On the Growth of the Prime Numbers Based Encoded Vector Clock

Ajay D. Kshemkalyani(✉) and Bhargav Voleti

University of Illinois at Chicago, Chicago, IL 60607, USA
{ajay,bvolet2}@uic.edu

Abstract. The vector clock is a fundamental tool for tracking causality in parallel and distributed applications. Unfortunately, it does not scale well to large systems because each process needs to maintain a vector of size n, where n is the total number of processes in the system. To address this problem, the encoded vector clock (EVC) was recently proposed. The EVC is based on the encoding of the vector clock using prime numbers and uses a single number to represent vector time. The EVC has all the properties of the vector clock and yet uses a single number to represent global time. However, the single number EVC tends to grow fast and may soon exceed the size of the traditional vector clock. In this paper, we evaluate the growth rate of the size of the EVC using a simulation model. The simulations show that the EVC grows relatively fast, and the growth rate depends on the mix of internal events and communication events. To overcome this drawback, the EVC can be used in conjunction with several scalability techniques that can allow the use of the EVC in practical applications. We then present a case study of detecting memory consistency errors in MPI one-sided applications using EVC.

Keywords: Causality · Vector clock · Prime numbers · Encoding
Happened-before relation · Scalability · Performance

1 Introduction

The ordering of events and states is a basic operation in the analysis of parallel and distributed executions. It is used in parallel applications such as dynamic race detection in multithreaded programs. It is used in distributed applications such as checkpointing and rollback recovery, mutual exclusion, debugging, and replication-based data stores. For example, in replication-based data stores, the ordering of reads and updates to a shared object is required to determine the object's most recent value. Logical clocks have been proposed to order events without the need for tightly synchronized physical clocks. These logical clocks order events based on the *causality* relation on events, defined by Lamport [13]. The ordering of events based on the causality relation is also required for enforcing causal consistency in data stores. Thus, tracking causality and evaluating causality between different events and between different states of a distributed execution is an important problem.

G. Fahrnberger et al. (Eds.): ICDCIT 2019, LNCS 11319, pp. 169–184, 2019.
https://doi.org/10.1007/978-3-030-05366-6_14

The simplest form of logical clocks, proposed by Lamport [13], uses a scalar clock at each process in the system. If two events are related by causality, their scalar clock values are so ordered. However, the causality relation between events cannot be inferred from the values of the scalar clocks of events. To overcome this drawback, vector clocks have been proposed [5,14]. The vector clock is a fundamental tool for tracking causality in distributed applications. Unfortunately, vector clocks do not scale well to large systems because each process needs to maintain a vector of size n, where n is the total number of processes in the system. To address this problem, the encoding of the vector clock using prime numbers to use a single number to represent vector time was proposed [9]. This encoding preserves the properties of the vector clock by maintaining only a single number – a big integer – at each process. The tick, merge, and comparison operations on the encoded vector clock (EVC) were proposed in [9]. This result also showed how to timestamp global states and how to perform operations – namely, the union, intersection, common causal past computation, and comparison – on the global states using the EVC. All these operations on the EVCs of events and on the EVCs of global states have equal or lower time complexity than the corresponding operations on traditional vector clocks in the uniform cost model. As the EVC values are big integers, the time complexities of the operations on EVCs are also expressed in the logarithmic cost model. However, these complexities are incomparable with the complexities of operations on traditional vector clocks in the uniform cost model.

Contributions: Although the EVC is a single big integer rather than a vector of integers, the drawback of the EVC is that it appears to grow fast. In this paper, using a simulation model, we evaluate the growth rate of the size of the EVC. Assuming that the integer data type used by programming languages is represented in 32 bits, we compute the number of events in the execution until the EVC size reaches $32n$, as a function of n. We also study via simulations, how many system events it takes until the size of the single big integer number EVC at some process becomes $32n$. We do this by computing the size of the EVC as a function of the number of events in the execution, for a fixed n. We also show that the growth rate of the EVC depends on the ratio of internal events to communication events, and analyze this dependency. Our simulation results confirm the intuition that the single number EVC grows fast.

To overcome the drawback that the EVC grows quite fast, we can use four techniques. These are: ticking the clock only at application-relevant events, the use of detection regions within which the EVC is tracked, resetting the EVC in the system when the size of the EVC at some process reaches a threshold such as $32n$ or when a global synchronization is performed, and using logarithms of the EVC rather than the EVC itself. A judicious use of these scalability techniques can control the size of the EVC, and can be used to guarantee that the size of the EVC never exceeds the size of the traditional vector clock.

Outline: In Sect. 2, we give the system model and present preliminaries. In Sect. 3, we give the simulation results on the growth of the EVC. Section 4 gives scalability techniques for the EVC. In Sect. 5, we discuss the application of the

scalability techniques in a case study of detecting memory consistency errors in MPI one-sided applications using EVC. We give concluding remarks in Sect. 6.

2 System Model and Preliminaries

2.1 System Model

A distributed system is modeled as an undirected graph (P, L), where P is the set of processes and L is the set of communication links connecting them. Let $n = |P|$. Between any two processes, there may be at most one logical channel over which the two processes communicate asynchronously. A logical channel from P_i to P_j is formed by paths over links in L. We do not assume FIFO logical channels; thus the messages may be delivered out of order.

The execution of process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \cdots \rangle$, where e_i^k is the k^{th} event at process P_i. An event at a process can be an *internal* event, a *message send* event, or a *message receive* event. Let $E = \bigcup_{i \in P} \{e \,|\, e \in E_i\}$ denote the set of events in a distributed execution. The causal precedence relation between events induces an irreflexive partial order on E. This relation is defined as Lamport's "happened before" relation [13], and denoted as $\rightarrow$. An execution of a distributed system is thus denoted by the tuple $(E, \rightarrow)$. Lamport designed the scalar clock, which is a function C that assigns integer timestamps to events such that if $e \rightarrow f$, then $C(e) < C(f)$. However, the drawback of scalar clocks is that $C(e) < C(f)$ does not imply that $e \rightarrow f$.

2.2 Vector Clocks

Mattern [14] and Fidge [5] designed the vector clock which assigns a vector V to each event such that: $e \rightarrow f \iff V(e) < V(f)$. This is called the *strong clock consistency condition*. Thus, the vector clock overcomes the drawback of the scalar clock. Each process P_i maintains a vector clock V. Events are timestamped by the current clock value. The vector clocks, initialized to the 0-vector, are updated by the following rules.

1. Before an internal event happens at process P_i, $V[i] = V[i] + 1$ (local tick).
2. Before process P_i sends a message, it first executes $V[i] = V[i] + 1$ (local tick), then it sends the message piggybacked with V.
3. When process P_i receives a message piggybacked with timestamp U, it executes
   $\forall k \in [1 \ldots n], V[k] = \max(V[k], U[k])$ (merge);
   $V[i] = V[i] + 1$ (local tick)
   before delivering the message.

The vector clock is a fundamental tool to characterize causality in distributed executions [10,17]. Charron-Bost has shown that to capture the partial order $(E, \rightarrow)$, the size of the vector clock is the dimension of the partial order [2], which is bounded by the size of the system, n. Thus, each process needs to maintain a

vector of size n to represent the local vector clock. Unfortunately, this does not scale well to large systems. Several works in the literature attempted to reduce the size of vector clocks [11,12,15,19–21], they had to make some compromises in accuracy or alter the system model, and in the worst-case, were as lengthy as vector clocks. To address this problem, the encoding of the vector clock using prime numbers to use a single number to represent vector time was proposed [9].

1. Initialize $t_i = 1$.
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);
   $t_i = t_i * p_i$ (local tick)
   before delivering the message.

Fig. 1. Operation of EVC t_i at process P_i [9].

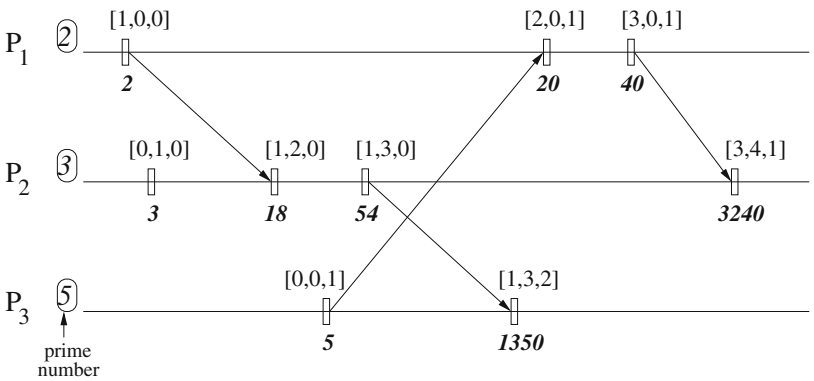

Fig. 2. Illustration of using EVC. The vector timestamps and EVC timestamps are shown above and below each timeline, respectively. In real scenarios, only the EVC is stored and transmitted.

2.3 Encoded Vector Clock

Instead of using a vector of size n, [9] proposed that the vector can be encoded into a single number using n distinct prime numbers. The encoding of vector

clocks using primes was used for detecting locality-aware conjunctive predicates in large-scale systems [18]. Each process P_i is associated with a unique prime number p_i. A vector clock containing n elements, $V = [v_1, v_2, \cdots, v_n]$, can be encoded by n distinct prime numbers $p_1, p_2, \cdots, p_n$ as:

$$Enc(V) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}.$$

However, only being able to encode a vector clock into a single number is insufficient to track causal relations. The EVC technique was developed [9] to show how to implement the basic operations of a vector clock, namely, local tick, merge, and compare. The encoded vector clock t_i (initialized to 1) is operated at process P_i as shown in Fig. 1. To manipulate the EVC, each process needs to know only its own prime and not the primes of other processes.

- For a local tick, t_i is multiplied by p_i.
- For a merge of timestamps t_i and s, the $LCM(t_i, s) = \frac{t_i * s}{GCD(t_i, s)}$ is computed. Merging two EVCs requires computing the LCM, which does not require factorization.

The operations using EVC are illustrated in Fig. 2 using an example execution over three processes.

3 Simulations

For n processes in the system and f_i events at each process P_i, the maximum EVC timestamp across all processes is at least as large as $O(\prod_{i=1}^{n} p_i^{f_i})$. This is because at each event (send, receive, or internal) at P_i, the EVC gets multiplied by p_i, and in addition, at receive events, an LCM computation over two EVCs may significantly increase the EVC. From this observation, we can see that EVC timestamps grow fast. We ran simulations to test the growth rate of EVCs. The simulations were done in Rust and used the GMP library. We simulated distributed executions with a random communication pattern. As parameters, we used the number of processes, n, and the probability of a send (versus internal) event, denoted pr_s. The destination of a message from a send event was chosen at random. We timestamped events using EVCs, and measured the size of the EVC in bits. We used the first n prime numbers for the n processes.

We define the *overflow process* to be that process which is earliest to have its EVC size exceed $32n$ bits. The size $32n$ was chosen for comparison because this is the constant size used by traditional vector clocks, assuming each integer in the vector clock is represented by 4 bytes.

3.1 Simulation Results

Number of Events Until EVC Size Becomes $32n$ as a Function of n: Figure 3 shows the number of events executed in the system until the EVC size reaches $32n$ bits at the overflow process, as a function of n. We varied n from 10

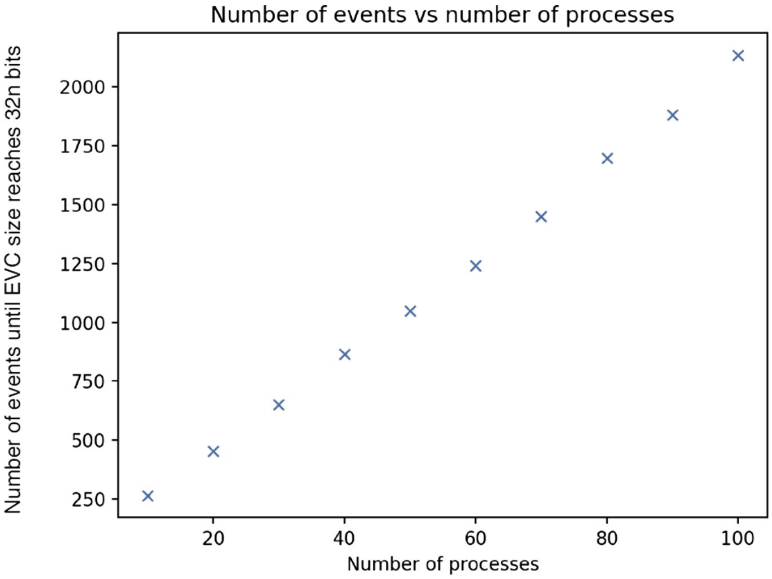

Fig. 3. Number of events needed for EVC to reach a size of $32n$ as a function of the number of processes n in the system.

to 100, and plotted the average of 10 runs for each setting, assuming that pr_s, the probability of send events (versus internal events), was 0.6. The plot turns out to be almost a straight line.

For the range of n tested (10–100), typically 21 to 25 events were executed at some process before the EVC size exceeded $32n$ at the overflow process. As this number in the interval $[21, 25]$ appears small, we conduct a worst-case strawman analysis to show that this number is reasonable. As $pr_s = 0.6$, $probability(send\,event) = probability(receive\,event) = 0.6/1.6$. We can approximate this as assuming that every third event is a receive event. Now consider, for example, $n = 60$. The simulation uses the 60 lowest prime numbers, and a significant number of them need 8 bits for representation. At each event, we multiply t_i by p_i, so the size of the EVC increases by 8 bits. In addition, at every third event (a receive event), the size of the EVC can double in the worst case due to the LCM operation. (Doubling of the size of the EVC due to LCM computation is more likely in the initial part of the execution because the LCM is likely to be computed over relative prime numbers.) So the worst-case progression of the size of the EVC in bits at a process P_i can be approximated as:

$$8, 16, 32 \text{ and } 40 \text{ (event } e_i^3), 48, 56, 112 \text{ and } 120 \text{ (event } e_i^6),$$
$$128, 136, 272 \text{ and } 280 \text{ (event } e_i^9), 288, 296, 592 \text{ and } 600 \text{ (event } e_i^{12}),$$
$$608, 616, 1232 \text{ and } 1240 \text{ (event } e_i^{15}), 1248, 1256, 2512 \text{ and } 2520 \text{ (event } e_i^{18}).$$

At the 18th event at P_i, the EVC size exceeds $60 \times 32 = 1920$ bits. As per the simulation, the overflow happens at the 1250/60th event, which is the 21st event, at the overflow process, so this worst-case analysis is reasonably accurate.

This analysis indicates that receive events cause the EVC to grow very fast due to the LCM computation.

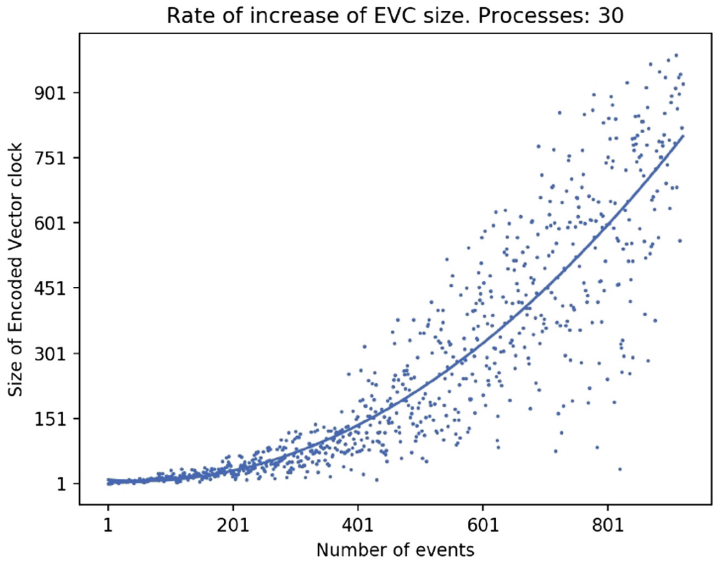

Fig. 4. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 30$.

Size of EVC as a Function of Number of Events: In our next experiment, we measured the size of the EVC in bits as a function of the number of events executed in the system. Figures 4, 5, and 6, show the scatter-plots for a system with $n = 30, 60, 100$ processes, respectively. For these executions, pr_s, the probability of send event (versus internal event) was chosen as 0.5. In these plots, the number of events on the X-axis is such that the size of the EVC in bits is always less than that of the traditional vector clocks. The Y-axis shows the size of the EVC in bits until the size equals $32n$. The maximum size $32n$ was chosen because this is the constant size used by traditional vector clocks, assuming each integer in the vector clock is represented by 4 bytes.

Consider for example, Fig. 5, which uses parameters $n = 60$ and $pr_s = 0.5$. There were about 1800 events in the systemwide execution (or an average of $1800/60 = 30$ events at a process) until the EVC size reached 1920 $(= 60 \times 32)$ bits at the overflow process.

Number of Events Until EVC Size Becomes $32n$ as a Function of Ratio of Event Types: We also varied the percentage of internal events (where the

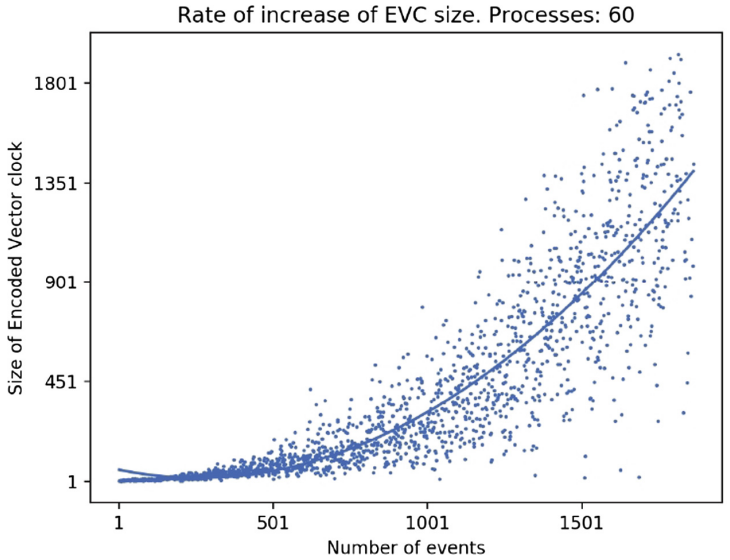

Fig. 5. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 60$.

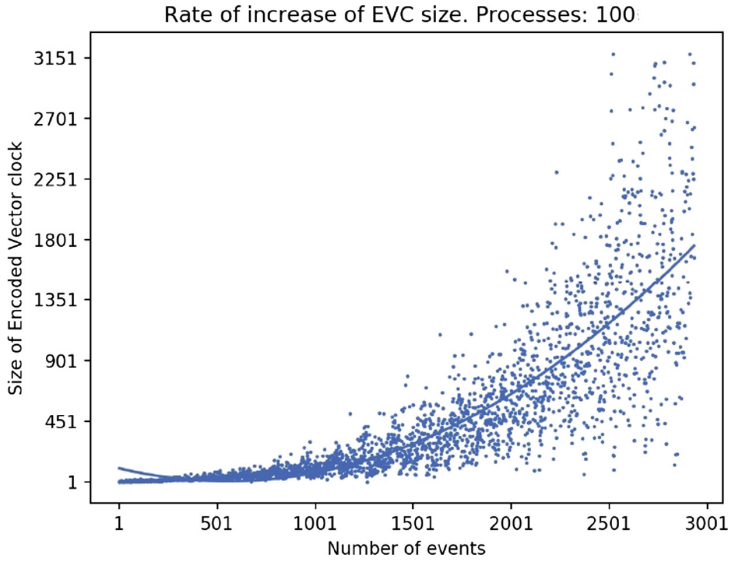

Fig. 6. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 100$.

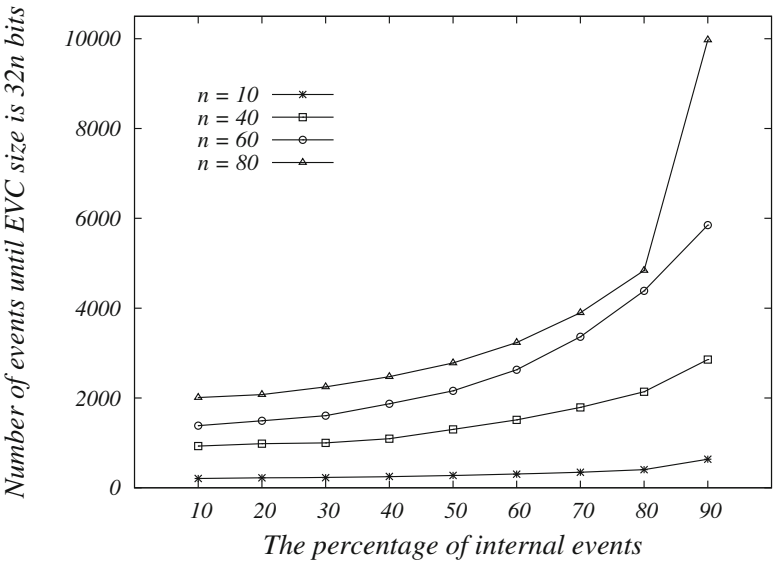

Fig. 7. Total number of events until EVC size reaches $32n$ bits, for different n and different percentages of internal events.

total number of events included send, receive, and internal events), and varied n, and observed the total number of events in the system until the EVC size reaches $32n$ bits at the overflow process. The observations are plotted in Fig. 7. For a given n, as the percentage of internal events increased, symbolizing an increasingly smaller proportion of receive events (and hence fewer LCM computations), the rate of increase of the total number of events until the EVC size reached $32n$ bits kept increasing. In particular, when $probability(internal\,event) > 0.8$, there was a more noticable rate of increase of the total number of events (until the EVC size reached $32n$ bits at the overflow process). This shows that as the proportion of send events and corresponding receive events decreases progressively, particularly below 10%, due to the fewer resulting LCM computations at receive events, the EVC grows much less rapidly, thereby resulting in a much larger number of system events until the EVC size reaches $32n$ bits. This corroborates the earlier observation that receive events cause the EVC to grow very fast due to the LCM computation.

Consider, for example, the value for $n = 60$, $probability(internal\,event) = 0.9$ which implies that $probability(receive\,event) = 0.05$. We again conduct a strawman analysis. We assume the prime numbers take up to 8 bits representation. As before, let us assume each LCM computation causes the EVC size (in bits) to double because the execution has just begun and the LCM is likely to be computed over relative prime numbers. Then, the receive event occurs every 20 events, at which time the EVC size increases by a factor of 2. So the worst-case progression of the size of the EVC in bits at a process P_i can be approximated as:

$$8, \cdots 152, 304 \text{ and } 312 \text{ (event } e_i^{20}),$$
$$320, \cdots 464, 928 \text{ and } 936 \text{ (event } e_i^{40}),$$
$$944, \cdots 1088, 2176 \text{ and } 2184 \text{ (event } e_i^{60}).$$

At the 60th event at P_i, or equivalently at around the $60 \times 60 = 3600$th event in the execution, the EVC size exceeds $60 \times 32 = 1920$ bits. As per the simulation graph (Fig. 7), the overflow happens at around the 6000th event in the execution, and this can be justified by applying a correction to the worst-case strawman analysis. Note that in the simulation, there is a delay for the message transmission. Hence, in the small initial window (before steady state) that the results in Fig. 7 depict, actually $probability(receive\, event) < 0.05$ and hence there are more than 20 non-receive events per receive event. Hence, there are more than 60 events at the overflow process P_i and hence more than 3600 events in the system until overflow occurs. This supports the simulation result of 6000 events.

4 Scalability

As seen in Sect. 3, the EVC timestamps grow fast and eventually they will exceed the size of vector clocks. However, we can use several strategies to alleviate this problem and control the maximum size of the EVC. In particular, these strategies can be used to guarantee that the EVC size is always less than the vector clock size.

4.1 Relevant Events

It suffices if the local clock does not tick at every event but only at events that are relevant to the application. Thus, the EVC does not grow so fast. This strategy is used in the context of predicate detection [18]. The local clock ticks only when the variables in the predicate alter the truth value of the predicate. As another example, the local clock ticks only at synchronization events in MPI application programs [4]; see the case study in Sect. 5.

4.2 Detection Regions

In large-scale systems, the application requiring a vector clock may be confined to only a subset of m processes, where $m < n$. An example of this is locality-aware predicate detection [18]. The subset of m processes forms a detection region. Processes within the detection region maintain a single number for the EVC. Additionally, for processes outside the detection region, we can cut down the storage cost and make the solution more practical for large-scale systems. When a process P_j outside the region first receives a message piggybacked with an EVC timestamp, it simply stores this single number. Although P_j will not tick the EVC locally since there is no corresponding component in the vector clock

for P_j, it may still receive multiple messages. Each time this happens, P_j simply executes the merge operation by calculating the LCM of two numbers. (P_j needs to store the EVC and to do the merge because it may later send messages back into the detection region, directly or transitively.)

4.3 Resetting EVC

We can adapt the clock resetting technique [22] to solve the problem when the clock overflows. This technique divides the execution of a distributed system into multiple phases. Each time the clock overflows at any process, the resetting algorithm terminates the current phase by sending control messages while ensuring there is no computation message sending from the current phase to the next phase, nor from the next phase to the current phase. The reset protocol involves a period of send inhibition of messages, and the local clock gets reset in a strongly consistent (i.e., transitless) global state [1,8]. The use of clock resetting also may require that the phase number be maintained along with the EVC, to enable (if required) the timestamp comparison of events in different phases.

It is up to the application to determine when the EVC overflows. If we say that the clock overflows when the size of the EVC equals $32n$ bits at some process, then we can guarantee that the size of EVC is always less than that of traditional vector clocks.

We can also reset the EVCs globally when there is a naturally occuring global system state in which all previous events are ordered (as per the "happened before" relation $\longrightarrow$) before all subsequent events. For example, such global synchronization occurs at a global barrier or fence instruction in MPI programs [7]; see the case study in Sect. 5. The system state immediately after a global synchronization is a transitless global state.

4.4 Using Logarithms of EVC

As the EVC technique uses exponentiation, logarithms can be used to store and transmit the EVCs. This can result in a significant reduction in the size of EVCs. We note that since logarithms involve finite-precision arithmetic, their use is subject to the introduction of errors due to the limited precision. This may potentially affect the outcome of the test of comparison of a pair of EVC timestamps in determining causality between the corresponding events.

5 Case Study

We review a case study of detecting memory consistency errors in MPI one-sided applications using EVC [4]. MPI one-sided communication, also known as MPI remote memory access, does not require sends to be matched with corresponding receive instructions [7]. Only one process takes part in the data movement (using unilateral instructions such as `MPI_Put` and `MPI_Get` rather than matching pairs of `MPI_Send` and `MPI_Receive`). It decouples data transfer between

processes from synchronization between the processes. This eliminates overhead from unneeded synchronization and allows for greater concurrency. This also eliminates message matching and buffering overheads that are incurred in traditional two-sided communication, leading to significant reduction in communication costs. These advantages of one-sided communication come at a cost – the programs are more prone to synchronization bugs, such as memory consistency errors.

In simple terms, a memory consistency error is a write to a location (through a local store instruction or through a remotely issued `MPI_Put`) that is concurrent with another write or a read (through a local load instruction or a locally or remotely issued `MPI_Get`) to the same memory location at the same process [3]. We elaborate on "is concurrent with" semi-formally. The $\xrightarrow{hb}$ "happened before" relation between events a and b is the transitive closure of the union of the program order and synchronization order. The program order at a process specifies that a previous instruction is executed before a later instruction. The synchronization order across processes orders events by the order in which synchronization instructions are executed (e.g., `MPI_Send` at a source process completes before `MPI_Receive` at the destination process). The consistency order $\xrightarrow{co}$ on events a and b guarantees that the memory effects of a are visible before b [7]. This order is necessary because synchronization instructions such as `MPI_Win_lock/unlock` order memory accesses but do not synchronize processes. For example, if a is nonblocking, and a and b both access overlapping buffers, there is no consistency order because of a potential race condition due to a being nonblocking. Now, the $\xrightarrow{cohb}$ relation on events is the transitive closure of the intersection of the $\xrightarrow{co}$ and $\xrightarrow{hb}$ relations [7]. If the $\xrightarrow{cohb}$ does not hold between a pair of events, that pair of events is concurrent under $\xrightarrow{cohb}$. Thus, two memory operations are concurrent if there are no $\xrightarrow{co}$ and $\xrightarrow{hb}$ between them. If there are two concurrent events accessing the same memory location and at least one of them is an update operation (whether local or remote), then there is a memory consistency error in an MPI one-sided program execution. Note that a memory consistency error may be of two types: either within an epoch at the same process, or across processes.

Although MPI one-sided communication calls may cause memory consistency errors with other such calls or load/store operations, not every pair of operations will cause such errors. This is because MPI applications use synchronization calls (such as `MPI_Barrier` and `MPI_Win_fence`) to enforce $\xrightarrow{co}$ and/or $\xrightarrow{hb}$ between two operations. Only when two operations fall within a concurrent program region may memory consistency errors arise. A *concurrent program region* is defined as a group of program regions across multiple (all) processes, that can be executed concurrently without $\xrightarrow{co}$ and $\xrightarrow{hb}$ ordering relations, i.e., program regions that are not ordered by $\xrightarrow{cohb}$ [3]. Each *program region* is formed of one or multiple epochs, where an *epoch* is formed by a pair of one-sided synchronization calls.

MC-Checker [3] is a tool for identifying memory consistency errors. Using trace files, it generates a dynamic data access DAG whose nodes are the events and edges represent the "happened before" relation. The DAG represents a set of concurrent regions. A concurrent region begins and ends with a global synchronization operation (such as `MPI_Barrier` and `MPI_Win_fence`). In the general case, the set of concurrent regions forms a partial order. However, the set of concurrent regions is totally ordered, assuming a single MPI communicator. Each concurrent region is modeled as a graph: the set of nodes are the events in MPI one-sided programs and the edges are the $\xrightarrow{cohb}$ relation. Each concurrent region is (independently) analyzed to detect memory consistency errors – such an error exists between each pair of conflicting operations that are not ordered by $\xrightarrow{cohb}$. MC-Checker detects conflicting operations within each epoch of a program region, and across processes within the concurrent region.

Typically, two-sided communication is used along with one-sided communication in high-performance computing applications. In order to detect memory consistency errors, transitive dependencies between processes, such as those induced by send and receive operations by several different processes, need to be captured. MC-Checker [3] suffers the drawback that it does not take into account such transitive dependencies, because capturing such dependencies would require building a complete DAG of dependencies between events for analysis, which would require maintaining vector clocks. However, vector clocks do not scale and they impose high overheads; as a result MC-Checker did not use vector clocks and this led to the introduction of false positives in reporting memory consistency errors.

The MC-CChecker tool [4] overcame this drawback by using the EVC, thereby eliminating the false positives reported by MC-Checker while still maintaining low overheads. As the $\xrightarrow{cohb}$ relation is specified only on synchronization events within and across processes (these are the relevant events), the EVC scheme also needs to timestamp only such events. MC-CChecker adapted the EVC rules of Fig. 1 [9] to MPI one-sided communication system as follows [4].

R1. For two consecutive synchronization events, if $e_i^x \xrightarrow{cohb} e_i^{x+1}$, then $t_i^{x+1} = t_i^x * p_i$.

R2. If e_i^x is `fence` (or `barrier`) and e_j^y is the corresponding `fence` (or `barrier`), then a message m from e_i^x to e_j^y is timestamped $tm = t_i^x$. On receipt at P_j, $t_j^y = LCM(tm, t_j^y)$.

R3/R4/R5. If e_i^x is `post`/`complete`/`send` and e_j^y is the corresponding `start`/`wait`/`receive`, then a message m from e_i^x to e_j^y is timestamped $tm = t_i^x$; and then a local tick is executed at P_i. On receipt at P_j, $t_j^y = LCM(tm, t_j^y)$.

For simplicity, it is assumed that `post` $\xrightarrow{cohb}$ `start` and `complete` $\xrightarrow{cohb}$ `wait`. Only synchronization operations are timestamped as the goal is to represent an area (termed as a *separate region*) formed between two consecutive synchronization operations, including the former but excluding the latter; the timestamps of all events within the separate region equal the timestamp of the representing (former) synchronization event's timestamp.

Along the lines of the test in [9], $e_i^x \xrightarrow{cohb} e_j^y$ if and only if t_i^x divides t_j^y. The two events are concurrent under $\xrightarrow{cohb}$ if and only if the EVC timestamp of neither event divides that of the other. MC-CChecker considers concurrent regions like MC-Checker, but using EVC timestamped information built after analyzing the trace files. MC-CChecker loads concurrent regions one by one from trace files. Once MC-CChecker loads one concurrent region, it detects memory consistency errors within each epoch similar to MC-Checker. However, for errors across processes, it examines the concurrency of each pair of separate regions for each concurrent region. If two separate regions are executed concurrently, MC-CChecker checks the accessed memory of each pair of operations belonging to the two separate regions to flag memory consistency errors (if the two operations are concurrent under $\xrightarrow{cohb}$, conflict, and access the same location).

Experiments run on HPC platforms using three different MPI applications showed that MC-CChecker used low processing time and memory usage, when checked for up to 128 processes. The scalability study compared MC-CChecker using EVC and using traditional vector clocks, for systems ranging from 512 up to 8192 processes. The study showed that with EVC, execution time and memory usage are linear (with respect to n), whereas with traditional vector clocks, both execution time and memory usage were significantly higher and increased in much larger proportion.

In this case study, the relevant events were the synchronization events; only these were timestamped by MC-CChecker using EVCs. Further, each concurrent region contained a program region from a different process. All the concurrent regions were totally ordered, assuming a single MPI communicator. (Without this assumption, the concurrent regions form a partial order.) The boundary between two adjacent concurrent regions was implemented by global synchronization calls such as `MPI_Barrier` and `MPI_Win_fence`. The start of each concurrent region corresponded to a global synchronization where there was no concurrency between events in the previous concurrent region and in the following one. Each concurrent region was a unit of computation [1,8], and the boundary between two adjacent/consecutive concurrent regions corresponded to a global transitless state. MC-CChecker safely reset the EVC of each process to 1 at the start of each concurrent region. Using the combination of these two techniques, viz., tick at relevant event, and reset at the start of each concurrent region, the size of the EVCs at the processes remained small and grew linearly (with n), as the MC-CChecker scalability study showed.

6 Conclusions

Vector clocks are important in distributed and parallel systems, but are not very scalable because they have a space complexity of $O(n)$. The encoding of the vector clock using prime numbers, to use a single number to represent vector time, has the potential to save on the space overheads of vector clocks. A drawback of EVCs is that they grow fast and soon overflow, i.e., exceeding the space used by

traditional vector clocks soon occurs. To understand this growth phenomenon, we showed the results of simulations to examine how fast the EVC grows. The simulations confirm that the EVC grows relatively fast, and the growth rate also depends on the ratio of internal events to communication events. In particular, receive events which use an LCM computation cause the size of the EVC to grow more significantly.

Scalability approaches for the EVC to deal with the overflow problem can be used. These include ticking the clock only at application-relevant events and only at processes where such events occur, and resetting the EVC throughout the system at a transitless global state when it overflows at some process or at a global synchronization. A judicious use of these scalability approaches can control the size of the EVC and can be used to guarantee that the size of the EVC never exceeds the size of the traditional vector clock. We considered a case study of using EVC for detecting memory consistency errors in MPI applications that use one-sided communication. Using the combination of two scalability approaches, viz., ticking at relevant event, and resetting at the start of each concurrent region, the size of the EVCs at the processes remained small, grew linearly, and was significantly much less than that using traditional vector clocks.

The EVC timestamps in the case study were assigned after analyzing the program traces. It would be interesting to determine whether they can be assigned in an on-line manner efficiently. Another future direction is to examine whether the EVCs can be used instead of traditional vector clocks in tools for dynamic race detection in multithreaded programs, such as DJIT+ [16] and FastTrack [6].

Acknowledgements. We thank Rahul Sathe for his help with the simulations.

References

1. Ahuja, M., Kshemkalyani, A.D., Carlson, T.: A basic unit of computation in distributed systems. In: 10th International Conference on Distributed Computing Systems (ICDCS 1990), 28 May–1 June 1990, Paris, France, pp. 12–19 (1990)
2. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Inf. Process. Lett. **39**(1), 11–16 (1991)
3. Chen, Z., Dinan, J., Tang, Z., Balaji, P., Zhong, H., Wei, J., Huang, T., Qin, F.: MC-Checker: detecting memory consistency errors in MPI one-sided applications. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, 16–21 November 2014, pp. 499–510 (2014)
4. Diep, T.D., Fürlinger, K., Thoai, N.: 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, EuroMPI 2018, pp. 9:1–9:11 (2018)
5. Fidge, C.J.: Logical time in distributed computing systems. IEEE Comput. **24**(8), 28–33 (1991)
6. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, 15–21 June 2009, pp. 121–133 (2009)

7. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., Underwood, K.D.: Remote memory access programming in MPI-3. TOPC **2**(2), 9:1–9:26 (2015)
8. Kshemkalyani, A.D.: A framework for viewing atomic events in distributed computations. Theor. Comput. Sci. **196**(1–2), 45–70 (1998)
9. Kshemkalyani, A.D., Khokhar, A.A., Shen, M.: Encoded vector clock: using primes to characterize causality in distributed systems. In: Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, 4–7 January 2018, pp. 12:1–12:8 (2018)
10. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, Cambridge (2011)
11. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) OPODIS 2014. LNCS, vol. 8878, pp. 17–32. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14472-6_2
12. Kulkarni, S.S., Vaidya, N.H.: Effectiveness of delaying timestamp computation. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, 25–27 July 2017, pp. 263–272 (2017)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
14. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Parallel and Distributed Algorithms Conference, pp. 215–226 (1988)
15. Meldal, S., Sankar, S., Vera, J.: Exploiting locality in maintaining potential causality. In: Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1991, pp. 231–239 (1991)
16. Pozniansky, E., Schuster, A.: MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. Concurr. Comput. Pract. Exp. **19**(3), 327–340 (2007)
17. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: in search of the holy grail. Distrib. Comput. **7**(3), 149–174 (1994)
18. Shen, M., Kshemkalyani, A.D., Khokhar, A.A.: Detecting unstable conjunctive locality-aware predicates in large-scale systems. In: IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, 27–30 June 2013, pp. 127–134 (2013)
19. Singhal, M., Kshemkalyani, A.D.: An efficient implementation of vector clocks. Inf. Process. Lett. **43**(1), 47–52 (1992)
20. Torres-Rojas, F.J., Ahamad, M.: Plausible clocks: constant size logical clocks for distributed systems. Distrib. Comput. **12**(4), 179–195 (1999)
21. Ward, P.A.S., Taylor, D.J.: A hierarchical cluster algorithm for dynamic, centralized timestamps. In: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, 16–19 April 2001, pp. 585–593 (2001)
22. Yen, L., Huang, T.: Resetting vector clocks in distributed systems. J. Parallel Distrib. Comput. **43**(1), 15–20 (1997)