush_local* only pose local $\xrightarrow{cohb}$ between two groups of operations before and after them on the calling process. In other words, the synchronization calls only affect the detection of conflicting memory consistency errors within an epoch, not across processes. Hence, it is unnecessary to assign clocks to the operations. In addition, the only thing we need to do is to supplement appropriate $\xrightarrow{cohb}$ orderings between two groups of operations occurring before and after the synchronization operations. The other calls in MPI-3 one-sided communication can also be behaved in the similar manner as the calls we considered thus far.



(a) BTbroadcast

(b) lockopts

(c) SKaMPI

Fig. 9. Execution time.

7. Related work

There are only a few studies to detect bugs in MPI one-sided applications. MUST [24] which is an enhancement of Marmot is a debugger so as to tackle errors related to parameters of MPI one-sided calls. In addition, it assists programmers to reveal deadlocks existing in MPI one-sided programs. Another approach also effectively discloses latent deadlocks in MPI RMA programs but it detects them based on model checking [25]. Nonetheless, among the aforementioned remedies, none of them addresses memory consistency errors.

There are few solutions to resolve memory consistency errors. An effort to uncover memory consistency errors leverages mirror memory [26]. The number of entries in mirror memory and window memory is the same; however, mirror memory takes less memory than window memory. The justification for using the mirror memory is to know the current state of the corresponding window memory being 'read', 'write' or 'no_op', thereby detecting memory consistency errors precisely. However, the on-the-fly approach is unfeasible in practice since each time it remotely writes (*MPI_Put*) or reads (*MPI_Get*) into the window memory of the target process, it needs to check the mirror memory first. For that reason, it is implemented by putting *PMPI_Get* inside both *MPI_Put* and *MPI_Get* via the MPI profiling interface (PMPI) [4], which leads to performance depletion and even conflicts implementation principles of MPI one-sided communication. Moreover, the scope of this research has not considered local memory accesses (load/store) yet. Hence, it still cannot detect the conflicting operations between MPI communication calls and the local memory accesses.

To the best of our knowledge, MC-Checker is the most cutting-edge approach to deal with memory consistency errors in MPI one-sided applications. Nevertheless, MC-Checker cannot perform a complete analysis of the happened-before relation after building DAG because the complete analysis causes DN-Analyzer of MC-Checker to take much time when traversing DAG. If performing a complete analysis, MC-Checker is very difficult to scale well. Due to the incomplete utilization of the happened-before relation, MC-Checker is prone to produce false positives [11]. Our clock-based approach called MC-CChecker works as an enhancement of MC-Checker by taking full advantage of the encoded vector clock, which is proven scalable and still fully preserves the happened-before relation. Consequently, MC-CChecker outperforms MC-Checker in terms of correctness thanks to the ability to eliminate the source of false positives while still preserving acceptable overheads about runtime as well as memory, especially scalability.

Besides, there is another approach called Nasty-MPI [27,28] to address synchronization errors in MPI-3 one-sided applications. Nevertheless, the approach concentrates mainly on forcing pessimistic execution paths to occur in order to make latent bugs manifest rather than detect them. Therefore, the approach is far different from MC-CChecker.

8. Conclusions and future work

In this paper, we present a novel time-stamping system which is implemented in MC-CChecker to help overcome the limitation of MC-Checker in terms of correctness. The time-stamping system makes use of the encoded vector clock instead of the incomplete

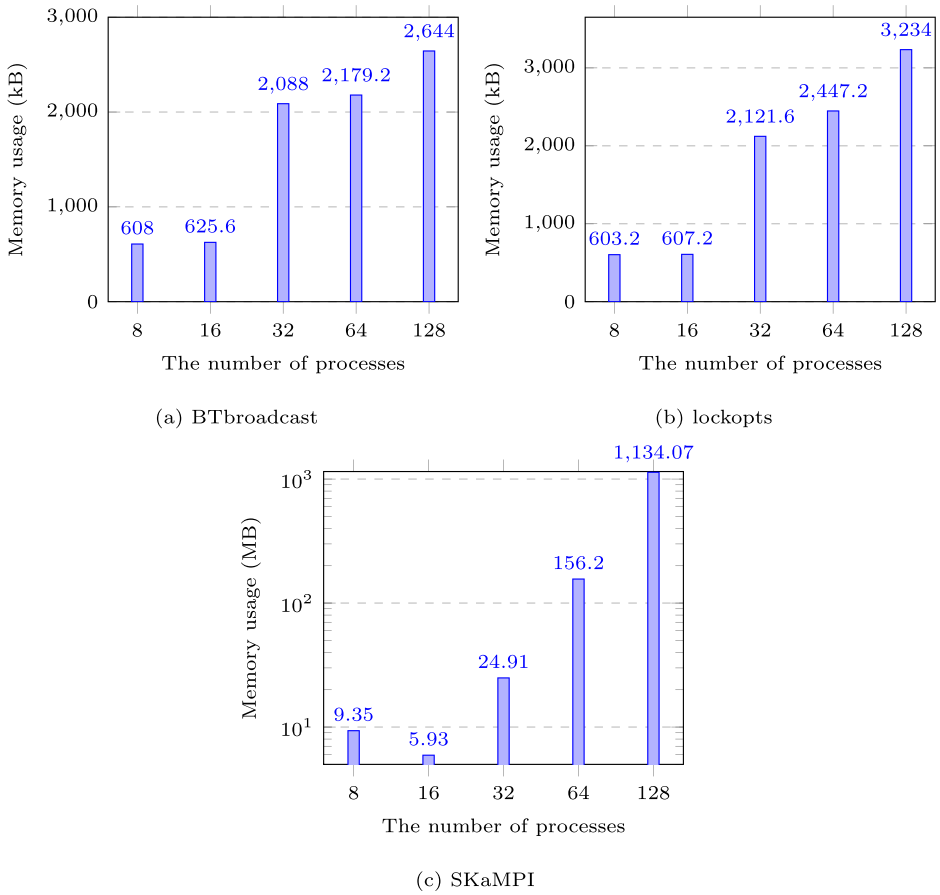


Fig. 10. Memory usage.

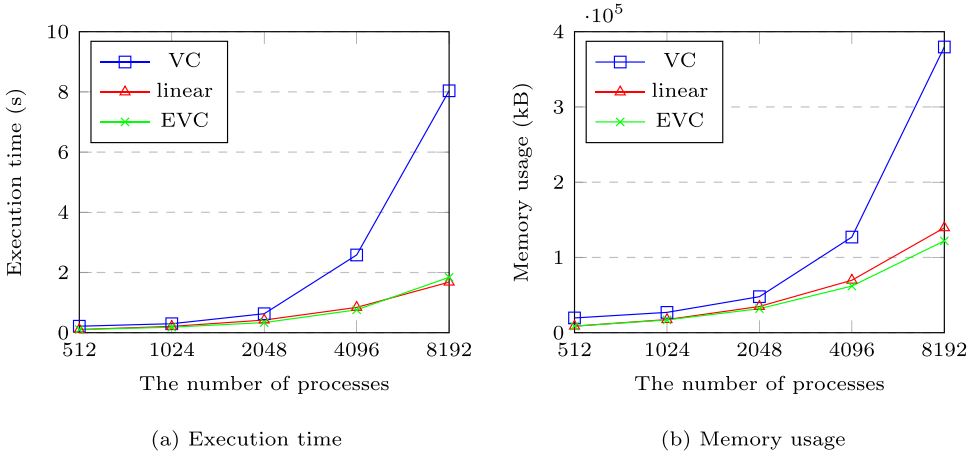


Fig. 11. Scalability when analyzing lockopts.

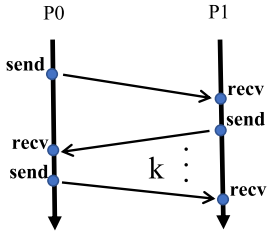


Fig. 12. The program segment containing send-recv pairs in SKaMPI.

happened-before relation utilized in MC-Checker. The experimental findings show the application of the time-stamping system to MC-CChecker completely helps eliminate the source of false positives while still preserving reasonable overheads of runtime and memory. Hence, the time-stamping system is a state-of-the-art remedy to detect memory consistency errors in MPI one-sided applications. Furthermore, we present a variant of the encoded vector clock to mitigate overhead of MC-CChecker and discuss its extension to MPI-3 one-sided communication.

Since MC-CChecker reuses ST-Analyzer of MC-Checker, MC-CChecker is still susceptible to false negatives due to the unsolved issues in respect of pointer aliasing [11]. Addressing the pointer

(1) Initialize $t_i = p_i$.
(2) Before an internal event happens at process P_i, $t_i = t_i * p_i$ (local tick).
(3) Before process P_i sends a message, it first executes $t_i = t_i * p_i$ (local tick), then it sends the message piggybacked with t_i.
(4) When process P_i receives a message piggybacked with timestamp s, it executes $t_i = LCM(s, t_i)$ (merge); before delivering the message.

Fig. 13. A variant of the encoded vector clock to mitigate overhead of the time-stamping system.

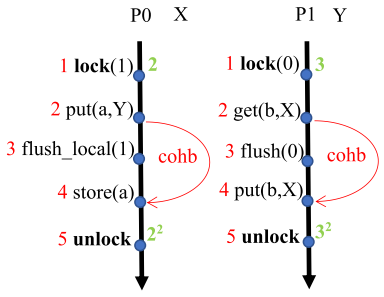


Fig. 14. The possible workings of MC-CChecker in MPI-3 one-sided applications.

aliasing is a fairly challenging problem. We intend to deal with the issue to considerably enhance the correctness of MC-CChecker. Additionally, the trace files' size affects the execution time and memory usage of DN-Analyzer. Optimizing the trace files is feasible because most MPI one-sided programs contain many repeated program segments. Hence, the optimization for trace files can help improve the execution time as well as memory usage of DN-Analyzer.

Acknowledgments

This research was funded by Department of Science and Technology - HCMC & HCMUT (under grant number 46/2018/HĐ-QKHCN). The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

References

[1] Top500, 2018,. https://www.top500.org.
[2] W. Gropp, M. Snir, Programming for exascale computers, Comput. Sci. Eng. 15 (6) (2013) 27–35.
[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomput. Front. Innov. 1 (1) (2014) 5–28.
[4] MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, 3.1 ed., 2015.
[5] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood, Remote memory access programming in MPI-3, ACM Trans. Parallel Comput. 2 (2) (2015) 9.
[6] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, R. Thakur, An implementation and evaluation of the MPI 3.0 one-sided communication interface, Concurr. Comput. 28 (17) (2016) 4385–4404.
[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, Comput. Phys. Commun. 181 (9) (2010) 1477–1489.
[8] C. Oehmen, J. Nieplocha, Scalablast: a scalable implementation of blast for high-performance data-intensive bioinformatics analysis, IEEE Trans. Parallel Distrib. Syst. 17 (8) (2006) 740–749.
[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, 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, 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] D. Kranzlmüller, Event Graph Analysis for Debugging Massively Parallel Programs, na, 2000.
[13] A.D. Kshemkalyani, A. Khokhar, M. Shen, Encoded vector clock: using primes to characterize causality in distributed systems, in: Proceedings of the 19th International Conference on Distributed Computing and Networking, ACM, 2018, p. 12.
[14] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[15] T.-D. Diep, K. Fürlinger, N. Thoai, MC-CChecker: a clock-based approach to detect memory consistency errors in MPI one-sided applications, in: Proceedings of the 25th European MPI Users' Group Meeting, ACM, 2018, p. 9.
[16] W. Augustin, M.-O. Straub, T. Worsch, Benchmarking one-sided communication with SKaMPI 5, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2005, pp. 301–308.
[17] Supernode-xp, 2018a, https://www.hpcc.hcmut.edu.vn.
[18] Leibniz Supercomputing Centre, Munich, Germany: SuperMUC petascale system, 2018, https://www.lrz.de/services/compute/supermuc/systemdescription.
[19] The GNU MP bignum library, 2018, https://www.gmplib.org.
[20] G.R. Luecke, S. Spanoyannis, M. Kraeva, The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI origin 2000 and a Cray T3E-600, Concurr. Comput. 16 (10) (2004) 1037–1060.
[21] Mpich2: a high-performance and widely portable implementation of the message passing interface (MPI) standard, 2018. http://www.mpich.org
[22] Ohio supercomputer center, 2018, http://www.osc.edu.
[23] A.D. Kshemkalyani, B. Voleti, On the growth of the prime numbers based encoded vector clock, in: International Conference on Distributed Computing and Internet Technology, Springer, 2019, pp. 169–184.
[24] T. Hilbrich, M. Schulz, B.R. de Supinski, M.S. Müller, MUST: a scalable approach to runtime error detection in MPI programs, in: Tools for high performance computing 2009, Springer, 2010, pp. 53–66.
[25] S. Pervez, G. Gopalakrishnan, R.M. Kirby, R. Thakur, 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.
[26] M.-Y. Park, 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.
[27] R. Kowalewski, K. Fürlinger, Nasty-MPI: Debugging synchronization errors in MPI-3 one-sided applications, in: European Conference on Parallel Processing, Springer, 2016, pp. 51–62.
[28] R. Kowalewski, K. Fürlinger, Debugging Latent Synchronization Errors in MPI-3 One-Sided Communication, 2017.