2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

An Optimal Vector Clock Algorithm for Multithreaded Systems

Xiong Zheng
Electrical and Computer Engineering
The University of Texas
Austin, USA
zhengxiongtym@utexas.edu

Vijay K. Garg
Electrical and Computer Engineering
The University of Texas
Austin, USA
garg@ece.utexas.edu

***Abstract*—Tracking causality (or happened-before relation) between events is useful for many applications such as debugging and recovery from failures. Consider a concurrent system with n threads and m objects. For such systems, either a vector clock of size n is used with one component per thread or a vector clock of size m is used with one component per object. A natural question is whether one can use a vector clock of size strictly less than the minimum of m and n to timestamp events. In this paper, we give an algorithm that uses a hybrid of thread and object components, and is guaranteed to return the minimum number of components necessary for vector clocks. We first consider the case when the interaction between objects and threads is statically known. This interaction is modeled by a thread-object bipartite graph. Our algorithm is based on finding the maximum bipartite matching of such a graph and then applying König-Egerváry Theorem to compute the minimum vertex cover to determine the optimal number of components necessary for the vector clock. We also propose two mechanisms to compute such a vector clock when the computation is revealed in an online fashion. Our evaluation on different types of graphs indicates that the offline algorithm generates a size vector clock which is significantly less than the minimum of m and n. These mechanisms are more effective when the underlying bipartite graph is not dense.**

***Index Terms*—optimal vector clocks, bipartite matching**

I. Introduction

A fundamental problem in parallel and distributed systems is to determine the order relationship between events of a distributed computation as defined by Lamport's *happened-before relation* [9]. The problem arises in many areas including debugging and visualization of parallel and distributed programs.

Vector clocks, which were introduced independently by Fidge [3]–[5] and Mattern [11], and their variants [10] are widely used to capture the causality between events in parallel and distributed computations. To capture the causality, each event is timestamped with the current value of the local vector clock at the time the event is generated. The order relationship between two events can then be determined by comparing their timestamps. A vector clock contains one component for every process in the system. This results in message and space overhead of n integers in a distributed system consisting of n processes. In shared-memory systems, there are two kinds of vector clocks. Consider a concurrent system with n threads and m objects. For such systems, either vector clocks of size n is used with one component per thread or a vector clock of size m is used with one component per object. A natural question is whether one can use a vector of size strictly less than the minimum of m and n to timestamp events. We show that this is indeed possible.

Consider the example in Fig. 1. In this example, t_1, t_2, t_3, t_4 are threads, and o_1, o_2, o_3, o_4 are objects used by these threads. Each circle represents an operation. To order these operations, traditionally, either all threads, or all objects are used as components of a vector clock to infer the happened-before order. However, notice that all the operations are related to thread t_2, object o_2 or object o_3. Therefore, we can use a vector clock composed of t_2, o_2, and o_3 to timestamp all events. This mixed vector-clock is size of 3 which is smaller than the number of threads and the number of objects. This example shows that a vector of size strictly less than the minimum of m and n can be used to timestamp a computation.

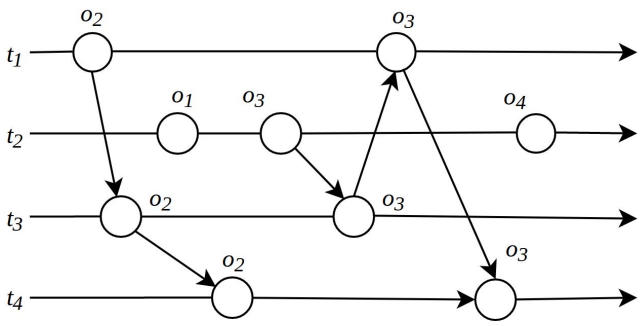

Fig. 1. A computation of threads operating on shared objects

Let us represent a computation of threads operating on objects as a bipartite graph with the set of threads as the left nodes and the set of objects as the right nodes. An edge in this bipartite graph corresponds to an event in a computation relating the thread to the object on which the operation is performed. A vertex cover of a graph is a set of vertices such that each edge in the graph is incident to at least one vertex of the set. Clearly, the size of the minimum vertex cover of a

This work was partially supported by NSF CSR-1563544,CNS-1812349

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00215

bipartite graph is no greater than the minimum of the number of left nodes and the number of right nodes. Therefore, by converting a computation to a thread-object bipartite graph (introduced in section III), we can use the minimum vertex cover of this bipartite graph to determine the components of the vector clock for this computation. On this basis, we give an offline algorithm that uses a hybrid of thread and object components which is smaller in size than the traditional vector clocks to timestamp a computation. This algorithm is guaranteed to return the minimum number of components necessary for a vector clock. It is based on finding a maximum bipartite matching in the thread object bipartite graph and then applying König-Egerváry Theorem [1] to determine the optimal number of components necessary for the vector clock.

We also consider the case when the interaction between threads and objects is not known *a priori*. We propose two mechanisms to address this problem as well as compare performance with the traditional solution.

In summary, this paper makes the following contributions:

- We introduce the notion of a mixed vector-clock that satisfies the vector clock condition with fewer entries than the thread-based or the object-based clock.
- We give an optimal offline algorithm to determine which threads and objects should be used for a mixed vector-clock.
- We give two mechanisms to compute the mixed vector-clock when the events of a computation arrive in an online fashion.

II. System Model and Notation

In this section, we present our model of a concurrent system. The system consists of n sequential processes (or threads) denoted by $T = \{t_1, t_2, \ldots, t_n\}$ performing operations on m objects denoted by $O = \{o_1, o_2, \ldots, o_m\}$. From now on, we use threads or processes interchangeably. A *computation* in the happened-before model is defined as a tuple $(E, \rightarrow)$ where E is the set of events and $\rightarrow$ is a partial order on events in E. Each thread executes a sequence of events. Each event is performed on a single object. We assume that all operations on any single object are performed sequentially (for example, by using locks). For an event $e \in E$, $e.t$ denotes the thread on which e occurred and $e.o$ denotes the object on which e occurred.

Then Lamport's happened-before relation ($\rightarrow$) on E is the smallest transitive relation such that:

1. If $e.t = f.t$ and e immediately precedes f in the sequence of events in thread $e.t$, then $e \rightarrow f$.

2. If $e.o = f.o$ and e immediately precedes f in the sequence of events on the object $e.o$, then $e \rightarrow f$

Two events e and f are said to be *comparable* if $e \rightarrow f$ or $f \rightarrow e$. If e and f are not comparable, they are said to be *concurrent* and this relationship is denoted by $e \parallel f$.

We define a thread-object graph as an undirected bipartite graph $G = (T, O, R)$ where R is the set of edges between the set of threads T and the set of objects O. R is defined as

$$R = \{(t, o) \mid \text{the object } o \text{ is accessible to thread } t\}$$

In most applications, G is not dense, i.e., a thread typically has references to only a small subset of objects.

The set of events E with the order imposed by Lamport's happened before relation defines a partially ordered set or *poset*. A subset of elements $C \subseteq E$ is said to form a *chain* iff $\forall e, f \in C : e \rightarrow f$ or $f \rightarrow e$. By our definition of threads all operations done by a single thread form a chain. Similarly, all operations done on a single object also form a chain.

Thread-based vector clocks maintain a vector v of size $|T|$ with each thread and object. Whenever, a thread t executes an operation e on object o it gets the timestamp $e.v$ as

$$e.v = max(t.v, o.v); e.v[e.t] + +;$$

Both t and o update their vector to $e.v$.

Object-based vector clocks maintain a vector v of size $|O|$ with each thread and object. Whenever, a thread t executes an operation e on object o it gets the timestamp $e.v$ as

$$e.v = max(t.v, o.v); e.v[e.o] + +;$$

Both t and o update their vector to $e.v$.

We design a vector clock called *mixed vector-clock* that uses a combination of threads and objects for its components. Clearly, thread-based and object-based vector clocks are special cases of our scheme. Moreover, the total number of components in a mixed vector clock is always less than or equal to the minimum of the thread and the object based vector clocks.

III. An Offline Algorithm

In this section, we first formally define the thread-object bipartite graph for a computation. Next, we give an offline algorithm to compute the optimal mixed vector-clock by computing the maximum matching in the thread-object bipartite graph and obtaining a minimum vertex cover. Then, we show that the mixed vector-clock given by this offline algorithm is optimal in terms of size.

A. The Thread-object Bipartite Graph

A thread-object computation is composed of events which are in the form of some specific thread doing some operations on a specific object. Notice that such a computation only involves two parties: threads and objects. An operation relates an thread and an object. Therefore, such a computation could be modeled as a bipartite graph if we only focus on the relation between the two parties, i.e., for thread t and object o, we only care about whether t has any operation on o or not and ignore exactly how many operations that t has on o. If a thread has at least one operation on an object, then there is an edge between them in the bipartite graph. We call such a bipartite graph a *thread-object bipartite graph*. The computation shown

in Fig. 1 can be converted to the thread-object bipartite graph given in Fig. 2. The filled vertices represent the minimum vertex cover of this bipartite graph or the components of our *mixed vector-clock*.

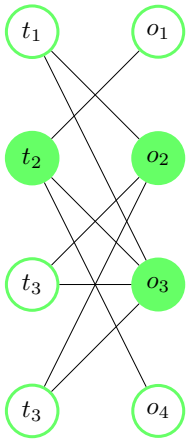

Fig. 2. Thread-Object Bipartite Graph of A Computation

B. The Offline Algorithm

Let $G = \{T, O, R\}$ be the thread-object bipartite graph of a given computation. Given a computation, assuming that G is given or constructed by the trace generator, we show how to obtain our mixed vector-clock by computing a minimum vertex cover of this bipartite graph. In order to compute such a vertex cover, we use König-Egerváry Theorem.

Theorem 1 (König-Egerváry Theorem): In any bipartite graph, the size of a maximum matching equals the size of a minimum vertex cover.

Based on König-Egerváry's theorem, we first compute a maximum matching of the thread-object bipartite graph. One simple and efficient algorithm is the bipartite matching algorithm given by Hopcroft and Karp [8], which achieves the time complexity of $O(|V|^{5/2})$ in a bipartite graph with the vertex set V.

The basics of this matching algorithm are as follows. At each iteration, the algorithm searches for shortest augmenting paths denoted as $\{R_1, R_2, ..., R_t\}$ relative to existing matching M and augments the current matching. The new matching M' is obtained by $M \oplus R_1 \oplus R_2 \oplus \cdots \oplus R_t$, where $\oplus$ represents symmetric difference. When there is no augmenting path in the bipartite graph, the maximum matching is found. The details of this algorithm can be found in [8].

Next, given the maximum matching, we directly apply König-Egerváry Theorem to convert the maximum matching to the minimum vertex cover to get the mixed vector-clock. The pseudocode to compute the mixed vector-clock is given in **Algorithm 1**. In this algorithm, line 1 computes a maximum matching M^* of G. Given M^*, at line 2 we compute the set of unmatched threads, denoted as S. Line 3-9 is the procedure to convert the maximum matching M^* to a minimum vertex cover. Z is the set of nodes in the graph which are connected by M^*-alternating paths to S. The minimum vertex cover C^* can be computed as $(T - Z) \cup (O \cap Z)$. This procedure can be found in the proof for König-Egerváry Theorem in [1] (Theorem 5.3).

Algorithm 1 Minimum *Mixed vector-clock*

1: M^* := a maximum matching of G
2: Compute the set of unmatched threads, denoted as S
3: Let $Z := S$
4: **for** $s \in S$ **do**
5: Start from s, BFS search via alternating paths
6: Let B_s denote the set of vertices traversed by BFS
7: $Z := Z \cup B_s$
8: **end for**
9: Return $C^* = (T - Z) \cup (O \cap Z)$

Given the minimum vertex cover of the thread-object bipartite graph, the mixed vector-clock is simply constructed by assigning each thread or object in the minimum vertex cover as a component in the vector clock.

C. Timestamping Events Using mixed vector-clock

The offline algorithm gives a mixed vector-clock. To timestamp events in a thread object computation, we let each thread and each object keep a mixed vector-clock v. Let $comp(v)$ denote the components of v, which is the set composed of the threads and objects in C^*. We initialize each component of the mixed vector-clock on each thread and object to be 0. Now, let us look at how each thread modifies its mixed vector-clock to track causality of operations. For thread t, while performing operation e on object o, thread t needs to check whether itself or object o is in the mixed vector-clock and increases the component correspondingly. Here is how to update the timestamp of event e:

$$\boldsymbol{if\ o \in comp(v)\ then:}\ e.v = max(t.v, o.v); e.v[e.o] + +;$$

$$\boldsymbol{if\ t \in comp(v)\ then:}\ e.v = max(t.v, o.v); e.v[e.t] + +;$$

Both thread t and object o update their mixed vector-clock to be $e.v$.

Fig. 3 shows the timestamp for each event in the computation given in Fig. 1. The components in the mixed vector-clock correspond to thread t_2, object o_2 and object o_3, respectively. Initially, the *mixed vector-clock* for all threads and objects are $[0,\ 0,\ 0]$.

D. Proof of Correctness and Optimality

Let e be an operation in the computation, $e.t$ be the associated thread of e, $e.o$ be the associated object of e, $e.c$ be the component which is in the mixed vector-clock. We have $e.c \in \{e.t, e.o\}$. Let e and f be any two operations in the computation. We show the correctness and optimality of our vector clock algorithm.

Lemma 1: Let $e \neq f$. Then,

$$e \nrightarrow f \Rightarrow f.v[e.c] < e.v[e.c]$$

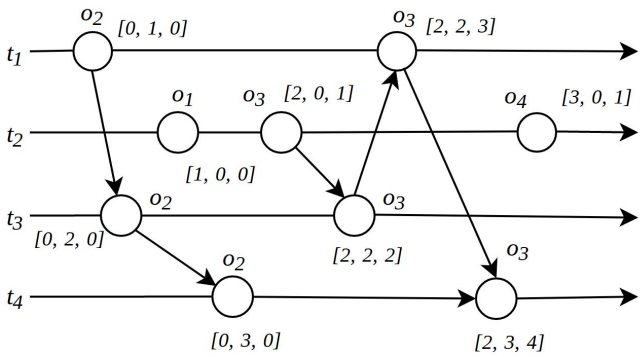

Fig. 3. Timestamping Events Using mixed vector-clock. The components correspond to thread t_2, object o_2 and object o_3.

Proof: Case 1, $e.t = f.t$. Before executing operation e, the vector clock kept in thread $e.t$ must be no less than $f.v$, since thread $e.t$ updates its vector to be the same as $f.v$ after operation f. At operation e, since we take $e.v$ to be the max of $e.t.v$ and $e.o.v$, and then increase the component $e.c$ by one, thus $f.v[e.c] < e.v[e.c]$.

Case 2, $e.o = f.o$. In this case, when executing operation e, we first update $e.v$ to be the max of $e.t.v$ and $e.o.v$, and $e.o.v$ must be at least $t.o.v$, so $e.v$ must be at least $f.v$ after taking the max. Since, $e.v$ increases its component $e.c$ by at least one, it follows that $f.v[e.c] < e.v[e.c]$.

Case 3, $e.t \neq f.t$ and $e.o \neq f.o$. We prove by induction on the length of any chain from an initial state to f. Let C denote any chain from an initial state to f. If f is the initial state, then clearly we have $f.v[e.c] < e.v[e.c]$. Let f' denote the operation that immediately precedes f on the chain C. So, we have $f' \rightarrow f$. We assume $f'.v[e.c] < e.v[e.c]$ by induction hypothesis. Since f and e involve different thread and object, the component $e.c$ would not be increased at operation f. Thus, we have $f.v[e.c] < e.v[e.c]$. ■

Theorem 2: (*Correctness:*) The *mixed vector-clock* is a valid vector clock.

Proof: In order to show the *mixed vector-clock* is a valid vector clock, we need to show that it satisfies the constraint: $\forall e, f : e \rightarrow f \Leftrightarrow e.v < f.v$.

($\Rightarrow$) $e \rightarrow f \Rightarrow e.v < f.v$ Since $e \rightarrow f$, there exists at least one chain from e to f. Let C be such a chain. Consider any two events $e_1, e_2 \in C$ such that e_1 immediately precedes e_2. We have either $e_1.t = e_2.t$ or $e_1.o = e_2.o$.

Case 1: $e_1.t = e_2.t$ As we can see from the vector clock algorithm, within the same thread, each event will first set its new vector clock to be the max of previous vector clock and vector on object. Thus $e_1.v < e_2.v$.

Case 2: $e_1.o = e_2.o$. e_1 precedes e_2 in the sequence of events on object $e.o$. Therefore, $e_2.v[e_1.o] > e_1.v[e_2.o]$ and for $\forall j \neq e_1.o$: $e_1.v[j] \leq e_2.v[j]$. Thus, $e_1.v < e_2.v$.

Thus, we have that $e.v < f.v$.

($\Leftarrow$) $e \nrightarrow f \Rightarrow e.v \not< f.v$

From Lemma 1, we have $e \nrightarrow f \Rightarrow f.v[e.c] < e.v[e.c]$. Thus, $e.v \not< f.v$. ■

Theorem 3: (*Optimality:*) The *mixed vector-clock* given by algorithm 1 is the minimum vector clock for any thread-object computation.

Proof: The offline algorithm makes use of the components of a minimum vertex cover v of the thread-object bipartite graph as the *mixed vector-clock*, which is also a valid vector clock of the computation. We need to show that v is minimum in size. Timestamping all events in a computation requires the vector clock being able to order each event. Suppose there exists a smaller vector clock v' which timestamps all events. The fact that events in computation corresponds to edges in its bipartite graph indicates that v' is also a vertex cover of the bipartite graph, which contradicts $|v| \leq |v'|$. So, the *mixed vector-clock* obtained by the offline algorithm is optimal in size. ■

IV. MIXED VECTOR-CLOCK FOR ONLINE COMPUTATION

In this section, we consider the case when the computation is not given in advance; instead, each event of the computation is revealed in an online fashion. We assume that only one event is revealed at any time. Thus, in the online setting, we need to maintain a valid dynamic vector clock when events of a computation arrive one at a time. The thread-object bipartite graph for the computation may change when events arrive in the online fashion. When an event $e = (t, o)$ is revealed, there could be two cases. The first case is when the thread t has already performed some operation on object o, i.e., there is already an edge between t and o in the current thread-object bipartite graph. In this case, the thread-object bipartite graph does not change. For the second case, thread t has never performed any operation on object o, i.e., there is no edge between t and o in the current thread-object bipartite graph. Thus, an edge between t and o should be added into the thread-object bipartite graph. Note that in the online setting the idea of using minimum vertex cover to be the components of mixed vector-clock cannot be applied, since the minimum vertex cover of the thread-object graph changes and the existing components in a mixed vector-clock should not be modified as a new event arrives. That is, we can only add new components to the vector clock.

The naive solution is to always choose the thread or always choose the object as components of the vector clock as a new event occurs. This mechanism would result in a vector clock with size equal to the number of threads or objects for all computations. Another intuitive mechanism is to randomly choose the object or the thread to add into the vector clock with equal probability. Notice that the hardness of timestamping an online computation stems from the unpredictability of future events. We can only estimate the future using information we already have. Therefore, we propose another mechanism which makes use of the partial computation occurred so far to predict the future events. Specifically, when a new event occurs, if the associated object is more *popular* than the associated thread,

then we choose the object, otherwise, we choose the thread. We propose the definition of popularity as follows.

Definition 1: The *popularity* of a vertex v in a bipartite graph $G = \{U, V, E\}$ is $pop(v) = \frac{deg_v}{|E|}$, where deg_v is the degree of vertex v and $|E|$ is the total number of edges in the graph. We say one node is more popular than another node if it has higher *popularity*.

We assume that only one event can occur at a single time. An event comes in with its associated thread and object. We are not supposed to modify components existing in the mixed vector-clock. The three mechanism are formally listed as:

1. *Naive:* Always choose threads or objects.

2. *Random:* Randomly choose the associated object or thread of the new event with equal probability.

3. *Popularity:* Based on popularity of threads and objects. When a new event comes in. If one of the associated thread or object is already in the vector clock, then the vector clock remains same. Otherwise, compute the popularity of the associated thread and object, add the one with higher popularity into the vector clock.

V. Evaluation

In this section, to evaluate the performance of our offline algorithm and compare the performance of the three mechanisms proposed for online setting, we consider the following two scenarios:

Uniform: Evaluation on a uniformly and randomly generated thread-object bipartite graph, i.e., each thread and object has same popularity.

Nonuniform: Evaluation on a thread-object bipartite graph in which a small fraction of objects and threads are much more popular than other threads and objects.

The bipartite graph in *Uniform* scenario is generated by adding an edge between each thread and each object with the same fixed probability. For *Nonuniform* scenario, the bipartite graph is generated by adding an edge between popular threads and objects with a higher probability and non-popular threads and objects with a smaller probability.

In our first evaluation, we consider how the graph density affects the vector clock size of the three mechanisms. We set the number of threads and objects in the computation to be 50, respectively, i.e., each side of the thread-object bipartite graph has 50 nodes. For each scenario, we compute the final vector clock size by applying the above three mechanisms to the above two different scenarios, as the density of thread-object bipartite graph increases. The results are shown in Fig. 4. The first important conclusion is that when the density of the bipartite graph is small, *Random* and *Popularity* method produce significantly smaller vector clock than the *Naive* method. However, when the density of graph exceeds a certain threshold, their performance becomes worse than *Naive*. In addition, we found that *Random* and *Popularity* mechanism can obtain much better solution in the *Nonuniform* case than the *uniform* case. Thus, we conjecture that these two methods are better suited in the computation in which some objects or threads are more popular than other objects and threads. Comparing performance of *Random* and *Popularity*, we found *Popularity* is a slightly better than *Random*. This can be explained by the fact that by choosing popular nodes as vector clock component, we can cover more edges. Thus, the vector clock size would be smaller.

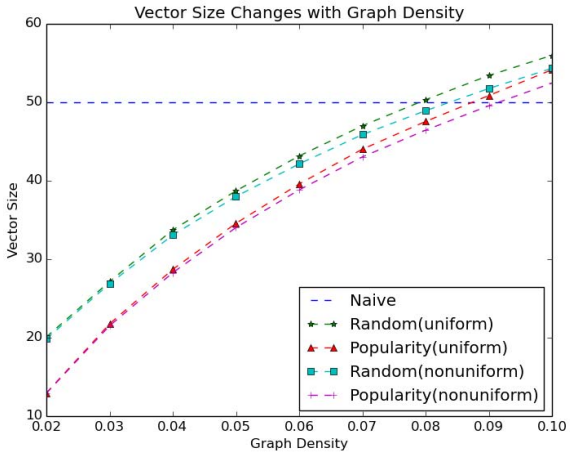

Fig. 4. Vector Size Varies as Graph Density Increases.

In our second evaluation, we fix the graph density to be 0.05 and evaluate the performance of the three mechanisms as we increase the number of nodes in the bipartite graph. From Fig. 5, we can see that as the number of nodes in the graph increases, the vector size increases. When the number of nodes is below a certain threshold, 70 here, *Random* and *Popularity* generates smaller vector clock size than *Naive*. Once the number of nodes exceeds that threshold, *Naive* is better, which means by simply choosing either all threads or objects as vector clock components gives a smaller vector clock. Therefore, we conclude that these two techniques are more effective in simple computations, i.e., computations which involve relatively a small number of threads and objects.

In our third evaluation, we want to know how far the online case drifts from the static case. We choose the *Popularity* mechanism for the online case. We also consider the *Naive* mechanism, which can be applied to the online case and the static case and generates the same vector clock. For the static case, we use our offline algorithm proposed in Section III. For this experiment, we first apply the *Popularity* mechanism as we reveal the edge of the graph one by one. Then, after we have the whole graph, we apply the offline algorithm. Fig. 6 shows the results we get when we set the number of nodes to be 50 and increase the graph density. Fig. 7 shows the results we get when we fix the density to be 0.05 and increase the number of nodes in the graph. First, we can notice that our offline algorithm generates a vector clock with a significantly smaller size than the *Naive* solution. For example, *Naive* has the vector clock of size 50 and the offline algorithm reduces that to be around 35, when there are 50 threads and the graph density is 0.05. Besides, in the online setting, although the *Popularity* mechanism cannot achieve as small vector size as the optimal

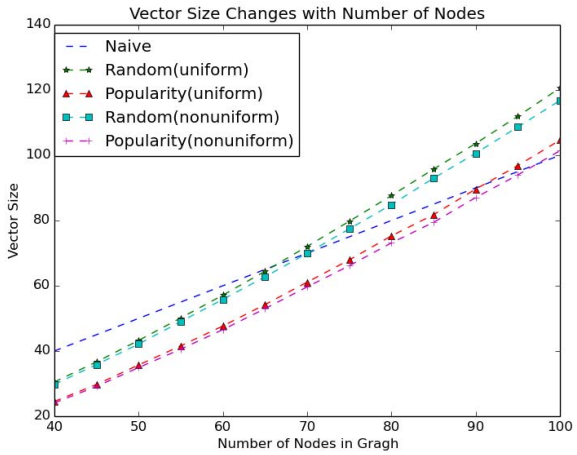

Fig. 5. Vector Size Varies as Number of Nodes Increases

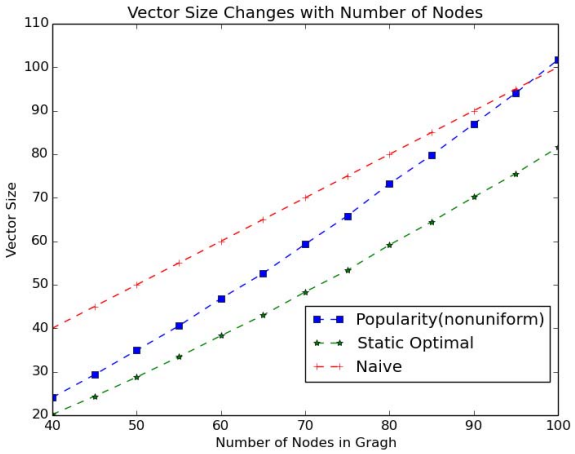

Fig. 7. Vector Size Varies as Number of Nodes Increases

solution, the gap is small. For example, *Popularity* generates a vector of size around 56 while the optimal is around 48, when there are 70 threads and the graph density is 0.05. Also, as the graph density or the number of nodes in the graph increases, the gap increases which indicates that the *Popularity* mechanism is not suitable for a relatively dense graph.

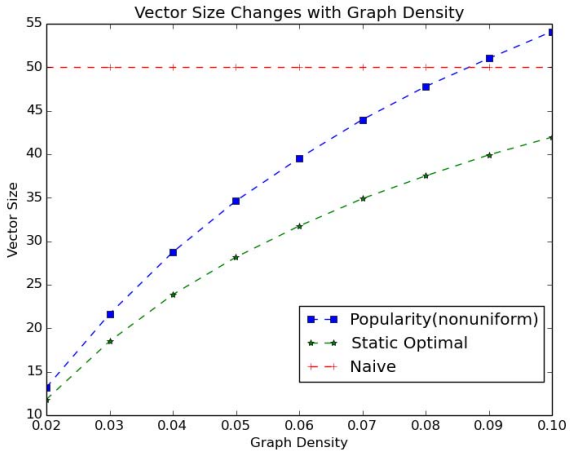

Fig. 6. Vector Size Varies as Graph Density Increases

VI. Related Work

Several techniques have been proposed to reduce the overhead imposed by Fidge/Mattern's vector clocks [3]–[5], [11]. Singhal and Kshemkalyani [13] present a technique to reduce the amount of data piggybacked on each message. The main idea is to only send those entries of the vector along with a message that have changed since a message was last sent to that process. Hélary *et al.* [7] further improve upon Singhal and Kshemkalyani technique and describe a suite of algorithms that provide different trade offs between space overhead and communication overhead. The ideas described in the two papers are orthogonal to the ideas presented in this paper and, therefore, can also benefit our timestamping algorithm by reducing its overhead.

Torres-Rojas and Ahamad [14] introduce another variant of vector clocks called *plausible clocks*. Unlike traditional vector clocks, plausible clocks are scalable because they can be implemented using fixed-length vectors independent of the number of processes. However, plausible clocks do not characterize causality completely because two events may be ordered even if they are concurrent. As a result, plausible clocks are useful only when imposing ordering on some pairs of concurrent events has no effect on the correctness of the application.

Several centralized algorithms for timestamping events have also been proposed [16]–[18]. They are mainly used for visualizing a distributed computation. An important objective of these algorithms is to reduce the amount of space required to store timestamps for all events in a computation while maintaining the time required for comparing two events (to determine their relationship) at an acceptable level. Ward presents two centralized algorithms to create vector timestamps whose size can be as small as the dimension of the partial order of execution [16], [17]. The second algorithm is an online version of the first one. The main idea is to incrementally build a realizer using Rabinovitch and Rival's Theorem [12], and then create timestamp vectors based on that realizer. In the online algorithm, the vector timestamps that have already been assigned to events may have to be changed later on arrival of a new event. In fact, timestamp of an event may be changed multiple times. Further, all timestamps may not be of the same length. This leads to a somewhat complicated precedence test.

Ward and Taylor present an offline algorithm for timestamping events based on decomposing processes into a hierarchy of clusters [18]. The algorithm exploits the observation that events within a cluster can only be causally dependent

on events outside the cluster through receive events from transmissions that occurred outside the cluster. As a result, non-cluster receive events can be timestamped much more efficiently than cluster receive events.

Agarwal and Garg [2] have proposed a class of logical clock algorithms, called chain clock, for tracking dependencies between *relevant* events based on generalizing a process to any chain in the computation poset. Their algorithm reduces the number of components required in the vector when the set of relevant events is a small fraction of the total events. Our work is closely related to this work. They provide two different algorithms: the first algorithm adds any newly arrived event to a chain with the guarantee that no more than $|P|$ chains are necessary where P is the set of processes. The second algorithm uses online chain decomposition of a poset to guarantee that no more that $(w+1)w/2$ chains are necessary where w is the width of the poset. In this paper, our algorithm uses components that are either for the process or for the object and guarantees that the number of components is never more than $min(|P|, |O|)$.

Garg, Skawratananond, and Mittal [6] have proposed an algorithm to timestamp messages in a distributed system. They assume that all messages are synchronous and show that such systems can have timestamps of vector clocks with dimension less than N. They define the notion of an undirected communication graph with the set of vertices as processes and the edges denoting which processes can communicate. They show that the number of components required is equal to the number of stars and triangles the communication graph can be decomposed into. Our technique of using vertex cover is inspired from that work even though their work is strictly for distributed systems and they do not consider mixed-clocks. We have two types of entities in our system — threads and objects and the dimension of the vector clock reduces to a vertex cover of the bipartite graph that represents the interaction between these entities.

Vaidya and Kulkarni [15] propose an inline algorithm for assigning timestamps for events in an asynchronous distributed system, which achieves a trade-off between an offline algorithm and an online algorithm. Their inline algorithm assigns a timestamp for an event when it occurs but can change this timestamp later on. They also show that by exploiting the knowledge of the communication graph, their inline algorithm can typically assign a much smaller timestamp than online algorithms. Specifically, the components of the timestamp generated by the inline algorithm correspond to the vertex cover of the communication graph and some additional integers.

VII. Conclusions

This paper proposes an offline algorithm to compute a mixed vector-clock composed of a mix of threads and objects, which is shown to be a correct vector clock and have optimal size, to timestamp events in a computation. Thread-object bipartite graph is constructed based on the given computation and then the minimum vertex cover of this bipartite graph is computed. The threads and objects in this vertex cover are adopted as the components of the mix vector clock. In a online computation in which events are coming one by one, two mechanisms are proposed, the *Random* mechanism which randomly choose the associated thread or object as vector clock component and the *Popularity* mechanism which choose the thread or object based on their popularity. By evaluating on thread-object bipartite graphs with different characteristics, we get the conclusion that the *Popularity* mechanism shows best performance on *non-uniform* graphs.

Acknowledgment

We would like to thank Himanshu Chauhan and all the anonymous reviewers for their comments on this paper.

References

[1] Bondy J. A. and Murty U. S. R. *Graph Theory with Applications*. North Holland, 1976.

[2] A. Agarwal and V. K. Garg. Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems. In *Proceedings of the, ACM Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2005.

[3] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial-Ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, pages 56–66, February 1988.

[4] C. J. Fidge. Partial Orders for Parallel Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 183–194, January 1989.

[5] C. J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[6] Vijay K. Garg, Chakarat Skawratananond, and Neeraj Mittal. Timestamping messages and events in a distributed system using synchronous communication. *Distributed Computing*, 19(5-6):387–402, 2007.

[7] J.-M. Hélary, M. Raynal, G. Melideo, and R. Baldoni. Efficient Causality-Tracking Timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1239–1250, 2003.

[8] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–31, Dec 1973.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[10] K. Marzullo and L. Sabel. Efficient Detection of a Class of Stable Properties. *Distributed Computing (DC)*, 8(2):81–91, 1994.

[11] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[12] I. Rabinovitch and I. Rival. The Rank of Distributive Lattice. *Discrete Mathematics*, 25:275–279, 1979.

[13] M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters (IPL)*, 43:47–52, August 1992.

[14] F. J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *Proceedings of the 10th Workshop on Distributed Algorithms (WDAG)*, pages 71–88. Springer-Verlag, 1996.

[15] Nitin H Vaidya and Sandeep S Kulkarni. Efficient timestamps for capturing causality. *arXiv preprint arXiv:1606.05962*, 2016.

[16] P. A. S. Ward. An Offline Algorithm for Dimension-Bound Analysis. In Dhabaleswar Panda and Norio Shiratori, editors, *Proceedings of the International Conference on Parallel Processing*, pages 128–136. IEEE Computer Society, 1999.

[17] P. A. S. Ward. An Online Algorithm for Dimension-Bound Analysis. In P. Amestoy *et al*, editor, *Proceedings of the Euro-Par*, Lecture Notes in Computer Science (LNCS), pages 144–153. Springer-Verlag, 1999.

[18] P. A. S. Ward and D. T. Taylor. A Hierarchical Cluster Algorithm for Dynamic, Centralized Timestamps. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 585–593, April 2001.