# QuickLex: A Fast Algorithm for Consistent Global States Enumeration of Distributed Computations

**Yen-Jung Chang and Vijay K. Garg**

**Department of Electrical and Computer Engineering**
**University of Texas at Austin, TX, USA**
{cyenjung@, garg@ece.}utexas.edu

## Abstract

Verifying the correctness of executions of concurrent and distributed programs is difficult because they show nondeterministic behavior due to different process scheduling order. Predicate detection can alleviate this problem by predicting whether the user-specified condition (predicate) could have become true in any global state of the given concurrent or distributed computation. The method is predictive because it generates inferred global states from the observed execution path and then checks if those global states satisfy the predicate. An important part of the predicate detection method is *global states enumeration*, which generates the consistent global states, including the inferred ones, of the given computation. Cooper and Marzullo gave the first enumeration algorithm based on a breadth first strategy (BFS). Later, many algorithms have been proposed to improve the space and time complexity. Among the existing algorithms, the Tree algorithm due to Jegou et. al. has the smallest time complexity and requires $\mathcal{O}(|P|)$ space, which is linear to the size of the computation $P$. In this paper, we present a fast algorithm, QuickLex, to enumerate global states in the lexical order. QuickLex requires much smaller space than $\mathcal{O}(|P|)$. From our experiments, the Tree algorithm requires 2–10 times more memory space than QuickLex. Moreover, QuickLex is 4 times faster than Tree even though the asymptotic time complexity of QuickLex is higher than that of Tree. The reason is that the worst case time complexity of QuickLex happens only in computations that are not common in practice. Moreover, Tree is built on linked-lists and QuickLex can be implemented using integer arrays. In comparison with the existing lexical algorithm (Lex), QuickLex is 7 times faster and uses almost the same amount of memory as Lex. Finally, we implement a parallel-and-online predicate detector for concurrent programs using QuickLex, which can detect data races and violation of invariants in the programs.

**1998 ACM Subject Classification** D.2.4 [Software/Program Verification]: Validation

**Keywords and phrases** consistent global state, algorithm, computation

**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p

## 1 Introduction

The technique of predicate detection is first proposed for distributed debugging [6]. Many tools also use this technique for detecting various types of bugs in concurrent systems [5, 10, 22, 17]. The problem of predicate detection is to detect if the user-specified condition (or simply *predicate*) could happen in the given concurrent or distributed computation, which models the execution trace of concurrent or distributed programs as a partially ordered set (*poset*) of events; each event corresponds to an operation of the program. On this poset, the inferred consistent global states of the system are generated and checked if any one of them satisfies the predicate.

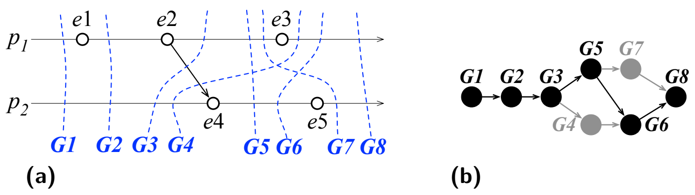

**Figure 1** (a) The captured logical order between events, which form a poset. The dashed lines are consistent global states of the program. (b) The relationship of the set of consistent global states.

We use Fig. 1a to explain the technique of predicate detection. In this computation, the event $e2$, which occurs on process $p_1$, sends a message to event $e4$, which occurs on process $p_2$. Because of the message, the causal dependency between $e2$ and $e4$ is established. We assume that the messages may have arbitrary delays but no process exhibits faulty behavior. Informally, a global state is consistent if there exists an execution path to reach the state. In Fig. 1a, the dashed lines show all consistent global states of the computation and each global state contains all the events to the left of the corresponding dashed line. For example, the global state $G4$ contains events $e1$, $e2$, and $e3$.

Fig. 1b shows the relationship of the consistent global states in Fig. 1a. Assume that the sequence $G1, G2, G3, G5, G6, G8$ of global states is the observed execution of the program. The objective of predicate detection is to generate the inferred global states $G4$ and $G7$. Hence, we do not need to re-execute the program in order to reach $G4$ and $G7$. In this paper, we study the method for enumerating all consistent global states, including the inferred ones, of the given computation. From now on, the term *computation* refers to a concurrent or a distributed computation, the term processes refers to threads in a concurrent computation or processes in a distributed computation, and the term *global state* means consistent global state; unless specified otherwise.

Enumerating all global states of a computation $P$ requires exponential time because the number of global states, $i(P)$, grows exponentially in $n$, which is the number of processes in the computation. With some particular assumptions, it may be reduced to polynomial time because only a partial set of global states is enumerated [10, 18, 24, 25, 27]. If no assumption is made regarding the predicate, i.e., the enumeration algorithm is general-purpose, then enumerating every global state is necessary. Thus, the time complexity of a general-purpose algorithm can be calculated by multiplying $i(P)$ by *the time complexity per global state*, which is the time to advance from one global state to the other. For simplicity, we use the time complexity per global state to represent the time complexity of a general-purpose algorithm.

Cooper and Marzullo [6] gave the first general-purpose enumeration algorithm based on a breadth first strategy (BFS) that requires $\mathcal{O}(n^3)$ time and exponential space in $n$, which is the number of processes in the computation $P$. Alagar and Venkatesan [2] presented the notion of global interval which reduces the space complexity to $\mathcal{O}(|P|)$. Steiner [29] gave an algorithm that uses $\mathcal{O}(|P|)$ time, and Squire [28] further improved the computation time to $\mathcal{O}(log|P|)$. Pruesse and Ruskey [26] gave an algorithm that enumerates global states in a combinatorial Gray code manner. The algorithm uses $\mathcal{O}(|P|)$ time and can be reduced to $\mathcal{O}(\Delta(P))$, where $\Delta(P)$ is the maximal in-degree of any event; however, the space grows exponentially in $|P|$. Later, Jegou et al. [19] and Habib et al. [15] improved the space complexity to $\mathcal{O}(|P|)$. Ganter [11] presented an algorithm, which enumerates global states in lexical order, and Garg [12] gave an implementation using vector clocks [9, 23]. The lexical algorithm requires $\mathcal{O}(n^2)$ time, but the algorithm requires no additional space besides the input, i.e., the computation. Note that the space complexity of an enumeration algorithm only considers the memory space that stores the intermediate information during the enumeration. Table 1 summarizes

**Table 1** Time and space complexity of algorithms.

| Algorithms | Time per Global State | Space |
|---|---|---|
| ***Cooper–Marzullo*** [6] | $\mathcal{O}(n^3)$ | exp. in $n$ |
| ***Alagar–Venkatesan*** [2] | $\mathcal{O}(n^3)$ | $\mathcal{O}(\lvert P\rvert)$ |
| ***Steiner*** [29] | $\mathcal{O}(\lvert P\rvert)$ | not available |
| ***Squire*** [28] | $\mathcal{O}(log\lvert P\rvert)$ | not available |
| ***Pruesse–Ruskey*** [26] | $\mathcal{O}(\lvert P\rvert)$ | exp. in $\lvert P\rvert$ |
| ***Jegou and Habib et al.*** [19, 15] | $\mathcal{O}(\Delta(P))$ | $\mathcal{O}(\lvert P\rvert)$ |
| ***Lexical*** [11, 12] | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ |
| ***QuickLex*** | $\mathcal{O}(n\cdot \Delta(P))$ | $\mathcal{O}(n^2)$ †) |

†) $n$ is the number of processes in the computation $P$. Thus, $n^2$ is usually much smaller than $|P|$ because $|P| = n \times m$, where $m$ is the least number of events per process and $m >> n$.

the time and space complexity of the algorithms and Table 4 in Appendix A lists the symbols that are used in this paper.

In this paper, we present QuickLex — a fast algorithm for global states enumeration in lexical order. In comparison with the existing lexical algorithm (Lex) [11, 12], QuickLex reduces the time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n\cdot \Delta(P))$. The time complexity can be reduced to $\mathcal{O}(n)$ for the commonly used computations [5, 21, 10, 15, 19], in which most events send and receive at most one message.

Both QuickLex and Lex algorithms enumerate global states in the same order. However, they are fundamentally different in computing the next global state in the lexical order. The Lex algorithm simply uses the current global state and vector clocks to determine the next global state. Thus, it has to repeatedly calculate the information that is reusable. QuickLex reduces the computational cost using two approaches. First, it preprocesses the computation and pre-calculates the statically reusable information. Second, it incorporates dynamic programming to reuse the dynamic information during the enumeration.

We evaluate QuickLex using multiple benchmarks including four computations that are captured from the executions of real-world applications. In our experiments, QuickLex is 7 times faster than the Lex algorithm [11, 12] and 4–5 times faster than the Tree algorithm [15, 19]. We note here that QuickLex is faster than the Tree algorithm even though the asymptotic worst case time complexity for the Tree algorithm is lower. There are two reasons for this. First, the time complexity of QuickLex is calculated as the worst case, which is not a common computation in practice. Second, the Tree algorithm needs to store its temporary information, which is a spanning tree, in a linked-list, which induces large overhead during enumeration. As far as space complexity is concerned, QuickLex uses almost the same amount of memory as Lex, which shows that the extra space for dynamic programming in QuickLex is quite small. The Tree algorithm uses 2–10 times more memory than QuickLex.

In [4], we have discussed a technique, named ParaMount, to decompose any lattice of global states into multiple sublattices and to enumerate those using Lex in a parallel-and-online fashion. Since Lex and QuickLex are similar, we can easily speed up ParaMount by replacing Lex with QuickLex. The experimental results show that QuickLex speeds up ParaMount by a factor of 3. The technique in [4] focuses on the high-level parallelization of the enumeration of consistent global states, which uses sequential enumeration algorithms for its subroutines. In this paper, we focus on the fast sequential enumeration algorithm.

The rest of the paper is organized as follows. Section 2 gives the model of computation. Section 3 presents the algorithm of QuickLex. Section 4 shows the experimental results. Section 5 discusses the applications of QuickLex. Section 6 concludes this paper.

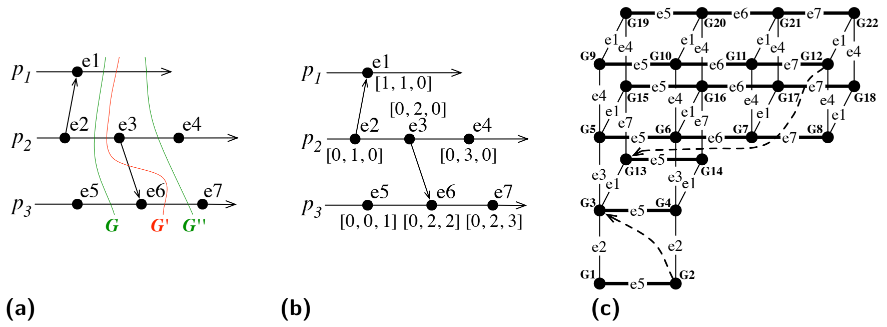

**Figure 2** (a) A computation is composed of a partially ordered set (*poset*) of events. $G$ and $G''$ are consistent global states and $G'$ is an inconsistent global state. (b) The vector clocks of the events. (c) The distributive lattice formed by the set of consistent global states of the computation.

## 2 The Model of Computations

The observed execution of the program is modeled as a computation that is composed of a poset $P = (E, \rightarrow)$ of events, which contains a set $E$ of events together with Lamport's happened-before (HB) relation $\rightarrow$ [20]. Fig. 2a shows a graphical representation of a computation with three processes $p_1$, $p_2$, and $p_3$. The horizontal arrows represent the total order of the events that occur on the same process. The arrows between two events that occur on different processes represent messages. The HB relation between two events $e$ and $f$ is established by the following rules:

1. If $e$ occurs before $f$ on the same process, then $e \rightarrow f$.
2. If $e$ sends a message and $f$ receives the message, then $e \rightarrow f$.
3. If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

In the computation, the HB relation between events is captured using vector clocks [9, 23]. A vector clock, $vc$, is an array of integers. For an event $e$, which occurs on process $p_i$, the integer $e.vc[i]$ is the index of $e$ among the events that occur on $p_i$. For $j \neq i$, $e.vc[j]$ is the largest index of event $f$ among the events that occur on process $p_j$ such that $f \rightarrow e$. For instance, the vector clock of event $e7$ in Fig. 2b is $[0, 2, 3]$, which means the index of the current event $e7$ is 3. Moreover, the event $e3$, which has index 2 in $p_2$, happened before $e7$.

### 2.1 The Lattice of Consistent Global States

A *consistent global state* $G$ is a subset of $E$, such that if $G$ includes any event $f$, then it also includes all events that happened before $f$ [3]. Formally, $G \subseteq E$ is a consistent global state if

$$\forall e, f : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G).$$

In Fig. 2a, for instance, the global states $G$ and $G''$ are consistent and $G'$ is not, because $e3 \rightarrow e6$ but $e3 \notin G'$.

A global state can equivalently be identified by the maximal events of each process. These maximal events are simply represented by an array of integers, in which the $i$-th integer indicates the index of the maximal event among the events that occur on process $p_i$. If the index is zero then no event on the corresponding process is included in the global state. For instance, $G''$ in Fig. 2a is represented by $[1, 2, 2]$. The symbol $G[i]$ denotes the maximal event of process $p_i$ in $G$, e.g., $G''[2]$ refers to event $e3$.

The set of consistent global states forms a distributive lattice [7]. Fig. 2c shows the lattice that is formed by the consistent global states of the computation. Each node of the lattice corresponds to a consistent global state and the edge label denotes the event that takes the system from one consistent global state to the other. *The objective of QuickLex is to enumerate the lattice of consistent global states of the computation in the lexical order.* From now on, the term *global state* means consistent global state; unless specified otherwise.

## 2.2 Lexical Order among the Global States of the Computation

A lexical algorithm explores the lattice of global states using a pre-defined total order, called lexical order (denoted $\prec$), among the global states. The order $\prec$ is defined on global states as follows:

$$G \prec G' \Leftrightarrow \exists k : (\forall i : 1 \le i < k : G[i] = G'[i]) \wedge (G[k] < G'[k]),$$

where $G$ and $G'$ are two arbitrary global states in the lattice. In Fig. 2c, the lexical order of the two global states $G2 = [0,0,1]$ and $G3 = [0,1,0]$ is $G2 \prec G3$. The number of each global state in Fig. 2c is its lexical order among the global states in the lattice.

## 2.3 Remote Events and Predecessor of an Event

If an event $r$ sends a message to an event $e$, $r$ is the remote event of $e$. Formally, an event $r$ is a *remote* event of event $e$ if 1) $r \to e$, 2) $r$ and $e$ occur on different processes, and 3) there does not exist any event $f$ such that $r \to f \to e$. If an event does not have any remote event, it is a local event. In Fig. 2a, for example, event $e6$'s remote event is event $e3$. Similarly, event $d$ is the *predecessor* of $e$ if 1) $d \to e$, 2) $d$ and $e$ occur on the same process, and 3) there does not exist any event $f$ such that $d \to f \to e$. In Fig. 2a, event $e6$'s predecessor is $e5$.

## 3 QuickLex

### 3.1 Overview

For simplicity, we consider the array of indices of a global state as a number and each index is a single digit of that number. Fig. 3 shows the mapping between an array of indices and a number of digits. In a global state, the processes at the left are high priority processes and those at the right are low priority processes.

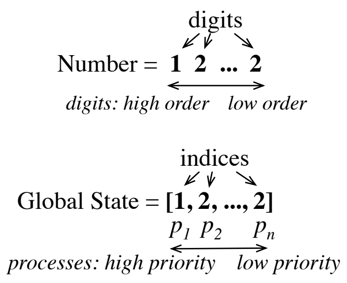

**Figure 3** A number consists of multiple digits and the array of indices, which is considered as a number and each index is considered as a digit of that number.

**Algorithm 1** QuickLex($P$)

**Input:** A computation $P$ with $L$ as the least global state and $M$ as the greatest global state.

```
1: $G := L$                              ▷ Use $L$ as the initial global state.
2: for every event $e$ in $P$ do LOCATEREMOTEEVENTS($e$)
3: INITIALSTACKS()
4: while true do
5:     enumerate($G$)                    ▷ Evaluate the predicate on $G$.
6:     $k :=$ PROPAGATE($M$)             ▷ Find $p_k$ to propagate.
7:     if $k < 1$ then break             ▷ true: no process to propagate.
8:     $G[k] := G[k] + 1$                ▷ Add the new event $e_k$ into $G$.
9:     RESET($k$)                        ▷ Reset the maximal events of lower priority processes, i.e., $p_{k+1}$ to $p_n$.
10: end while
```

To advance from one global state to the other (which is also referred as one *iteration* in this paper) in the lexical order, we use the notion of *carrying over* from algebraic addition, in which we continuously add one to the low-order digit of a number and propagate the carry to a higher order digit that has not reached its limit. Then, all lower order digits are reset to their least value. Similarly, QuickLex contains two main parts. The first part adds the next event of the least priority process $p_n$ into the current global state. If the next event of $p_n$ is not available (e.g., if the limit of the digit is reached), the carry is propagated to a higher priority process, say $p_k$. The second part resets the maximal events of lower priority processes, i.e., $p_{(k+1)}$ to $p_n$. *The challenge for the two parts is that the limit and the least value of a digit are not fixed; they dynamically change with the values of other digits.*

Algorithm 1 shows the pseudo code of QuickLex, which takes as input a computation $P$. The least global state $L$ and the greatest global state $M$ of $P$ are acquired from the computation itself and no additional calculation is needed. Take Fig. 2a for example, where $L = [0, 0, 0]$ and $M = [1, 3, 3]$. QuickLex enumerates every global state $G$ such that $L \preceq G \preceq M$. The function LOCATEREMOTEEVENTS at line 2 pre-calculates the reusable information for the PROPAGATE procedure. The function INITIALIZESTACK at line 3 initializes the memory space for dynamic programming, which speeds up the RESET procedure.

***Part 1 (lines 6-8):*** Informally, an event is *enabled* if it can be added into the current global state $G$ without violating the consistency of $G$. Therefore, there might be multiple enabled events with respect to $G$. Since we enumerate global states in the lexical order, the PROPAGATE procedure locates the enabled event that occurs on the process that has the least priority, say $p_k$. If $k$ is 0, then the next global state has exceeded the maximal global state $M$ and hence the enumeration is terminated; otherwise, the enabled event is added into $G$.

When $k$ is decided by the PROPAGATE procedure, the processes in the computation are divided into two sets: $P_h$ and $P_l$. The set $P_h$ contains the processes whose priorities are higher or equal to process $p_k$, and $P_l$ contains those whose priorities are lower than $p_k$. In Fig. 2a, for example, if $k = 2$, then $P_h = \{p_1, p_2\}$ and $P_l = \{p_3\}$. From now on, the symbols $p_h$ and $p_l$ denote an arbitrary process in $P_h$ and $P_l$, respectively. Moreover, $h \leq k < l$.

***Part 2 (line 9):*** After part 1, the maximal events for $P_h$ are decided and fixed. Thus, we need to ensure that all the events of $P_l$ that happened before the events of $P_h$ will be included in the next global state. We define the *maximum dependency event* of any process $p_l$ as the event, which has the largest index among the events that occur on $p_l$, that has to be included in $G$ due to the consistency of the HB relation. The procedure RESET finds the maximum dependency event for every $p_l$.

The details of the first and second part of QuickLex are described next.

## 3.2 Part 1: Procedure PROPAGATE and the Enabled Event $e_k$

We first use Fig. 2 to show how part 1 works during an iteration of QuickLex. Assume that the current global state is $G2 = [0, 0, 1]$ and thus the next global state to be enumerated is $G3 = [0, 1, 0]$. The advance from $G2$ to $G3$ is shown as a dashed arrow in Fig. 2c. First, event $e6$ is considered as the next event to be added into $G2$. However, $e6$ cannot be included in $G2$ because $e3 \rightarrow e6$ and $e3 \notin G2$, i.e., $e6$ is not enabled. Thus, the carry is propagated to $p_2$. Since event $e2$ is enabled, it is added to $G2$. Now, we have reached an intermediate global state $[0, 1, 1]$. In this example, the maximal event $G[3]$ of $p_3$ will be reset to 0 in the second part of QuickLex and hence $G3 = [0, 1, 0]$ is reached.

▶ Definition 1. An event $e$ is enabled in a global state $G$ iff all events that happened before $e$ are included in $G$.

**Algorithm 2** Locate the set $R(e)$ of remote events for event $e$

```
1: function LOCATEREMOTEEVENTS($e$)
2:     Let $d$ be $e$'s predecessor.
       ▷ Find the new HB relation on event $e$.
3:     for $i$ from 1 to $n$ except $e.pid$ do    ▷ $e.pid$ is the id of the process on which $e$ occurs.
4:         if $d.vc[i] \neq e.vc[i]$ then Add $event(i, e.vc[i])$ into $RCandidate$.
5:     end for
       ▷ Find the direct HB relation on event $e$.
6:     for every $r \in RCandidate$ do
7:         Let $r'$ be any other event in $RCandidate$.
8:         if $r.vc[r.pid]$ is larger than all $r'.vc[r.pid]$ then Add $r$ into $R(e)$.
9:     end for
10: end function
```

**Algorithm 3** Procedure PROPAGATE and Function ISENABLED

```
Input: The maximal global state $M$.
Output: The process $p_k$ to propagate.
1: procedure PROPAGATE($M$)
2:     for $k$ from $n$ to 1 do    ▷ From $p_n$ to $p_1$.
3:         if $G[k]+1 \leq M[k]$ then    ▷ $G + e_k \preceq M$
4:             $e_k :=$ the next event on process $p_k$.
5:             if ISENABLED($e_k$) then return $k$
6:         end if
7:     end for
8:     return 0    ▷ No process to propagate.
9: end procedure

Input: The next event $e_k$ on process $p_k$.
Output: Returns true if $e_k$ is enabled w.r.t. $G$.
10: function ISENABLED($e_k$)
11:     if $e_k$ is a local event then return true
12:     if $\forall r \in R(e_k)$ s.t. $r.vc[r.pid] > G[r.pid]$ then return true    ▷ $r.pid$ is the id of the process on which $r$ occurs.
13:     return false
14: end function
```

Assuming that event $e$ occurs on process $p_i$, this condition can be determined using the property of vector clocks [9, 23]: $(e.vc[i] = G[i]+1) \wedge (\forall j \neq i : e.vc[j] \leq G[j])$. Unfortunately, it takes $\mathcal{O}(n)$ time to compare the vector clocks in the later part of the condition. QuickLex uses the following theorem to reduce the time complexity to $\mathcal{O}(\Delta(P))$, where $\Delta(P)$ is the maximal number of remote events for any event:

▶ **Theorem 1.** Let $R(e)$ be the set of remote events of event $e$, which occurs on process $p_i$, and event $d$ be the predecessor of $e$, then $e$ is enabled iff $d \in G$ and $\forall r \in R(e) : r \in G$.

**Proof.** In Appendix B. ◀

Theorem 1 reduces the computational cost of the procedure that determines whether event $e$ is enabled by ignoring the events that transitively happened before $e$. For example, if event $e$ is a local event, which does not have any remote event, then $e$ is enabled when its predecessor is included in $G$. In a computation $P$, $\Delta(P)$ is at most $(n-1)$ because there are at most $(n-1)$ events that occur on different processes and send messages to $e$. If any event in $P$ can have at most one remote event [5, 21, 10, 15, 19], then $\Delta(P)$ can be reduced to $\mathcal{O}(1)$.

Algorithm 2 uses the property of vector clocks to locate the set $R(e)$ of remote events for any event $e$. The function has two steps. In the first step (lines 2-5), the vector clock of $e$ and that of $e$'s predecessor are compared. If the $i$-th value (except the one for $e$ itself) of $e$'s vector clock is updated, then a new HB relation is established between $e$ and *event(i, e.vc[i])*, which is the event, whose index is $e.vc[i]$, that occurs on process $p_i$. However, we are interested in only direct HB relation because of Theorem 1. Thus, the second step (lines 6-9) uses another property of vector clocks: if event $r$ has not happened-before event $r'$, then the vector clock of $r'$ does not contain $r$'s latest clock value, i.e., $r.vc[r.pid]$, where $pid$ is the id of the process on which $r$ occurs. Note that Algorithm 2 is invoked only once in the beginning of QuickLex and the calculated $R(e)$ for event $e$ is reused during the enumeration.

Algorithm 3 shows the procedure PROPAGATE. The procedure decides which process to propagate starting from the least to the highest priority processes in order to follow the

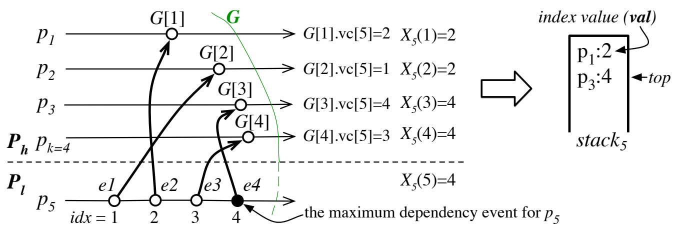

**Figure 4** The symbol $X_l(i)$ denotes the function $max_{1 \le j \le i} G[j].vc[l]$. The upside-down $stack_5$ on the right is the actual $stack_l$ that is used by QuickLex.

lexical order. Moreover, the event that occurs after the currently maximal event of process $p_k$ is chosen. Thus, the predecessor of $e_k$ is always included in $G$. The function isEnabled checks if either one of the following two conditions holds to determine whether $e_k$ is enabled: 1) $e_k$ is a local event or 2) all remote events of $e_k$ are included in $G$. If any event in the computation has at most one remote event, then isEnabled takes constant time. If $e_k$ is enabled, then propagate has found the process $p_k$ and it returns $k$. If the process $p_k$ does not exist, which implies that $M$ is reached, then propagate returns 0.

## 3.3 Part 2: Procedure reset and the Maximum Dependency Events

The maximal events of $P_l$ are not always reset to index 0. Assume that we are advancing from $G12 = [0, 3, 3]$ to $G13 = [1, 1, 0]$ in Fig. 2. After propagate decides $k = 1$, we reach the intermediate global state $[1, 3, 3]$. However, we cannot simply reset the global state to $[1, 0, 0]$ because it is not consistent; it includes $e1$ but does not include $e2$ even though $e2 \to e1$ (see Fig. 2a). So, the procedure reset has to find the *maximum dependency events* of $p_2$ and $p_3$ that would satisfy the consistency of the global state.

From now on, the symbol $G_m[l]$ denotes the maximum dependency event of $p_l$, which becomes the maximal event $G[l]$ of $p_l$ after reset. When $e_k$ is decided, the maximal events of $P_h$ are also decided. The maximum dependency event $G_m[l]$ for every $p_l$ can be calculated using the property of vector clocks:

$$G_m[l] = \max_{1 \le j \le n} (G[j].vc[l])$$

For simplicity, the expression $\max_{1 \le j \le i}(G[j].vc[l])$ is denoted by the symbol $X_l(i)$ from now on. Fig. 4 shows how the maximum dependency event $G_m[l]$ of a process $p_l$ is identified by $X_l(n)$. In Fig. 4, the events $e1$, $e2$, $e3$, and $e4$ are four events that occur on process $p_5$. Assume that their indices are 1, 2, 3, and 4, respectively. Suppose that $k = 4$. Thus, $G[4]$ is the new event $e_k$. The fifth indices of the vector clocks of the maximal events of $p_1$, $p_2$, $p_3$, and $p_4$ are shown in the figure (i.e., $G[1].vc[5]$, $G[2].vc[5]$, $G[3].vc[5]$, and $G[4].vc[5]$). The bold arrows between events are the HB relations that are obtained from these indices. Since $G[3].vc[5]$ has the largest index, i.e., 4, it follows that $e4$ is the maximum dependency event of $p_5$. In other words, $G_m[5] = X_5(4) = 4$.

In fact, $G_m[l]$ can be identified by $X_l(k)$ instead of $X_l(n)$:

▶ Theorem 2. For a global state $G$, $k$, and any process $p_l$, $X_l(i) = X_l(k)$ for all $i > k$.

**Proof.** Assume that the condition is not true, i.e., $\exists i : i > k : X_l(i) > X_l(k)$. The condition implies that $G_m[l] \to e_i$, which is an event that occurs on process $p_i$. Because $i > k$, we get $p_i \in P_l$ and thus $e_i \to G_m[i]$; so $e_i$ is included in $G$. Moreover, since $G_m[i]$ is a maximal dependency event, there exists an event $e_h$ such that $G_m[i] \to e_h$, where $e_h$ occurs on a process $p_h$, where $h \le k$.

**Algorithm 4** Incremental update of array $X_l$

```
Input: The process id of $p_l$, the decided $k$, and $\forall i : 1 \le i \le n : X_l[i] = X_l(i)$ w.r.t. global state $F$.
Output: $\forall i : 1 \le i \le n : X_l[i] = X_l(i)$ w.r.t. global state $G$.

1: function UPDATEARRAYX($l, k$)
2:     $X_l[k] := \max\big(X_l[k-1], G[k].vc[l]\big)$
3:     for $i$ from $(k+1)$ to $n$ do $X_l[i] := X_l[k]$
4: end function
```

**Algorithm 5** Initialize stacks for every process

```
1: function INITIALIZESTACKS()
2:     for $i$ from 1 to $n$ do ▷ For every process $p_i$ in $P$.
3:         push $[p_1 : G[1].vc[i]]$ into $stack_i$
4:         for $j$ from 1 to $(i-1)$ do ▷ $k < i$ is always true.
5:             if $top.val < G[j].vc[i]$ then
6:                 push $[p_j : G[j].vc[i]]$ into $stack_i$
7:         end for
8:     end for
9: end function
```

Due to the transitivity of HB relation, we get $G_m[l] \to e_i \to G_m[i] \to e_h$ and hence $X_l(h)$ also contains the largest value of $X_l(i)$. Since $h \le k < i$, we get $X_l(h) = X_l(k) = X_l(i)$, which contradicts the assumption. ◀

According to Theorem 2, $X_l(k)$ has had the largest clock value among $X_l(i)$ for all $i$. Consequently, $G_m[l]$ can be identified by $X_l(k)$. Now we show how to calculate the value of $X_l(k)$ in amortized constant time for each iteration using dynamic programming. It is easy to see that the value of $X_l(i)$ can be represented with the following recursive equation:

$$X_l(i) = \begin{cases} G[1].vc[l], & \text{if } i = 1 \\ max\big(X_l(i-1), G[i].vc[l]\big), & \text{otherwise} \end{cases} \tag{1}$$

We use an auxiliary integer array $X_l$ for each process $p_l$, in which each value $X_l[i]$ stores the true value of $X_l(i)$. Note that $X_l(i)$ is the true value of $\max_{1 \le j \le i}(G[j].vc[l])$ and $X_l[i]$ is a calculated result. The array $X_l$ has to satisfy the invariant:

$\forall i : 1 \le i \le n : X_l[i] = X_l(i)$

For any global state $G$ and a given $k$, we can calculate the array $X_l$ for each process $p_l$ with respect to $G$. Assume that $F$ is the previous global state of $G$ in the lexical order. Instead of calculating the array $X_l$ for $G$ from scratch, we incrementally construct $X_l$ from that of $F$. The incremental update procedure is shown in the function UPDATEARRAYX in Algorithm 4.

▶ **Theorem 3.** Function UPDATEARRAYX maintains the invariant of $X_l$ after the incremental update.

**Proof.** We consider the three intervals of the values in $X_l$:

(a) $i < k$: Since the maximal events of $P_h$ are not changed, the true values of $X_l(i)$ for $i < k$ remain the same. Thus, UPDATEARRAYX does not need to update $X_l[i]$ for $i < k$.

(b) $i = k$: $X_l[i]$ is updated at line 2 using equation (1), where the true value of $X_l(i-1)$ is obtained from $X_l[i-1]$.

(c) $i > k$: $X_l[i]$ is updated at line 3 using Theorem 2. ◀

Since the results of $X_l$ are non-decreasing, we only need to store the values that are larger than their previous one and the process ids of the events that provide the values. For instance, $stack_5$ in Fig. 4 is the actual stack (which is shown upside down) for storing the results of $X_5$. In $stack_5$, the top entry $[p_3 : 4]$ means $X_5[3] = X_5[4] = \cdots = X_5[n] = 4$ and the bottom entry $[p_1 : 2]$ means $X_5[1] = X_5[2] = 2$.

**Algorithm 6** Function updateStack and Procedure reset

```
Input: The process id of $p_l$ and the decided $k$.
Output: The top value of $stack_l$ is $G[l]$.
 1: function updateStack($l, k$)
 2:     pop $stack_l$ until $top.pid \le k$.
 3:     if $top.val < G[k].vc[l]$ then
 4:         if $top.pid = k$ then $top.val := G[k].vc[l]$
 5:         else push $[p_k : G[k].vc[l]]$ into $stack_l$
 6:     end if
 7: end function

Input: The decided $k$.
Output: The maximum dependency events of $P_l$ are found.
 8: procedure reset($k$)
 9:     for $l$ from $(k+1)$ to $n$ do
10:         UpdateStack($l, k$)
11:         $G[l] := top.val$          ▷ Set $G[l]$ to $G_m[l]$.
12:     end for
13: end procedure
```

Algorithm 5 constructs the $stack_i$ of each process $p_i$ for the initial global state of a computation, which is $[0, 0, ..., 0]$. Although $k$ does not exist in the initial global state, we know that $k < i$ for each process $p_i$ because of the definition of $P_l$. Therefore, it is safe to assume that $k = (i-1)$ when constructing $stack_i$. It is easy to see that the construction of $stack_i$ is equivalent to the construction of the array $X_i$. Moreover, the function updateArrayX in Algorithm 4 can be converted to the function updateStack in Algorithm 6. Line 2 of updateArrayX is equivalent to lines 2-6 of updateStack and line 3 of updateArrayX is achieved by the property of $stack_l$.

▶ Theorem 4. $G_m[l]$ can be identified using $stack_l$ in an amortized constant time per global state.

**Proof.** At line 2 of Algorithm 6, if $stack_l$ pops $m$ entries, then there exist $m$ iterations that cumulatively pushed $m$ entries into $stack_l$. Therefore, the cost of the pop operations can be evenly charged to the $m$ iterations and be reduced to amortized constant time. The operations at lines 4 and 5 take constant time. As a result, the time complexity for updating a stack is amortized constant time per global state. ◀

Finally, lines 8-13 of Algorithm 6 shows the procedure reset, which updates $stack_l$ for every $p_l$. The maximum dependency event of $p_l$ is identified from the top entry of $stack_l$.

## 3.4 The Correctness and Worst Time Complexity of QuickLex

▶ Theorem 5. QuickLex enumerates the lattice of global states of a computation in the lexical order such that every global state is enumerated exactly once.

**Proof.** Assume that $F$ is the previously enumerated global state and $G$ is the current global state to be enumerated.

**Lexical Order:** Since propagate adds a new event $e_k$ to $F$, we get $\exists k : (\exists i : 1 \le i < k : F[i] = G[i]) \land (F[k] < G[k])$ and hence $F \prec G$.

**Exactly Once:** Since $F \prec G$, every global state is enumerated at most once. We next show that every global state is enumerated at least once. Since $F \prec G$, we get $\forall i : 1 \le i < k : F[i] = G[i]$ and $G[k] = F[k] + 1$. Assume that $F'$ is a consistent global state such that $F \prec F' \prec G$. We consider the following cases:

(a) $F'[k] < F[k]$: This case implies that $F' \prec F$, which contradicts the assumption $F \prec F'$.

(b) $F'[k] = F[k]$: Since $F \prec F'$, this case implies that there exists a process $p_{k'}$ such that $k' > k$ and $p_{k'}$ has an enabled event w.r.t. $F$. However, propagate locates the enabled event from $p_n$ to $p_1$ and hence $k' \le k$. A contradiction.

(c) $F'[k] = F[k] + 1 = G[k]$: After RESET, any $p_l$ cannot have a maximal event that is smaller than its maximum dependency event $G_m[l]$ due to the consistency of the HB relation. Thus, we get $\nexists l : F'[l] < G[l] = G_m[l]$. So, $F'$ does not exist.

(d) $F'[k] > F[k] + 1 = G[k]$: This case implies that $G \prec F'$, which contradicts the assumption $F' \prec G$. ◀

▶ **Theorem 6.** The worst case time complexity of QuickLex is $\mathcal{O}(n \cdot \Delta(P))$ per global state.

**Proof.** There are two main procedures during each iteration of QuickLex: PROPAGATE and RESET. We first analyze the worst time complexity of PROPAGATE. Each invocation the function ISENABLED takes $\mathcal{O}(\Delta(P))$ time and the *for* loop of PROPAGATE is executed at most $n$ iterations. So, the worst time complexity of PROPAGATE is $\mathcal{O}(n \cdot \Delta(P))$ time.

We now analyze the worst case time complexity of RESET. Each invocation of the function UPDATESTACK takes amortized $\mathcal{O}(1)$ time and the *for* loop of RESET is executed at most $n$ iterations. So, the worst time complexity of RESET is amortized $\mathcal{O}(n)$ time. As a result, the worst time complexity of each iteration of QuickLex is $\mathcal{O}(n \cdot \Delta(P))$. ◀

## 4 Evaluation

### 4.1 Setup of Benchmarks

**Table 2** The information of benchmarks and runtimes (sec.) of each algorithm.

| Benchmark | $n$ | #events | #global states | BFS | Tree | Lex | QuickLex |
|---|---|---|---|---|---|---|---|
| ***d-300*** | 10 | 300 | 42,695,907 | 58.43 | 3.80 | 3.41 | 0.76 |
| ***d-500*** | 10 | 500 | 237,475,992 | 375.06 | 19.40 | 18.67 | 3.78 |
| ***d-10K*** | 10 | 10,000 | 4,962,876,973 | 8,211.87 | 393.74 | 448.28 | 86.38 |
| ***bank*** | 8 | 96 | 815,730,721 | out of memory | 56.67 | 64.37 | 9.69 |
| ***tsp*** | 8 | 105,282 | 13,474,170 | 9.85 | 1.63 | 2.37 | 0.37 |
| ***hedc*** | 12 | 216 | 4,486,599,595 | out of memory | 322.04 | 488.22 | 78.34 |
| ***elevator*** | 12 | 38,528 | 27,643,588,608 | out of memory | 2,248.39 | 4,677.12 | 660.40 |
| ***w-4*** | 4 | 480 | 9,381,251 | 2.51 | 0.88 | 0.38 | 0.16 |
| ***w-8*** | 8 | 480 | 7,392,009,768 | out of memory | 609.74 | 454.28 | 128.03 |
| ***w-12*** | 12 | 480 | 206,379,406,870 | out of memory | 19,225.98 | 21,303.66 | 3,996.17 |
| ***w-16*** | 16 | 480 | 991,493,848,554 | out of memory | 111,452.52 | 179,844.62 | 23,263.05 |

Table 2 shows the information of the benchmarks that are used in the experiments. The benchmarks contain three different sets of computations. The benchmarks that start with the prefix "*d-*" are randomly generated posets of events for modeling distributed computations. The benchmarks *bank*, *tsp*, *hedc*, and *elevator* are the computations that are captured from the executions of real-world concurrent applications. We establish the HB relation in these concurrent computations using the following rules [10, 21]:

1. If $e$ occurs before $f$ on the same thread, then $e \rightarrow f$.
2. If event $e$ corresponds to a thread releasing a lock and $f$ corresponds to subsequent acquisition of that lock (including implicit locks and monitors), then $e \rightarrow f$.
3. If the parent thread forks a new thread on event $e$ and the child thread is created on event $f$, then $e \rightarrow f$. Similarly, if a child thread terminates on event $e$ and the parent thread joins the child thread on event $f$, then $e \rightarrow f$.
4. If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

The benchmark *banking* contains a typical error pattern in concurrent programs [8]; *tsp* is a parallel solver for the traveling salesman problem; *hedc* is a crawler for searching Internet archives; and *elevator* is a discrete event simulator for an elevator system. The benchmarks *tsp*, *hedc*, and *elevator* are the benchmark programs that are used in [5, 10, 30].

Finally, the benchmarks that start with the prefix "$w$-" have the same number of events, i.e., 480 events, but different number of processes in the computation. The set of benchmarks is used to show how different $n$ influences the performance of enumeration algorithms, and therefore we keep the number of events constant.

## 4.2 Compared Enumeration Algorithms

Besides QuickLex, we implemented the breadth-first strategy (BFS) algorithm [6, 12], the ideal tree traversal algorithm (Tree) [19, 15], and the original lexical algorithm (Lex) [11, 12]. In BFS algorithm [6], a global state might be enumerated more than once, so we use the strategy in [12] to ensure that every global state is enumerated exactly once. In our experiments, we use the improved BFS algorithm.

For Lex [12], we improve the nested *for* loops of function LeastGlobalState(). Each of the *for* loop goes through process $p_1$ to process $p_n$, which takes $\mathcal{O}(n^2)$ time. However, looping through all processes is not necessary. We modify the first loop, which only loops from $p_1$ to $p_k$, and the second loop, which only loops from $p_{k+1}$ to $p_n$. Although the time complexity remains the same, the practical runtime is improved significantly. In our experiments, we use the improved Lex algorithm.

The Tree algorithm [15, 19] finds a backward spanning tree in the lattice of global states, where the root is the global state that contains all events, e.g., the state $G22$ that is shown in Fig. 2c. Then it traverses the spanning three in a depth-first manner. The performance of Tree mainly dependents on *SList* [19], which is a customized linked list that continuously adds and removes the nodes of the spanning tree. So, we use the following implementation techniques to improve its performance. First, we calculate the least number of nodes that is required by *SList* during the enumeration. Then, we pre-allocate all the nodes in an object pool, which is implemented using an array, and reuse the nodes through the enumeration procedure. Second, each node of *SList* has a counter that has to be updated and there are $\Delta(P)$ nodes that need to be updated in each iteration. We replace the counter with a timestamp, which achieves the same functionality but only needs to be set once and requires no further updates. Hence, the cost of the update is reduced from $\mathcal{O}(\Delta(P))$ time to constant time. From our empirical observations, the implementation enhancements have reduced approximately 50% of the original running time and 90% of the original memory usage. In our experiments, we use the improved Tree algorithm.

## 4.3 Experimental Results

The input of the compared algorithms is the vector clocks of the events in the computation and the output is the set of global states of the computation. Table 2 also shows the experimental results. All the experiments are conducted on a Linux machine with an Intel Xeon 2.67GHz CPU and the heap size of Java virtual machine is limited to 2GB. The runtime is measured in seconds. As it can be seen, BFS algorithm has the worst performance because of its high time complexity. Moreover, it failed to finish on more than half of the benchmarks because it ran out of the available 2GB memory. The reason is that it has to store intermediate global states for future iterations and the number of intermediate global states might grow exponentially in $n$ in the worst case.

We first compare the runtimes of Tree, Lex, and QuickLex in the first and second set of benchmarks. Fig. 5 shows the normalized runtimes of each algorithm with respect to the runtime of Tree. We normalized the runtimes to those of Tree because it has an amortized $\mathcal{O}(1)$ per global state and the smallest theoretical time complexity among the existing

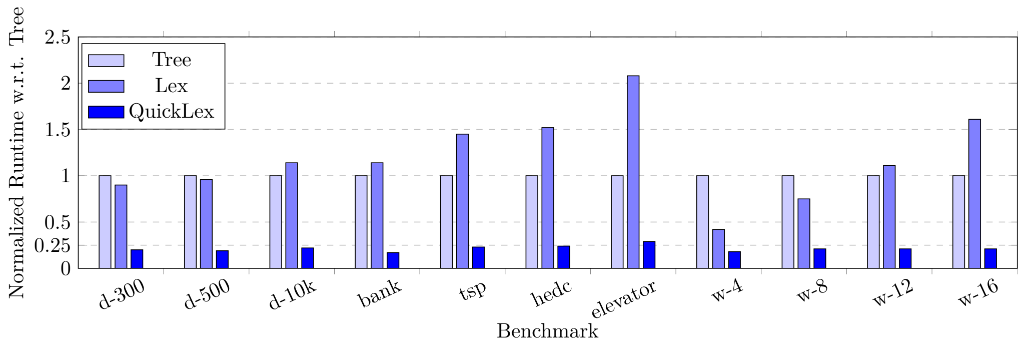

**Figure 5** Normalized runtime of each algorithm w.r.t. the runtime of Tree algorithm.

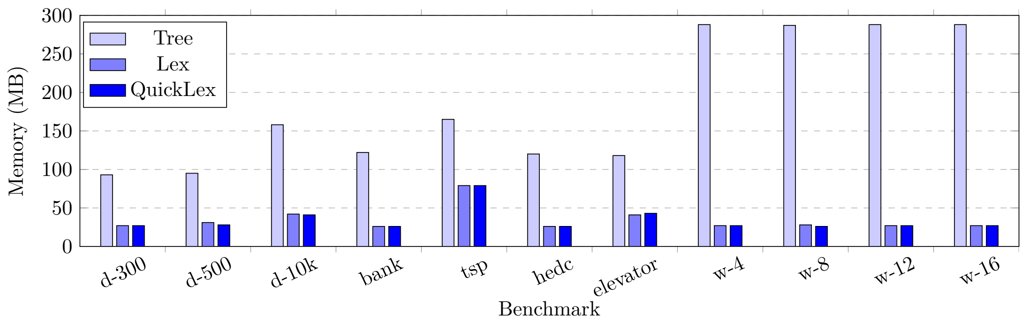

**Figure 6** Memory usage of Tree, Lex, and QuickLex algorithm.

enumeration algorithms. From Fig. 5, QuickLex is approximately 7 times faster than Lex and consistently 4–5 times faster than Tree. One reason that Tree is not as fast as QuickLex is that its intermediate information has to be stored in a linked list and therefore the cost of accessing the information is high.

We now compare the runtimes of Tree, Lex, and QuickLex in the third set of benchmarks; the benchmarks that start with the prefix "$w$-". From Fig. 5, we can see that the normalized runtimes of Lex increase as the number of processes increases. On the other hand, the normalized runtimes of QuickLex are consistently 4 times faster than those of Tree, which shows that the time complexity of QuickLex can achieve amortized $\mathcal{O}(1)$ per global state in practice. The detailed explanation is discussed in Appendix C.

Fig. 6 shows the memory usage of the compared enumeration algorithms. Since Lex is stateless, its memory is mainly used for storing the input, i.e., the computation. From Fig. 6, QuickLex uses almost the same amount of memory even though QuickLex requires additional $\mathcal{O}(n^2)$ space to store the stacks for dynamic programming. The $\mathcal{O}(n^2)$ space is quite small because the space only stores integers. Tree, however, consumes much more memory space than Lex and QuickLex because it needs to store the information regarding its backward spanning tree, whose size is linear to $O(|P|)$. Note that $|P|$ is much larger than $n^2$. Assume that each process in $P$ has at least $m$ events, then $|P|$ is at least as large as $(n \times m)$. Usually, $m$ is much larger than $n$.

**Table 3** The performance of ParaMount with different enumeration algorithms.

| Benchmark | Information | | | Runtime (ms) | | # Detection |
|---|---|---|---|---|---|---|
| | LoC | Thread | #Var | Lex | QuickLex | |
| *banking* | 139 | 4 | 7 | 72 | 20 | 1 |
| *set (faulty)* | 223 | 4 | 10 | 152 | 69 | 1 |
| *set (correct)* | 260 | 4 | 10 | 110 | 51 | 0 |
| *arraylist1* | 1,474 | 4 | 6 | 19 | 19 | 3 |
| *arraylist2* | 1,377 | 4 | 16 | 22 | 15 | 0 |
| *sor* | 255 | 4 | 20 | 81 | 25 | 0 |
| *elevator* | 547 | 4 | 23 | 890 | 667 | 0 |
| *tsp* | 702 | 4 | 36 | 114 | 42 | 1 |
| *raytracer* | 1,885 | 4 | 77 | 1240 | 236 | 1 |
| *hedc* | 25,027 | 8 | 345 | 940 | 335 | 4 |

## 5 The Applications of QuickLex

### 5.1 Predicate Detection in Concurrent Systems

In this section, we compare the performance of Lex and QuickLex in real-world applications. In [4], we implemented a predicate detector, named ParaMount, for concurrent programs. The detector captures the execution trace of users' program using Java bytecode injection [1]. The captured execution trace is converted to a concurrent computation using the methods discussed in [10, 21]. Specifically, the detector captures 1) the read and write operations of all variables, 2) the causal dependency of fork-and-join operations of thread, and 3) the causal dependency of the acquisition-and-release operations of locks (including implicit locks and monitors) in users' program. The causal dependency is represented by HB relation in the computation.

Afterwards, ParaMount uses a sequential enumeration algorithm (e.g., Lex or QuickLex) as the subroutine to enumerate the set of global states in an online-and-parallel fashion. During the enumeration, each global state is checked for the predicate corresponding to data races. A data race occurs when conflicting operations (i.e., a pair of read-write or write-write operations) are concurrently executed on the same memory address by different threads. In summary, the detector takes as input a program and outputs the variables that have data races.

Table 3 shows the result of the detection. "LoC" shows the lines of code of the benchmark program. "Thread" shows the number of threads that are used to drive each benchmark. "#Var" shows the number of variables of the benchmark. Every variable is checked if it is accessed by different threads without the protection of any lock. Besides the four real-world applications that are used in Section 4, we also use the following applications. The benchmarks *set (faulty)* and *set (correct)* are incorrect and correct implementations of the concurrent set [16]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *sor* is a scientific computation application; and *raytracer* is a benchmark for measuring the performance of a 3D raytracer. The benchmarks *sor*, *elevator*, *tsp*, *raytracer*, and *hedc* are also used in [5, 10, 30].

The running time of ParaMount includes the time to inject bytecode for monitoring, to execute the benchmark program, to capture the executed events, to enumerate global states, and to evaluate the predicate of data races. The column "Lex" shows the original execution time of ParaMount using the Lex as its subroutine and column "QuickLex" shows the improved execution time. In average, QuickLex improves the execution time of ParaMount by a factor of 3. "#Detection" shows the number of variables that have data races; all the detected variables are also detected by [5, 10], so the results do not have false positives.

## 5.2 Other Applications

In [13, 14], it has been shown that many families of combinatorial objects can be mapped to the lattice of global states of appropriate posets. Thus, lexical traversal that is discussed in this paper can also be used to efficiently enumerate all subsets of $[n]$, all subsets of $[n]$ of size $m$, all permutations, all permutations with a given inversion number, all integer partitions less than a given partition, all integer partitions of a given number, and all $n$-tuples of a product space.

## 6 Conclusion

In this paper, we presented a fast algorithm, named QuickLex, for global states enumeration of concurrent and distributed computations. In comparison with the original lexical algorithm, QuickLex has a preprocessing procedure and incorporates dynamic programming to reduce the time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot \Delta(P))$. In the evaluation section, we implemented and compared QuickLex with several existing enumeration algorithms, i.e., BFS [6, 12], Lex [11, 12], and Tree [19, 15]. Moreover, these algorithms are enhanced with different techniques. From our experimental results, QuickLex is 7 times faster than Lex and 4–5 times faster than Tree. The experiments also show that QuickLex can achieve amortized constant time for a certain type of computations. QuickLex uses almost the same amount of memory as Lex while Tree requires 2–10 times more memory than QuickLex. For the real-world applications, QuickLex is used to implement an online-and-parallel predicate detector for concurrent programs. The experimental results show that the detector speeds up 3 times in comparison with its previous version (which uses Lex as its subroutine).

## References

1 ASM – a java bytecode engineering library.

2 Sridhar Alagar and Subbarayan Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering*, 27:412–417, 2001.

3 K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

4 Yen-Jung Chang and Vijay K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015.

5 Feng Chen, Traian Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

6 R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

7 B. A. Davey and H. A. Priestley. Introduction to lattices and order. In *Cambridge University Press*, Cambridge, UK, 1990.

8 E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

9 Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, 1988.

10 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of ACM SIGPLAN the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

11 Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

12 Vijay K. Garg. Enumerating global states of a distributed computation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 134–139, 2003.

13 Vijay K. Garg. Algorithmic combinatorics based on slicing posets. *Theoretical Computer Science*, 359(1-3):200–213, 2006.

14 Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications.* John Wiley & Sons, Inc., 2015.

15 Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Applied Mathematics*, 110(2-3):169–187, 2001.

16 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

17 Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

18 M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24:664 – 677, 1998.

19 Roland Jegou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In *Proceedings of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.

20 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

21 Y. Lei and R.H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

22 Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

23 Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 125–226, Chateau de Bonas, France, 1988.

24 Vinit A. Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of International Symposium in Distributed Computing*, pages 420–434, 2007.

25 Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

26 Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Springer LNCS Order*, 10:239–252, 1993.

27 Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

28 Matthew B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

29 George Steiner. An algorithm to generate the ideals of a partial order. *Operations Research Letters*, 5(6):317–320, 1986.

30 Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

## A The Symbols Used in this Paper

**Table 4** Symbols used in this paper.

| Symbol | Meaning or exchangeable terms |
|---|---|
| $G$ | consistent global state, ideal. |
| $P$ | poset of events, computation. |
| $n$ | #processes in $P$, width of $P$. |
| $i(P)$ | #global states in $P$, #ideals of $P$. |
| $\|P\|$ | #events in $P$, size of $P$. |
| $\Delta(P)$ | #remote events or maximal in-degree of any event. |
| $e, f$ | arbitrary events in $P$ unless specified otherwise. |
| $p_i$ | a process whose process ID is $i$. |
| $e_i$ | an arbitrary event that occurs on process $p_i$. |
| $G[i]$ | the maximal event of process $p_i$ w.r.t. global state $G$. |
| $P_h, P_l$ | the set of processes with higher and lower priority, respectively. |
| $p_h, p_l$ | an arbitrary process in $P_m$ and $P_l$, respectively. |
| $G_m[l]$ | the maximum dependency event of process $p_l$ w.r.t. global state $G$. |
| $X_l(i)$ | the max function $\max_{1\leq j\leq i}(G[j].vc[l])$. |

## B Proof of Theorem 1

**Algorithm 7** Calculate vector clock for event $e$

```
Input: Event $e$ occurs on process $p_i$, predecessor $d$ of $e$, and the set $R(e)$ of remote events of $e$.
Output: The vector clock for event $e$.
1: function calVectorClock($e, d, R(e)$)
2:     $e.vc = d.vc$
3:     $e.vc[i] = d.vc[i] + 1$
4:     for $r \in R(e)$ do                          ▷ For every remote event $r$ of $e$
5:         for $j$ from 1 to $n$ do
6:             $e.vc[j] = max(e.vc[j], r.vc[j])$
7:         end for
8:     end for
9: end function
```

For every event $e$, which occurs on process $p_i$, its vector clock $e.vc$ is calculated using Algorithm 7. For example, the vector clock of event $e6$ in Fig. 2b is set to $[0, 0, 2]$ because of its predecessor $e5$. Then, its vector clock is set to $[0, 2, 2]$ due to its remote event $e3$. Now we show the proof of Theorem 1:

**Proof.** ($\Rightarrow$): From Definition 1.

($\Leftarrow$): The proof is shown by the information of vector clocks. Assume that the predecessor $d$ of $e$ is included in $G$, we get $(e.vc[i] = G[i] + 1)$.

Since $d$ is included in $G$, we also get $\forall j : 1 \leq j \leq n : d.vc[j] \leq G[j]$ due to the property of vector clocks. Assume that all remote events of $e$ are also included in $G$, we get $\forall r \in R(e) : (\forall j : 1 \leq j \leq n : r.vc[j] \leq G[j])$. From the calculation of vector clocks in Algorithm 7, we get $(\forall j \neq i : e.vc[j] \leq G[j])$. As a result, $e$ is enabled when its predecessor and all remote events are included in $G$. ◀

## C The Amortized Constant Time Enumeration of QuickLex

We now explain how QuickLex achieves amortized $\mathcal{O}(1)$ time per global state in practice. Suppose that any event in the computation can have at most one remote event, then the

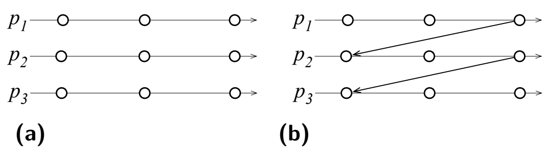

**Figure 7** (a) The best case for QuickLex. (b) The worst case for QuickLex.

worst time complexity of PROPAGATE is $\mathcal{O}(n)$ per global state. Recall that each call of PROPAGATE runs through $(n-k+1)$ processes before returning $k$. If there exist more than $(n-k+1)$ global states between current and most recent PROPAGATE call that returns the same $k$, then the cost of current PROPAGATE call can be charged to the iterations between these two PROPAGATE calls, which cumulatively enumerated $(n-k+1)$ global states. Thus, the current PROPAGATE call is amortized to $\mathcal{O}(1)$.

Fig. 7a illustrates the explanation. Assume that the cost of a PROPAGATE call is $c$ if the *while* loop of PROPAGATE executes $c$ iterations. For instance, the cost of a PROPAGATE call that returns $k = 2$ is 2. However, QuickLex has enumerated 4 global states, e.g., $[0, 0, 0]$, $[0, 0, 1]$, $[0, 0, 2]$, and $[0, 0, 3]$, between any two PROPAGATE calls that return $k = 2$. Consequently, the additional cost of the current PROPAGATE call, which returns $k = 2$, can be evenly charged to 5 global states, including the current one. Similarly, there are 17 global states for any PROPAGATE call that returns $k = 1$ to share the additional cost. As a result, the time complexity of any PROPAGATE call can be amortized to $\mathcal{O}(1)$ time per global state. The same reason holds for the time complexity of RESET.

Fig. 7b shows the worst case for QuickLex, in which only one global state exists between PROPAGATE calls. Therefore, the cost cannot be amortized and hence PROPAGATE takes $\mathcal{O}(n)$ time. The events in this computation are totally ordered, which is not a common computation.