A Time-Stamping System to Preserve the Ordering of Events in MPI One-Sided Communication

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

Nam Thoai
High Performance Computing Lab
Faculty of Computer Science and Engineering
HCMC University of Technology, VNUHCM, Vietnam
Email: namthoai@hcmut.edu.vn

Abstract—**Extreme-scale computing poses a myriad of challenges to the development of cutting-edge programming models. MPI one-sided communication is one of promising programming models for the extreme-scale systems. Unfortunately, the conventional approaches such as the happened-before relation as well as the logical clocks generally applicable to MPI two-sided communication appear absolutely impossible in the context of the one-sided mechanism. Hence, there is an urgent need for pursuing new remedies to preserve the ordering of events in MPI one-sided communication.**

In this literature, we propose a novel event relation with the aim of pointing out the arrangement of events in a one-sided program execution. Unlike the happened-before relation, the proposed completed-before relation defines accurately that an event completed as opposed to happened before another event. Based on the event relation, we construct a rudimentary time-stamping system in order to associate vector clocks with events during a program execution. Like the traditional time-stamping systems, suggested vector clocks also show the considerable knowledge about the other processes of a process at a given time in a program execution. The potential vector clocks are to facilitate further the advancement of algorithms, techniques or protocols in MPI one-sided communication.

Index Terms—**Message Passing Interface (MPI), one-sided communication, completed-before relation, vector clock**

I. 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.

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.

The concept of time plays a vital role in distributed systems because it is nearly the only way to help determine the ordering of events. Classical time-stamping systems [11], [12], [13] are effective methodologies to model the concept. These systems can be applied completely to MPI two-sided or collective programs. Nonetheless, they are too feasible to apply to MPI programs using one-sided communication. For this reason, we intend to propose a novel time-stamping system so as to preserve the ordering of events in MPI one-sided communication.

The rest of the paper is organized as follows. The next section reviews in detail traditional vector clock techniques expressing the ordering of events in distributed systems. Section III properly formulates the system model considered in this literature while Section IV demonstrates a workable completed-before relation for MPI one-sided communication. A time-stamping system able to use in place with the concept of time for MPI one-sided communication is elaborated in Section V along with its implementation being depicted in Section VI. Finally, Section VII gives some conclusions and the further enhancement of the study.

II. Background

Happened-before relation [11] is a well-known means in order to represent the partial ordering of events happened in a program execution in distributed systems. By definition, the happened-before relation on the set of events of a system, denoted by "$\rightarrow$", is the smallest relation satisfying the following three conditions:

1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

In addition, two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$. In the happened-before relation, it is notable that $a \not\rightarrow a$ for any event a in systems in which an event can happen before itself because that does not seem physically meaningful. This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

Based on the happened-before relation, many excellent techniques mimicking physical clocks have been conducted. Logical clocks, which are an excellent notion in order to substitute for the concept of time in distributed systems, have been extensively studied and enhanced. Most approaches are either based on Lamport clocks [11] or vector clocks [12], [13].

Lamport clock is a mechanism to capture the total order between events in distributed systems. We briefly summarize the algorithm. Each process P_i holds a counter C_i initialized 0. When an event a occurs, P_i increments C_i and assigns the new value to a, where $C(a)$ denotes the value assigned to a. If a is the sending of a message m by process P_i, then the message m contains a time-stamp $T_m = C_i(a)$. On the other hand, if a is the receipt of a message m, process P_i sets C_i to a value greater than or equal to its present value and greater than T_m. Hence, we can infer the *clock condition:* for any two events a and b, if $a \rightarrow b$ then $C(a) < C(b)$.

The time-stamping algorithm proposed by Lamport has one major limitation: it does not capture the partial order between events. To address this problem, the vector clock is proposed. Its insight is that a single counter on each process P_i should be replaced by a vector of clocks with size equal to the number of processes in the execution. The vector clock algorithm insures the bi-implication of the clock condition, that is, for any two events a and b, $a \rightarrow b$ if and only if $C(a) < C(b)$. Here, $C(a) < C(b)$ means that for all i, $C[i](a) \leq C[i](b)$. The clock update rules for the vector clock algorithm are as follows. Initially, for all k, $C_i[k]$ of P_i is set to 0. Upon event a in P_i, P_i increments $C_i[i]$ and associates it with a ($C(a)$). When P_i sends a message m to P_j, it attaches C_i to m ($m.C$). When P_j receives m, it updates its clocks as follows: for all k, $C_j[k] = max(C_j[k], m.C[k])$. If two events' vector clocks are incomparable, the events are considered concurrent.

On the other hand, this increased accuracy comes at a price: the algorithm must store and send the full vector clocks, which incurs significant overhead at large scales. While some optimizations, like vector clock compression, exist, the worst case behavior of vector algorithms is still unscalable. Therefore, following the trend of proliferation in size and complexity of parallel applications, many researches were conducted, which leads to some positive results in finding another substitution clock [14], [15] or reducing the vector clock size [16], [17].

III. System model

The parallel programs considered in this literature are ones based on MPI one-sided communication. Each program consists of n processes P_0, P_1, ..., P_{n-1} communicating together by exchanging messages through some communication channels. The channels are assumed to be dependable and comply with method FIFO (First In, First Out). In order to model a program execution, an event graph model [18] is utilized for illustrative purposes. The model elaborates interesting operations in all processes and their relations. An event graph is a directed one $G = (E, \xrightarrow{os})$, where the non-empty set of events E includes the events $e_{p,i}$ of G observed in the program execution, with i representing the sequential order on process P_p. The relations between events are completed-before ones for MPI one-sided mechanism (see Section IV). If there is an edge from an event $e_{p,i}$ to an event $e_{q,j}$ in G, $e_{p,i}$ completed before $e_{q,j}$, denoted by $e_{p,i} \xrightarrow{os} e_{q,j}$. The event set E encompasses three kinds of events: communication events, local access events and synchronization events.

The set of communication events C is comprised of:

- *put*$(rmem, ldata)$: write local data $ldata$ in the origin process into exposed remote memory $rmem$ in the target process. This event is similar to the execution of a send by the origin process and a matching receive by the target process in MPI point-to-point communication.

- *get*($lmem, rdata$): read remote data $rdata$ stored in the exposed remote memory of the target process into local memory $lmem$ in the origin process. This event is similar to the execution of a send by the target process and a matching receive by the origin process in MPI point-to-point communication.
- *accumulate*($rmem, ldata$): is similar to an event put($rmem, ldata$), but combine the local data $ldata$ to the target process with data that resides in exposed remote memory $rmem$ of that process, rather than replacing it.

The set of local access events A includes:

- *load*(var): read data into variable var in the current process.
- *store*(var, val): write val into variable var in the current process.

We assume that parameter var of both *load* and *store* must also be used by at least one communication event. Operations *load* and *store* unrelated to communication events are not taken into account because they seem close to meaningless in terms of MPI one-sided communication.

Up to this point, we knew that the set of synchronization events S consists of events $e_{p,i}$ of G observed in a program execution. However, it may be more necessary to model S in another way for this kind of event set because of the characteristic of using a couple of two synchronization events $e_{p,i}$ and $e_{p,j}$ in the form of an epoch to synchronize communication operations. From a different perspective, S can also be modeled as a set of *big events* $e_{p,i-j}$ which is one of the following four kinds:

- *fence-fence*: create either an access epoch at the origin process or an exposure epoch at the target process. It is started by an event *fence* and terminated by another *fence*.
- *start-complete*: create an access epoch at the origin process in conjunction with *post-wait*. It is started by an event *start* and is terminated by an event *complete*.
- *post-wait*: create an exposure epoch at the target process in conjunction with *start-complete*. It is started by an event *post* and is terminated by an event *wait*.
- *lock-unlock*: create an access epoch at the origin process without corresponding exposure epoch. It is started by an event *lock* and is terminated by an event *unlock*.

In other words, S encompasses $e_{p,i-k}$ which each contains a *small event* set $e_{p,i}$, ..., $e_{p,j}$, ..., $e_{p,k}$ with $i < j < k$. The events $e_{p,i}$ and $e_{p,k}$ can be any couple of events belonging to the following set: $\{$(*fence,fence*), (*start,complete*), (*post,wait*), (*lock,unlock*)$\}$. In addition, the event $e_{p,j}$ belongs to either the set of communication events C, the set of local access event A or the empty set.
Further, we assume that sequence of events *start-complete-post-wait* considered in this paper follows *strong synchronization* [4]. This means that *post* occurs before the matching *start* and *complete* occurs before the matching *wait*.

Prior work [19] modeled an event's progress as a subset of four possible states in MPI point-to-point communication. However, that is eminently unsuitable for MPI one-sided communication due to the characteristic of specifying all communication parameters, both for the sending side and for the receiving side of communication events. Therefore, the modeling should be adapted to suit events in MPI one-sided communication. There are three possible states of a MPI one-sided event *op* after a process invokes *op*:

- *issued*: *op* attains this state immediately after the process invokes it. All events are issued in the program order.
- *returned*: *op* reaches this state when the process finishes executing the code of *op*.
- *completed*: *op* reaches this state when *op* no longer has any visible effects on the local program state. All blocking events reach this state immediately after they return while non-blocking ones reach this state after their next blocking one returns.

IV. Completed-before relation for MPI one-sided communication

The happened-before relation proposed by Lamport [11] is an effective method which can be applied appropriately to parallel programs using MPI two-sided communication. Nonetheless, when applying it to concurrent programs utilizing MPI one-sided communication, the relation seems completely impracticable. Only the conditions 1 and 3 are reusable while the condition 2 is of no use to the one-sided mechanism because one process can access directly data of another process without any cooperation with the latter process, which differs from the tradition two-sided programming model. For this reason, we want to propose a completed-before relation for MPI one-sided communication with the aim of modeling more effectively the ordering of fundamental events in an one-sided program execution. In general, the completed-before relation differs from the happened-before in the respect where an event completed rather than happened before another event during a program execution.

Formally, completed-before relation over one-sided event set E in graph G, denoted by "$\xrightarrow{os}$", is the smallest relation satisfying the following conditions:

- *C*1. If $e_{p,i}$ and $e_{p,j}$ are events in the same process P_p, and $e_{p,i}$ completed before $e_{p,j}$ issues, then $e_{p,i} \xrightarrow{os} e_{p,j}$.
- *C*2. If $e_{p,i} \xrightarrow{os} e_{q,j}$ and $e_{q,j} \xrightarrow{os} e_{r,k}$ then $e_{p,i} \xrightarrow{os} e_{r,k}$.

The Condition 1 leads to the following corollaries:
Corollary 1: Assume that $e_{p,*}$ is an event belonging to either S ($e_{p,i-j}$) or A, $e_{p,k}$ is an event in the same process P_p and the events both issued outside synchronization events. If $e_{p,*}$ issued before $e_{p,k}$, then $e_{p,*} \xrightarrow{os} e_{p,k}$.
Corollary 2: Assume that $e_{p,i}$ is an event belonging to A, $e_{p,j}$ is an event in the same process P_p and the events both issued inside the same synchronization event. If $e_{p,i}$ issued before $e_{p,j}$, then $e_{p,i} \xrightarrow{os} e_{p,j}$.
Corollary 3: If $e_{p,i-k}$ is an event belonging to S and $e_{p,j}$ is an event issued within $s_{p,i-k}$, $e_{p,i} \xrightarrow{os} e_{p,j}$ and $e_{p,j} \xrightarrow{os} e_{p,k}$ with $i < j < k$.
In addition, the strong synchronization results in:
Corollary 4: If $e_{p,i}$ is *post* and $e_{q,j}$ is matching *start*, $e_{q,k}$ is

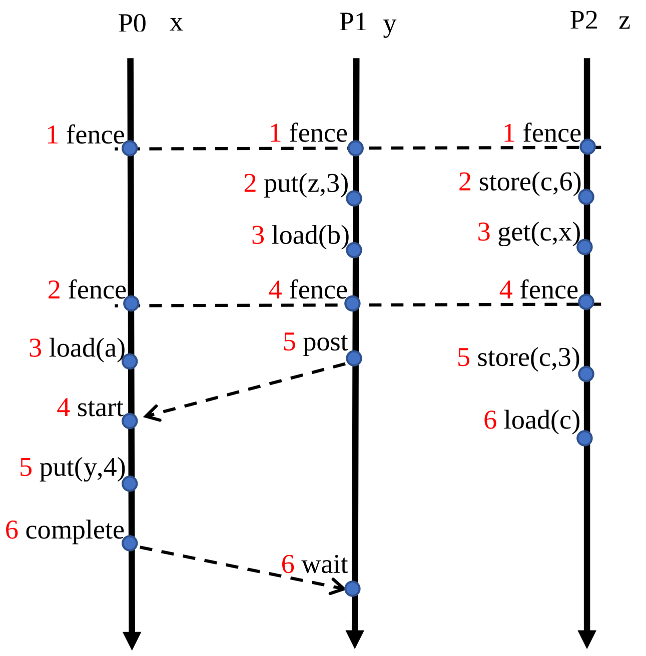

Fig. 1. Completed-before relation in MPI one-sided communication

complete and $e_{p,h}$ is corresponding *wait*, then $e_{p,i} \xrightarrow{os} e_{q,j}$ and $e_{q,k} \xrightarrow{os} e_{p,h}$ with $i < h$ and $j < k$.

Like Lamport's happened-before relation, the completed-before relation for MPI one-sided communication does also not include reflexive relation. Let $e_{p,i} \overset{os}{\not\to} e_{q,j}$ denote that $e_{p,i}$ did not complete before $e_{q,j}$ issues; then this implies that $e_{p,i} \overset{os}{\not\to} e_{p,i}$ for every event $e_{p,i}$ in an program execution. In addition, we consider $e_{p,i}$ and $e_{q,j}$ *concurrent*, denoted by "$\|$" if they are not ordered by $\xrightarrow{os}$. Therefore, $e_{p,i}$ and $e_{q,j}$ are concurrent if and only if $e_{p,i} \overset{os}{\not\to} e_{q,j}$ and $e_{q,j} \overset{os}{\not\to} e_{p,i}$.

Fig. 1 demonstrates some typical examples of the completed-before relation for MPI one-sided communication. There are three processes P_0, P_1 and P_2 exchanging data in a program execution. x, y and z, which are windows in each memory of $P0$, $P1$ and P_2 respectively, are made accessible to remote processes. a, b and c are local variables in P_0, P_1, P_2 alternately. $e_{0,1-2}$ is *fence-fence* and $e_{0,3}$ is *load(a)*, because $e_{0,1-2}$ issued before $e_{0,3}$, $e_{0,1-2} \xrightarrow{os} e_{0,3}$ based on Corollary 1. Because $e_{1,1-4}$ issued before $e_{1,5-6}$ and the events both belong to S, $e_{1,1-4} \xrightarrow{os} e_{1,5-6}$ by means of Corollary 1. As $e_{0,3}$ issued before $e_{0,4-6}$, $e_{0,3} \in A$ and $e_{0,4-6} \in S$, $e_{0,3} \xrightarrow{os} e_{0,4-6}$. Since $e_{2,5}$ issued $e_{2,6}$ and the events both belong to A, $e_{2,5} \xrightarrow{os} e_{2,6}$. $e_{2,2}$ is *store(c,6)*, $e_{2,3}$ is *get(c,x)* and $e_{2,2} \in A$; therefore, $e_{2,2} \xrightarrow{os} e_{2,3}$ based on Corollary 2. Because $e_{0,5}$ issued within $e_{0,4-6}$, $e_{0,4} \xrightarrow{os} e_{0,5}$ and $e_{0,5} \xrightarrow{os} e_{0,6}$ following Corollary 3. Based on Corollary 4, $e_{1,5} \xrightarrow{os} e_{0,4}$ and $e_{0,6} \xrightarrow{os} e_{1,6}$. As $e_{1,5} \xrightarrow{os} e_{0,4}$ and $e_{0,4} \xrightarrow{os} e_{0,5}$, $e_{1,5} \xrightarrow{os} e_{0,5}$. Additionally, $e_{1,2} \overset{os}{\not\to} e_{2,3}$ and $e_{2,3} \overset{os}{\not\to} e_{1,2}$; hence, $e_{1,2} \| e_{2,3}$.

V. A Vector Clock Algorithm for MPI One-Sided Communication

Based on the completed-before relation defined in the previous section, we also propose a rudimentary time-stamping system for MPI one-sided communication. The system preserves the completed-before relation between 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 $V_{p,i}[j]$. $V_{p,0}[j]$ set to 0 for each component j being in the range of 0 to $n-1$. Since the completed-before relation describes the ordering for 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 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 $e_{p,i} \xrightarrow{os} e_{p,j}$ 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* and event $e_{q,j}$ is *fence* 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.

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

Theorem 1:

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

Proof:

- Forward direction proof:
  The clock updating rules 1 and 2 of the vector clock protocol are derived from Corollaries 1, 2 and 3 of the completed-before relation while clock updating Rules 3 and 4 are originated by Corollary 4. Moreover, the clock updating rules all imply $V_{p,i}[p] \le V_{q,j}[p]$ in any circumstances. Consequently, $e_{p,i} \xrightarrow{os} e_{q,i} \implies V_{p,i}[p] \le V_{q,j}[p]$.
- Reverse direction proof:
  In nature, $V_{q,i}[p]$ in the time-stamping system is a counter to represent how many completed-before relations between two consecutive events occurring in process P_p which P_q know at a given time. For this reason, $V_{p,i}[p] \le V_{q,j}[p]$ shows that event $e_{q,j}$ knows the number of events occurring in process P_p more than $e_{p,i}$. Therefore, $V_{p,i}[p] \le V_{q,j}[p] \implies e_{p,i} \xrightarrow{os} e_{q,j}$.

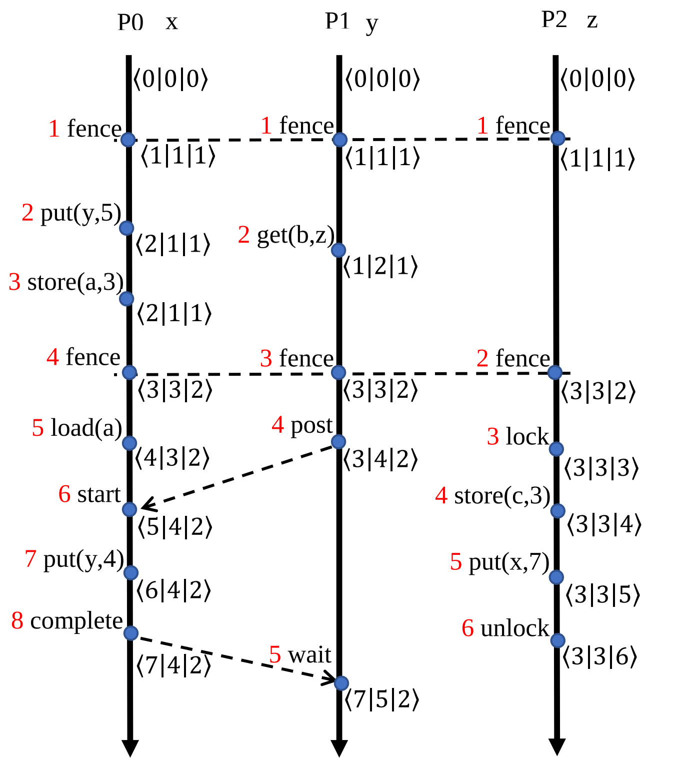

Fig. 2. Vector clock algorithm in MPI one-sided communication

From both forward and reserve direction proofs, Theorem 1 is proved completely. ■

Negation of Theorem 1, we have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \overset{gs}{\not\rightarrow} e_{q,j} \iff V_{p,i}[p] > V_{q,j}[p] \quad (1)$$

Similarly, we also have:

$$\forall e_{p,i}, e_{q,j} \in E : e_{q,j} \overset{gs}{\not\rightarrow} e_{p,i} \iff V_{q,j}[q] > V_{p,i}[q] \quad (2)$$

Besides, in the completed-before relation, two concurrent events are defined as follows:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \parallel e_{q,j} \iff (e_{p,i} \overset{gs}{\not\rightarrow} e_{q,j}) \wedge (e_{q,j} \overset{gs}{\not\rightarrow} e_{p,i}) \quad (3)$$

From (1), (2), (3), we attain:

$$\forall e_{p,i}, e_{q,j} \in E : e_{p,i} \parallel e_{q,j} \iff (V_{p,i}[p] > V_{q,j}[p]) \wedge (V_{q,j}[q] > V_{p,i}[q]) \quad (4)$$

Fig. 2 elaborates a few examples of the vector clock protocol for MPI one-sided communication. There are three processes P_0, P_1, P_2 communicating in a program execution. x, y and z, which are windows in each memory of $P0$, $P1$ and P_2 respectively, are made accessible to remote processes. a, b and c are local variables in P_0, P_1, P_2 alternately. At the outset, P_0, P_1, P_2 set their respective vector clocks $V_{0,0}$, $V_{1,0}$, $V_{2,0}$ for the same value $(0, 0, 0)$. After $e_{0,1}$, $e_{1,1}$ and $e_{2,1}$ become completed, $V_{0,1}$, $V_{1,1}$, $V_{2,1}$ are all $(1, 1, 1)$ based on Rules 1 and 2. Because $e_{0,1} \xrightarrow{os} e_{0,2}$, $V_{0,1}[0] = V_{0,2}[0] + 1$ based on Rule 1, and therefore, $V_{0,2} = (2, 1, 1)$. As $e_{0,2} \overset{gs}{\not\rightarrow} e_{0,3}$, $V_{0,3}$ has the same value with $V_{0,3}$ by dint of Rule 1. Similarly, $V_{1,2} = (1, 2, 1)$, $V_{0,4} = (3, 3, 2)$, $V_{1,3} = (3, 3, 2)$, $V_{0,2} = (3, 3, 2)$, $V_{0,5} = (4, 3, 2)$. Let us take $e_{1,4}$ and $e_{0,6}$ into consideration; Since $e_{1,4}$ is an event *post* and $e_{0,6}$ is the accompanying *start*, $V_{1,4} = (3, 4, 2)$ and $V_{0,6} = (5, 4, 2)$ by Rule 3. $V_{0,7} = (6, 4, 2)$ by means of Rule 1. As $e_{0,8}$ is an event *complete* and $e_{1,5}$ is the corresponding *wait*, $V_{0,8} = (7, 4, 2)$ as well as $V_{1,5} = (7, 5, 2)$ based on Rule 4. In a similar way, $V_{2,3} = (3, 3, 3)$, $V_{2,4} = (3, 3, 4)$, $V_{2,5} = (3, 3, 5)$, $V_{2,6} = (3, 3, 6)$. Two events $e_{0,5}$ and $e_{2,5}$ are concurrent in that $V_{0,5}[0] > V_{2,5}[0]$ and $V_{2,5}[2] > V_{0,5}[2]$.

```
int MPI_prefix (...)
{
    /* insert something here */
    int result = PMPI_prefix (...)
    /* insert something here */
    return result
}
```

Fig. 3. The implementation template for MPI calls

VI. Implementation

To implement the time-stamping system for MPI one-sided communication, we have to instrument all interesting events occurring in a program execution.

For synchronization and communication events, we use PMPI (MPI Profiling) [4] to instrument them in that they are actually MPI calls in nature. In 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 redefined in the way of encompassing its initial behavior along with other user-defined behaviors. Hence, programmers are able to interfere and tune the behaviors of MPI calls for their specific purposes. An implementation template is depicted in Fig. 3. We can evidently insert additional code segments in order to implement vector clocks for the kinds of events.

For local access events, there are two potential approaches to instrument relevant load and store events. One approach is static analysis [20] which identifies efficiently access events that need to be instrumented. This approach is to analyze each load/store instruction, branch, loop, the scope of each variable, and so on. Such complete and sound analysis is expensive and slow however, since it requires context-sensitive and parameter-sensitive inter-procedure analysis and sound pointer alias analysis. The analysis can be simplified without losing the interested load/store instructions. First, identifying all variables accessed by communication events. These variables are labeled as “relevant”. Then such labels are propagated by following pointer assignments or function calls involving pointers by following pointer assignments or function calls involving pointers. After that, all labeled variables are recorded

```
0: …
1: load/store instruction
2: signal_function( )
3: …
```

Fig. 4. The implementation template for access events

in the report. These variables are the ones we would like to instrument in the load/store instructions.

Static analysis is a complex approach due to compiler modifications. Hence, we use another approach utilized in [21]. The technique is to transform programmer's source code into a new one including signal functions. A template for implementing this is shown in Fig. 4. The signal functions contain necessary code to implement vector clocks for the local access events. The technique has longer waiting time, but simpler implementation in comparison with the static analysis.

VII. Conclusions and future Work

Logical clocks have a wide range of applications in distributed systems. An interesting application of logical clocks is the field of distributed debugging [22], [23]. To locate a bug that causes an error, programmers have to take account of the causal relationship between events in a program execution. Besides, logical clocks are applied to the area of distributed tracing [24]. By means of trace data, we are capable of analyzing the program performance. For this purpose, it is useful to get information about potential concurrency. An analysis of the time-stamped trace data of an execution can thus aid in discovering the degree of parallelism. Another use of logical clocks is the design of distributed algorithms [12]. Having in some sense the best possible approximation of global time should simplify the development of distributed algorithms and protocols.

In this paper, we define the completed-before relation which determines the ordering of events in MPI one-sided programs. Afterwards, we propose a novel basic vector clock to represent the logical time in the mechanism by dint of the completed-before relation. The vector clock has the size of the number of processes executing in during a program execution, thereby incurring a high overhead in both the memory usage and the run-time. Consequently, they need to be enhanced in size in next researches. The current approach is simply applied to MPI one-sided communication, thereby covering none of other mechanisms. Hence, we will extend the remedy in order to adapt to other MPI mechanisms encompassing not only point-to-point communication but also collective communication. Moreover, we will base several debugging and tracing techniques on the logical clock so as to detect serious performance problems of programs.

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] 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.

[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] R. Baldoni and G. Melideo, "k-dependency vectors: A scalable causality-tracking protocol," in *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on.* IEEE, 2003, pp. 219–226.

[15] J.-M. Hélary, M. Raynal, G. Melideo, and R. Baldoni, "Efficient causality-tracking timestamping," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1239–1250, 2003.

[16] A. Gidenstam and M. Papatriantafilou, "Adaptive plausible clocks," in *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on.* IEEE, 2004, pp. 86–93.

[17] F. J. Torres-Rojas, "Performance evaluation of plausible clocks," in *European Conference on Parallel Processing.* Springer, 2001, pp. 476–481.

[18] D. Kranzlmüller, *Event graph analysis for debugging massively parallel programs.* na, 2000.

[19] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Large scale verification of mpi programs using lamport clocks with lazy update," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on.* IEEE, 2011, pp. 330–339.

[20] 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.

[21] A.-T. Do-Mai, T.-D. Diep, and N. Thoai, "Message leak detection in debugging large-scale parallel applications," in *Advanced Computing and Applications (ACOMP), 2015 International Conference on.* IEEE, 2015, pp. 82–89.

[22] N. T. Thanh-Phuong Pham, "Optimizing the overhead of debugging long running parallel programs using smart clock," *Journal of Science and Technology*, vol. 52, no. 4A, pp. 91–100, 2014.

[23] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park, "Mpirace-check: Detection of message races in mpi programs," in *International Conference on Grid and Pervasive Computing.* Springer, 2007, pp. 322–333.

[24] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford, "Scalable timestamp synchronization for event traces of message-passing applications," *Parallel Computing*, vol. 35, no. 12, pp. 595–607, 2009.