Space Efficient Breadth-First and Level Traversals of Consistent Global States of Parallel Programs

Himanshu Chauhan(✉) and Vijay K. Garg

University of Texas at Austin, Austin, USA
himanshu@utexas.edu, garg@ece.utexas.edu

Abstract. Enumerating *consistent* global states of a computation is a fundamental problem in parallel computing with applications to debugging, testing and runtime verification of parallel programs. Breadth-first search (BFS) enumeration is especially useful for these applications as it finds an erroneous consistent global state with the least number of events possible. The total number of executed events in a global state is called its *rank*. BFS also allows enumeration of all global states of a given rank or within a range of ranks. If a computation on n processes has m events per process on average, then the traditional BFS (Cooper-Marzullo and its variants) requires $\mathcal{O}(\frac{m^{n-1}}{n})$ space in the worst case, whereas our algorithm performs the BFS requires $\mathcal{O}(m^2n^2)$ space. Thus, we reduce the space complexity for BFS enumeration of consistent global states exponentially, and give the first polynomial space algorithm for this task. In our experimental evaluation of seven benchmarks, traditional BFS fails in many cases by exhausting the 2 GB heap space allowed to the JVM. In contrast, our implementation uses less than 60 MB memory and is also faster in many cases.

1 Introduction

Parallel programs are not only difficult to design and implement, but once implemented are also difficult to debug and verify. The technique of predicate detection [12,17] is helpful in verification of these implementations as it allows inference based analysis to check many possible system states based on one execution trace. The technique involves execution of the program, and modeling of its trace as a partial order. Then all possible states of the model that are consistent with the partial order are visited and evaluated for violation of any constraints/invariants. A large body of work uses this approach to verify distributed applications, as well as to detect data-races and other concurrency related bugs in shared memory parallel programs [11,14,19,23]. Finding consistent global states of an execution also has critical applications in snapshotting of modern distributed file systems [1,27].

A fundamental requirement for this approach is the traversal of all possible consistent global states, or *consistent cuts*, of a parallel execution. Let us call the execution of a parallel program a *computation*. The set of all consistent cuts

S. Lahiri and G. Reger (Eds.): RV 2017, LNCS 10548, pp. 138–154, 2017.
DOI: 10.1007/978-3-319-67531-2_9

of a computation can be represented as a directed acyclic graph in which each vertex represents a consistent cut, and the edges mark the transition from one global state to another by executing one operation. Moreover, this graph has a special structure: it is a distributive lattice [24]. Multiple algorithms have been proposed to traverse the lattice of consistent cuts of a parallel execution. Cooper and Marzullo's algorithm [12] starts from the source — a consistent cut in which no operation has been executed by any process — and performs a breadth-first-search (BFS) visiting the lattice level by level. Alagar and Venkatesan's algorithm [2] performs a depth-first-search (DFS) traversal of the lattice, and Ganter's algorithm [15] enumerates global states in lexical order.

The BFS traversal of the lattice is particularly useful in solving two key problems. First, suppose a programmer is debugging a parallel program to find a concurrency related bug. The global state in which this bug occurs is a counter-example to the programmer's understanding of a correct execution, and we want to halt the execution of the program on reaching the first state where the bug occurs. Naturally, finding a small counter example is quite useful in such cases. The second problem is to check all consistent cuts of given rank(s). For example, a programmer may observe that her program crashes only after k events have been executed, or while debugging an implementation of Paxos [22] algorithm, she might only be interested in analyzing the system when all processes have sent their *promises* to the leader. Among the existing traversal algorithms, the BFS algorithm provides a straightforward solution to these two problems. It is guaranteed to traverse the lattice of consistent cuts in a level by level manner where each level corresponds to the total number of events executed in the computation. This traversal, however, requires space proportional to the size of the biggest level of the lattice which, in general, is *exponential* in the size of the computation. In this paper, we present a new algorithm to perform BFS traversal of the lattice in space that is polynomial in the size of the computation. In short, the contribution of this paper are:

- For a computation on n processes such that each process has m events on average, our algorithm requires $\mathcal{O}(m^2n^2)$ space in the worst case, whereas the traditional BFS algorithm requires $\mathcal{O}(\frac{m^{n-1}}{n})$ space (exponential in n).
- Our evaluation on seven benchmark computations shows the traditional BFS runs out of the maximum allowed 2 GB memory for three of them, whereas our implementation can traverse the lattices by using less than 60 MB memory for each benchmark.

The exponential reduction in space may come at the cost of a longer runtime to perform the BFS traversal. In the worst case, our algorithm may take $\mathcal{O}(m^2n^2)$ time per consistent cut. However, our experimental evaluation shows our runtimes are within the same order of magnitude to those of the traditional BFS.

2 Background

We model a computation $P = (E, \rightarrow)$ on n processes $\{P_1, P_2, \ldots, P_n\}$ as a partial order on the set of events, E. The events are ordered by Lamport's *happened-*

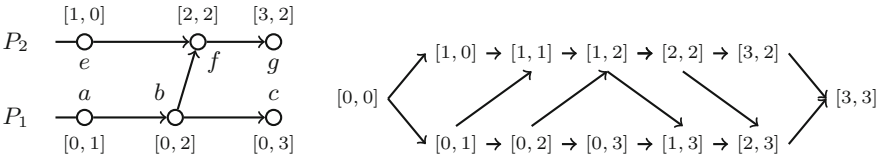
before ($\rightarrow$) relation [21]. This partially ordered set (poset) of events is partitioned into chains:

Definition 1 (Chain Partition). *A chain partition of a poset places every element of the poset on a chain that is totally ordered. Formally, if α is a chain partition of poset $P = (E, \rightarrow)$ then α maps every event to a natural number such that*

$$\forall x, y \in E : \alpha(x) = \alpha(y) \Rightarrow (x \rightarrow y) \vee (y \rightarrow x).$$

Generally, a computation on n processes is partitioned into n chains such that the events executed by process P_i $(1 \leq i \leq n)$ are placed on i^{th} chain.

Mattern [24] and Fidge [13] proposed *vector clocks*, an approach for time-stamping events in a computation such that the happened-before relation can be tracked. For a program on n processes, each event's vector clock is a n-length vector of integers. Note that vector clocks are dependent on chain partition of the poset that models the computation. For an event e, we denote $e.V$ as its vector clock. Throughout this paper, we use the following representation for interpreting chain partitions and vector clocks: if there are n chains in the chain partition of the computation, then the lowest chain (process) is always numbered 1, and the highest chain being numbered n. A vector clock on n chains is represented as a n-length vector: $[c_n, c_{n-1}, ..., c_i, ..., c_2, c_1]$ such that c_i denotes the number of events executed on process P_i. Hence, if event e was executed on process P_i, then $e.V[i]$ is e's index (starting from 1) on P_i. Also, for any event f in the computation: $e \rightarrow f \Leftrightarrow \forall j : e.V[j] \leq f.V[j] \wedge \exists k : e.V[k] < f.V[k]$. A pair of events, e and f, is concurrent iff $e \not\rightarrow f \wedge f \not\rightarrow e$. We denote this relation by $e||f$. Figure 1a shows a sample computation with six events and their corresponding vector clocks. Event b is the second event on process P_1, and its vector clock is $[0, 2]$. Event g is the third event on P_2, but it is preceded by f, which in turn is causally dependent on b on P_1, and thus the vector clock of g is $[3, 2]$.



(a) Computation (b) Lattice of consistent cuts

Fig. 1. A computation with vector clocks of events, and its consistent cuts

Definition 2 (Consistent Cut). *Given a computation $(E, \rightarrow)$, a subset of events $C \subseteq E$ forms a* consistent cut *if C contains an event e only if it contains all events that happened-before e. Formally, $(e \in C) \wedge (f \rightarrow e) \implies (f \in C)$.*

A consistent cut captures the notion of a possible global state of the system at some point during its execution [6]. Consider the computation shown in Fig. 1a. The subset of events $\{a, b, e\}$ is a consistent cut, whereas $\{a, e, f\}$ is not; because $b \rightarrow f$ (b happened-before f) but b is not included in the subset.

Vector Clock Notation of Cuts: So far we have described how vector clocks can be used to time-stamp events in the computation. We also use them to represent cuts of the computation. If the computation is partitioned into n chains, then for any cut G, its vector clock is a n-length vector such that $G[i]$ denotes the number of events from P_i included in G. Note that in our vector clock representation the events from P_i are at the i^{th} index from the right.

For example, consider the state of the computation in Fig. 1a when P_1 has executed events a and b, and P_2 has only executed event e. The consistent cut for this state, $\{a, b, e\}$, is represented by $[1, 2]$. Note that cut $[2, 1]$ is not consistent, as it indicates execution of f on P_2 without b being executed on P_1. The computation in Fig. 1a has twelve consistent cuts; and the lattice of these consistent cuts (in their vector clock representation) is shown in Fig. 1b.

Rank of a Cut: Given a cut G, we define $rank(G) = \sum G[i]$. The rank of a cut corresponds to the total number of events, across all processes, that have been executed to reach the cut.

In Fig. 1b, there is one source cut ($[0, 0]$) with rank 0, then there are two cuts each of ranks 1 to 5, and finally there is one cut ($[3, 3]$) has rank 6.

2.1 Breadth-First Traversal of Lattice of Consistent Cuts

Consider a parallel computation $P = (E, \rightarrow)$. The lattice of consistent cuts, $\mathcal{C}(E)$, of P is a DAG whose vertices are the consistent cuts of $(E, \rightarrow)$, and there is a directed edge from vertex u to vertex v if state represented by v can be reached by executing one event on u; hence we also have $rank(v) = rank(u) + 1$. The source of $\mathcal{C}(E)$ is the empty set: a consistent cut in which no events have been executed on any process. The sink of this DAG is E: the consistent cut in which all the events of the computation have been executed. Breadth-first search (BFS) of this lattice starts from the source vertex and visits all the cuts of rank 1; it then visits all the cuts of rank 2 and continues in this manner till reaching the last consistent cut of rank $|E|$. For example, in Fig. 1b the BFS algorithm will traverse cuts in the following order: $[0, 0], [0, 1], [1, 0], [0, 2], [1, 1], [0, 3], [1, 2], [1, 3], [2, 2], [2, 3], [3, 2], [3, 3]$.

The standard BFS on a graph needs to store the vertices at distance d from the source to be able to visit the vertices at distance $d + 1$ (from the source). Hence, in performing a BFS on $\mathcal{C}(E)$ we are required to store the cuts of rank r in order to visit the cuts of rank $r + 1$. Observe that in a parallel computation there may be exponentially many cuts of rank r. Thus, traversing the lattice $\mathcal{C}(E)$ requires space which is exponential in the size of input. The optimized vector clock based BFS traversal takes $\mathcal{O}(n^2)$ time per cut [16], where n is the number of processes in the computation.

2.2 Related Work

Cooper and Marzullo [12] gave the first algorithm for global states enumeration which is based on breadth first search (BFS). Let $i(P)$ denote the total number of consistent cuts of a poset P. Cooper-Marzullo algorithm requires $\mathcal{O}(n^2 \cdot i(P))$ time, and exponential space in the size of the input computation. The exponential space requirement is due to the standard BFS approach in which consistent cuts of rank r must be stored to traverse the cuts of rank $r+1$.

There is also a body of work on enumeration of consistent cuts in order different than BFS. Alagar and Venkatesan [3] presented a depth first algorithm using the notion of global interval which reduces the space complexity to $\mathcal{O}(|E|)$. Steiner [29] gave an algorithm that uses $\mathcal{O}(|E| \cdot i(P))$ time, and Squire [28] further improved the computation time to $\mathcal{O}(log|E| \cdot i(P))$. Pruesse and Ruskey [26] gave the first algorithm that generates global states in a combinatorial Gray code manner. The algorithm uses $\mathcal{O}(|E| \cdot i(P))$ time and can be reduced to $\mathcal{O}(\Delta(P) \cdot i(P))$ time, where $\Delta(P)$ is the in-degree of an event; however, the space grows exponentially in $|E|$. Later, Jegou et al. [20] and Habib et al. [18] improved the space complexity to $\mathcal{O}(n \cdot |E|)$.

Ganter [15] presented an algorithm, which uses the notion of lexical order, and Garg [16] gave the implementation using vector clocks. The lexical algorithm requires $\mathcal{O}(n^2 \cdot i(P))$ time but the algorithm itself is *stateless* and hence requires no additional space besides the poset. Paramount [8] gave a parallel algorithm to traverse this lattice in lexical order, and QuickLex [7] provides an improved implementation for lexical traversal that takes $\mathcal{O}(n \cdot \Delta(P) \cdot i(P))$ time, and $\mathcal{O}(n^2)$ space overall.

3 Uniflow Chain Partition

A uniflow partition of a computation's poset $P = (E, \rightarrow)$ is its partition into n_u chains $\{P_i \mid 1 \leq i \leq n_u\}$ such that no element (event of E) in a higher numbered chain is smaller than any element in lower numbered chain; that is if any event e is placed on a chain i then all causal dependencies of e must be placed on chains numbered lower than i. For poset $P = (E, \rightarrow)$, chain partition μ is uniflow if

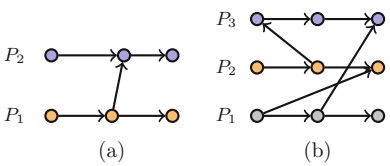


Fig. 2. Posets in uniflow partitions

$$\forall x, y \in P : \mu(x) < \mu(y) \Rightarrow \neg(y \not\rightarrow x) \tag{1}$$

Visually, in a uniflow chain partition all the edges, capturing happened-before relation, between separate chains always point upwards because their dependencies — elements of poset that are smaller — are always placed on lower chains. Figure 2 shows two posets with uniflow partition. Whereas Fig. 3 shows two posets with partitions that do not satisfy the uniflow property. The poset in

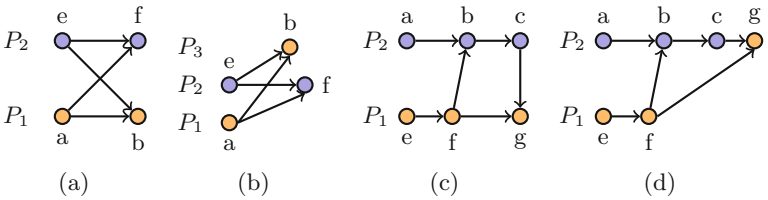


Fig. 3. Posets in (a) and (c) are not in uniflow partition: but (b) and (d) respectively are their uniflow partitions

Fig. 3(a) can be transformed into a uniflow partition of three chains as shown in Fig. 3(b). Similarly, Fig. 3(c) can be transformed into a uniflow partition of two chains shown in Fig. 3(d). Observe that:

Lemma 1. *Every poset has at least one uniflow chain partition.*

Proof. Any total order derived from the poset is a uniflow chain partition in which each element is a chain by itself. In this trivial uniflow chain partition the number of chains is equal to the number of elements in the poset.

The structure of uniflow chain partitions can be used for efficiently obtaining consistent cuts of larger ranks.

Lemma 2 (Uniflow Cuts Lemma). *Let P be a poset with a uniflow chain partition $\{P_i \mid 1 \leq i \leq n_u\}$, and G be a consistent cut of P. Then any $H_k \subseteq P$ for $1 \leq k \leq n_u$ is also a consistent cut of P if it satisfies:*

$$\forall i : k < i \leq n_u : H_k[i] = G[i], \text{ and}$$
$$\forall i : 1 \leq i \leq k : H_k[i] = |P_i|.$$

Proof. Using Eq. 1, we exploit the structure of uniflow chain partitions: the causal dependencies of any element e lie only on chains that are lower than e's chain. As G is consistent, and H_k contains the same elements as G for the top $n_u - k$ chains, all the causal dependencies that need to be satisfied to make H_k have to be on chain k or lower. Hence, including all the elements from all of the lower chains will naturally satisfy all the causal dependencies, and make H_k consistent.

For example, in Fig. 2(b), consider the cut $G = [1, 2, 1]$ that is a consistent cut of the poset. Then, picking $k = 1$, and using Lemma 2 gives us the cut $[1, 2, 3]$ which is consistent; similarly choosing $k = 2$ gives us $[1, 3, 3]$ that is also consistent. Note that the claim may not hold if the chain partition does not have uniflow property. For example, in Fig. 3(c), $G = [2, 2]$ is a consistent cut. The chain partition, however, is not uniflow and thus applying the Lemma with $k = 1$ gives us $[2, 3]$ which is not a consistent cut as it includes the third event on P_1, but not its causal dependency — the third event on P_2.

3.1 Finding a Uniflow Partition

The problem of finding a uniflow chain partition is a direct extension of finding the *jump number* of a poset [5,10,30]. Multiple algorithms have been proposed to find the jump number of a poset; which in turn arrange the poset in a uniflow chain partition. Finding an optimal (smallest number of chains) uniflow chain partition of a poset is a hard problem [5,10]. Bianco et al. [5] present a heuristic algorithm to find a uniflow partition, and show in their experimental evaluation that in most of the cases the resulting partitions are relatively close to optimal. We use a vector clock based online algorithm to find a uniflow partition for a computation. We present this algorithm in the extended version of the paper [9]. Note that we need to re-generate vector clocks of the events for the uniflow partition. This is a simple task using existing vector clock implementation techniques, and we omit these details.

4 Polynomial Space Breadth-First Traversal of Lattice

BFS traversal of the lattice of consistent cuts of any poset can be performed in space that is polynomial in the size of the poset. We do so by first obtaining the poset's uniflow chain partition, and then using this partition for traversal of cuts in increasing order of ranks. We start from the empty cut, and then traverse all consistent cuts of rank 1, then all consistent cuts of rank 2 and so on. For rank r, $1 \leq r \leq |E|$, we traverse the consistent cuts in the following lexical order:

Definition 3 (Lexical Order on Consistent Cuts). *Given any chain partition of poset P that partitions it into n chains, we define a total order called* lexical order *on all consistent cuts of P as follows. Let G and H be any two consistent cuts of P. Then, $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n \geq i > k : G[i] = H[i])$.*

Recall from our vector clock notation (Sect. 2) that the right most entry in the vector clock is for the least significant (lowest) chain. Consider the poset with a non-uniflow chain partition in Fig. 4(a). The vector clocks of its events are shown against the four events. The lexical order on the consistent cuts of this chain partition is: $[0,0] <_l [0,1] <_l [1,0] <_l [1,1] <_l [1,2] <_l [2,1] <_l [2,2]$. For the same poset, Fig. 4(b) shows the equivalent uniflow partition, and the corresponding vector clocks. The lexical order on the consistent cuts for this uniflow chain partition is: $[0,0,0] <_l [0,0,1] <_l [0,1,0] <_l [0,1,1] <_l [0,2,1] <_l [1,1,1] <_l [1,2,1]$. Note that the number of consistent cuts remains same for both of these chain partitions, and there is a one-to-one mapping between the consistent cuts in the two partitions. Hence, if the computation's uniflow partition is different from its original chain

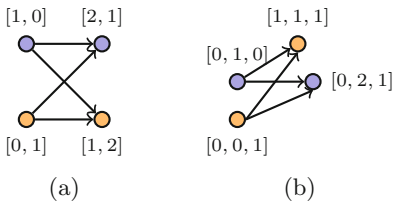


Fig. 4. Vector clocks of a computation in its original form, and in its uniflow partition

Algorithm 1. TraverseBFSUniflow(P)

Input: A poset $P = (E, \rightarrow)$ that has been partitioned into a uniflow chain partition of n_u chains, and the vector clock of the events have been regenerated for this partition.

```
1: G = new int[n_u]  // initial consistent cut
2: enumerate(G)  // evaluate the predicate on empty cut G.
3: for (r = 1; r ≤ |E|; r++) do
4:    //make G lexically smallest cut of given rank
5:    G = GetMinCut(G, r)
6:    while G ≠ null do
7:       enumerate(G)  // evaluate the predicate on G.
8:       //find the next bigger lexical cut of same rank
9:       G = GetSuccessor(G, r)
```

partition, we re-map the consistent cuts in uniflow partition to cuts in original partition.

Algorithm 1 shows the steps of our BFS traversal using a computation in a uniflow chain partition. From Lemma 1, we know that every poset has a uniflow chain partition. Recall that the vector clocks of the events depend on the chain partition of the poset. Thus, in generating this input we need two pre-processing steps: (a) finding a uniflow partition, and (b) regenerating vector clocks for this partition. For example, given a computation on two processes shown in Fig. 4(a), we will first convert it to the computation shown in Fig. 4(b). These steps are performed only once for a computation, and are relatively inexpensive in comparison to the traversal of lattice.

For each rank r, $1 \leq r \leq |E|$, Algorithm 1 first finds the lexically smallest consistent cut at of rank r. This is done by the GetMinCut (shown in Algorithm 2) routine that returns the lexically smallest consistent cut of P bigger than G of rank r. For example, in Fig. 5, GetMinCut$([0,0,0],4)$ returns $[0,1,3]$. Given a consistent cut G of rank r, we repeatedly find the next lexically bigger consistent cut of rank r using the routine GetSuccessor given in Algorithm 3. For example, in Fig. 5, GetSuccessor$([0,0,3],3)$ returns the next lexically smallest consistent cut $[0,1,2]$.

The GetMinCut routine on poset P assumes that the rank of G is at most r and that G is a consistent cut of the P. It first computes d as the difference between r and the rank of G. We need to add d elements to G to find the smallest consistent cut of rank r. We exploit the Uniflow Cut Lemma (Lemma 2) by adding as many elements from the lowest chain as possible. If all the elements from the lowest chain are already in G, then we continue with the second lowest chain, and so on. For example in Fig. 5, consider finding smallest consistent cut of rank 5 starting from $G = [0,0,2]$. In this case, we add all three elements from P_1 to reach $[0,0,3]$, and then add first two elements from P_2 to get the answer as $[0,2,3]$.

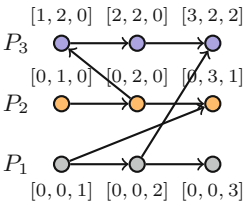
The GetSuccessor routine (Algorithm 3) finds the lexical successor of G at rank r. The approach for finding a lexical successor is similar to counting

numbers in a decimal system: if we are looking for successor of 2199, then we cannot increment the two 9 s (as we are only allowed digits 0–9), and hence the first possible increment is for entry 1. We increment it to 2, but we must now reset the entries at lesser significant digits. Hence, we reset the two 9 s to 0 s, and get the successor as 2200.

Algorithm 2. GetMinCut(G, r)

Input: G: a consistent cut of poset P from Algorithm 1
Output: Smallest consistent cut of rank r that is lexically greater than or equal to G.

1: $d = r - rank(G)$ // difference in ranks
2: **for** $(j = 1; j \leq n_u; j = j + 1)$ **do**
3: **if** $d \leq |P_j| - G[j]$ **then**
4: $G[j] = G[j] + d$
5: **return** G
6: **else** // take all the elements from chain j
7: $G[j] = G[j] + |P_j|$
8: $d = d - |P_j|$



Algorithm 3. GetSuccessor(G, r)

Input: G: a consistent cut of rank r
Output: K: lexical successor of G of rank r

1: $K = G$ // Create a copy of G in K
2: **for** $(i = 2; i \leq n_u; i{+}{+})$ **do** // lower chains to higher
3: **if** next element on P_i exists **then**
4: $K[i] = K[i] + 1$ // increment cut
5: **for** $(j = i - 1; j > 0; j - -)$ **do**
6: $K[j] = 0$ // reset lower chains
7: //fix dependencies on lower chains
8: **for** $(j = i + 1; j \leq n_u; j + +)$ **do**
9: **for** $(k = i - 1; k > 0; k - -)$ **do**
10: $vc =$ vector clock of event
11: number $G[j]$ on P_j
12: $K[k] = \text{MAX}(vc[k], K[k])$
13: **if** $rank(K) \leq r$ **then**
14: **return** GetMinCut(K, r)
15: **return** null // no candidate cut

Fig. 5. Illustration: GetSuccessor

In our GetSuccessor routine, we start at the second lowest chain in a uniflow poset, and if possible increment the cut by one event on this chain. We then reset the entries on lower chains, and then make the cut consistent by satisfying all the causal dependencies. If the rank of the resulting cut is less than or equal to r, then calling the GetMinCut routine gives us the lexical successor of G at rank r. Line 1 copies cut G in K. The for loop covering lines 2–13 searches for an appropriate element not in G such that adding this element makes the resulting consistent cut lexically greater than G. We start the search from chain 2, instead of chain 1, because for a non-empty cut G adding any event from the lowest chain to G will only increase G's rank as there are no lower chains to reset. Line 3 checks if there is any possible element to add in P_i. If yes, then lines 4–6 increment K at chain i, and then set all its values for lower chains to 0. To ensure that K is a consistent cut, for every element in K, we add its causal dependencies to K in lines 7–11. Line 12 checks whether the resulting consistent cut is of rank $\leq r$. If $rank(K)$ is at most r, then we have

found a suitable cut that can be used to find the next lexically bigger consistent cut and we call GetMinCut routine to find it. If we have tried all values of i and did not find a suitable cut, then G is the largest consistent cut of rank r and we return null.

In Fig. 5, consider the call of GetSuccessor $([1,2,3],6)$. As there is no next element in P_1, we consider the next element in P_2. After line 5, the value of K is $[1,3,0]$, which is not consistent. Lines 7–10 make K a consistent cut, now $K = [1,3,1]$. Since $rank(K)$ is 5, we call GetMinCut at line 13 to find the smallest consistent cut of rank 6 that is lexically bigger than $[1,3,1]$. This consistent cut is $[1,3,2]$.

The proof of correctness is given in the extended version of the paper [9].

4.1 Optimization for Time Complexity

We can find the lexical successor of any consistent cut in $\mathcal{O}(n_u^2)$ time, instead of $\mathcal{O}(n_u^3)$ time taken in GetSuccessor, by using additional $\mathcal{O}(n_u^2)$ space.

Observe that GetSuccessor routine iterates over $n_u - 1$ chains in the outer loop at line 2, and the two inner loops at lines 8 and 9 perform $\mathcal{O}(n_u^2)$ work in the worst case. When we cannot find a suitable cut of rank less than or equal to r (check performed at line 12), we move to a higher chain (with the outer loop at line 2). Thus, we repeat a large fraction of the $\mathcal{O}(n_u^2)$ work in the two inner loops at lines 8 and 9 for this higher chain. We can avoid this repetition by storing the combined causal dependencies from higher chains on each lower chain.

Algorithm 4. ComputeProjections(G)

Input: G: a consistent cut of rank r

```
1: for (i = n_u; i ≥ 1; i−−) do // go top to bottom
2:    val = G[i]  // event number in G on chain i
3:    vc = vector clock of event num val on chain i
4:    if i == n_u then  // on highest chain
5:       proj[i] = vc
6:    else // process relevant entries in vector
7:       for (j = i; j > 0; j − −) do
8:          //projection on chain i:
9:          proj[i][j] =MAX(vc[j], proj[i + 1][j])
```

Let us illustrate this with an example. Consider the uniflow computation shown in Fig. 6. Suppose we want the lexical successor of $G = [1,3,2]$. Then, for each chain, starting from the top we compute the projection of events included in G on lower chains. For example, $G[3] = 1$, and thus on the top-most chain, the projection is only the vector clock of the first event on P_3, which is $[1,0,0]$. Thus $proj[3] = [1,0,0]$. On P_2, the projection must include the combined vector clocks of $G[3]$ and $G[2]$ — the events from top two chains. As $G[2] = 3$, we use the vector clock of third event on P_2, which

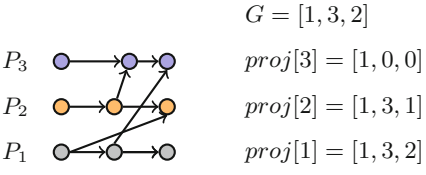


Fig. 6. Projections of a cut on chains

is $[0, 3, 1]$ as that event is causally dependent on first event on P_1. Combining the two vectors gives us the projection on P_2 as $proj[2] = [1, 3, 1]$.

Algorithm 4 shows the steps involved in computing the projections of a cut on each chain. We create an auxiliary matrix, $proj$, of size $n_u \times n_u$, to store these projections. In GetSuccessor routine, once we have computed a new successor by using some event on chain i, we need to update the stored projections on chains lower than i; and not all n_u chains. This is because the projections for unchanged entries in G above chain i will not change on chain i, or any chain above it. Hence, we only update the relevant rows and columns — rows and columns with number i or lower — in $proj$; i.e. only the upper triangular part of the matrix $proj$. We keep track of the chain that gave us the successor cut, and pass it as an additional argument to Algorithm 4. We read and update $n_u^2/2$ entries in the matrix, and not all n_u^2 of them.

Hence, the optimized implementation of finding the lexical successor of G requires two changes. First, every call of GetSuccessor (G, r) starts with first computing the projections of G using Algorithm 4. Second, we replace the two inner for loops at lines 8 and 9 in GetSuccessor by one $\mathcal{O}(n_u)$ loop to compute the max of the two vector clocks: vector clock of $K[i]$, and $proj[i]$. See the extended version of the paper for details [9].

4.2 Re-mapping Consistent Cuts to Original Chain Partition

The number of consistent cuts of a computation is independent of the chain partition used. Their vector clock representation, however, varies with chain partitions as the vector clocks of events in the computation depend on the chain partition used to compute them. There is a one-to-one mapping between a consistent cut in the original chain partition of the computation on n chains (processes), and its uniflow chain partition on n_u chains. We now show how to map a consistent cut in a uniflow chain partition to its equivalent cut in the original chain partition of the computation. Let $P = (E, \rightarrow)$ be a computation on n processes, and let n_u be the number of chains in its uniflow chain partition. If G_u is a consistent cut in the uniflow chain partition, then its equivalent consistent cut G for the original chain partition (of n chains) can be found in $\mathcal{O}(n_u + n^2)$ time.

We do so by mapping two additional entries with the new vector clock of each event for uniflow chain partition: the chain number c, and event number e from the original chain partition over n chains. For example, in Fig. 4(b), for uniflow vector clock $[1, 1, 1]$, its chain number in original poset is 1, and its event number on that chain is 2. When generating the uniflow vector clocks, we populate these entries in a map. Given a uniflow vector clock uvc, the call to OriginalChain(uvc) returns c, and OriginalEvent(uvc) returns e. To compute G from G_u, we use these two values from the corresponding event for each entry in G_u. We start with I as an all-zero vector of length n. Now, we iterate over G_u, and we update I by setting $I[c] = max(I[c], e)$. As vector G_u has length n_u, this step takes $\mathcal{O}(n_u)$ time. We now initiate G as an all-zero vector clock of length n, and for each entry $I[k]$, $1 \leq k \leq n$, we get the vector clock, vce, of event $I[k]$ on chain k in the original computation. We then set G

to the component-wise maximum of G and vce. As there are n entries in I, and for each non-zero entry we perform $\mathcal{O}(n)$ work in updating G (in lines 11–14 in Algorithm 5) the total work in this step is $\mathcal{O}(n^2)$.

Algorithm 5. REMAP(G_u, n_u, n)

Input: G_u: a consistent cut in uniflow chain partition on n_u chains
Output: G: equivalent consistent cut in original chain partition on n chains

```
1:  G = new int[n]  // allocate memory for G
2:  I = new int[n_u]  // reduction vector
3:  for (i = n_u; i ≥ 1; i − −) do  // go over all the uniflow chains
4:    uvc =event number G_u[i]'s vector-clock on uniflow chain i
5:    //chain of this event in original poset
6:    c = OriginalChain(uvc)
7:    //uvc's event number on chain c in original poset
8:    e = OriginalEvent(uvc)
9:    if I[c] < e then  // update indicator with e
10:     I[c] = e
11: for (j = n; i ≥ 1; i − −) do  // go over chains in original poset
12:   vce =event number I[j]'s vector-clock on chain j in original poset
13:   for (k = n; k ≥ 1; k − −) do  // update G entries
14:     G[k] =MAX(G[k], vce[k])
15: return G
```

4.3 Traversing Consistent Cuts of a Given Rank

A key benefit of our algorithm is that it can traverse all the consistent cuts of a given rank, or within a range of ranks, without traversing the cuts of lower ranks. In contrast, the traditional BFS traversal must traverse, and store, consistent cuts of rank $R-1$ to traverse cuts of rank R, which in turn requires it to traverse cuts of rank $R-2$ and so on.

To traverse all the cuts of rank R, we only need to change the loop bounds at line 3 in Algorithm 1 to for $(r = R; r \leq R; r++)$. Thus, starting with an empty cut we can find the lexically smallest consistent cut of rank r in $\mathcal{O}(n_u)$ time with the GETMINCUT routine. Then we repeatedly find its lexical successor of the same rank, until we have traversed the lexically biggest cut of rank R. Similarly, consistent cuts between the ranks of R_1 and R_2 can be traversed by changing the loop at line 3 in Algorithm 1 to: for $(r = R_1; r \leq R_2; r++)$.

Lemma 3. *Let L_k denote the number of consistent cuts of rank k for a computation $(E, \rightarrow)$. Then, traversing consistent cuts of rank r takes $\mathcal{O}(n_u^2 L_r)$ time with Algorithm 1. For the same traversal, the traditional BFS algorithm requires $\mathcal{O}(n^2 \sum_{k=1}^{r} L_k)$ time, and Lex algorithm takes $\mathcal{O}(n^2 \sum_{k=1}^{|E|} L_k)$ time.*

5 Time and Space Complexity

Algorithm 1 requires a computation in its uniflow chain partition. Multiple polynomial time algorithms exist to find a non-trivial uniflow chain partition of a poset, and we give a vector clock based online algorithm to find one that takes $\mathcal{O}(n)$ time per event. We analyze the worst case time and space complexities of our algorithms.

Given any computation on n processes and E events, we can find its trivial uniflow chain partition in $\mathcal{O}(n|E|log|E|)$ time by lexically ordering the vector clocks of all the events. Suppose the number of chains in the uniflow partition is n_u, then the step of computing new vector clocks takes $\mathcal{O}(n_u|E|\cdot\Delta)$ time where Δ is the maximum in-degree of any event in the computation; note that $\Delta \leq n$. The GetMinCut sub-routine has only one for loop that iterates over the chains of the uniflow partition. Hence, it takes $\mathcal{O}(n_u)$ time in the worst case. The optimized version of finding the successor, sub-routine GetSuccessorOptimized, takes $\mathcal{O}(n_u^2)$ time in the worst case due to the two nested for loops at lines 3, and 10. Hence, for any rank, our algorithm requires $\mathcal{O}(n_u^2)$ time per consistent cut in the uniflow partition. Re-mapping this cut to the original computation takes $\mathcal{O}(n_u + n^2)$ time. Thus, we take $\mathcal{O}(n_u^2 + n^2)$ time per consistent cut.

Theorem 1. *Given a computation $P = (E, \rightarrow)$ on n processes, Algorithm 1 performs breadth-first traversal of its lattice of consistent cuts using $\mathcal{O}((n_u + n)|E|)$ space which is polynomial in the size of the computation.*

Proof. Storing the original computation requires $\mathcal{O}(n|E|)$ space — each event's vector clock having at most n integers. Vector clocks for the uniflow chain partition with n_u chains takes $\mathcal{O}(n_u)$ space per event. Thus, we require $\mathcal{O}(n_u)|E|$ additional space overall to store the computation in its uniflow form. Traversing the lattice as per Algorithm 1 only requires $\mathcal{O}(n_u^2)$ space as at most two vectors of length n_u are stored/created during this traversal, and we use the auxiliary matrix of $n_u \times n_u$ size in the optimized implementation of GetSuccessor. From Lemma 1 we know that $n_u \leq |E|$. Thus, the worst case space complexity is $\mathcal{O}(|E|^2 + n|E|)$ which is polynomial in the size of the input.

6 Experimental Evaluation

We conduct an experimental evaluation to compare the space and time required by BFS, Lex, and our uniflow based traversal algorithm to traverse consistent cuts of specific ranks, as well as all consistent cuts up to a given rank. We do not evaluate DFS implementation as previous studies have shown that Lex implementation outperforms DFS based traversals in both time and space [7,8,16]. Lexical enumeration is significantly better for enumerating all possible consistent cuts of a computation [7,8]. However, it is not well suited for only traversing cuts of specified ranks, or finding the smallest counter example. For these tasks, BFS traversal remains the algorithm of choice. We optimize the traditional BFS implementation as per [16] to enumerate every global state exactly

once. We use seven benchmark computations from recent literature on traversal of consistent cuts [7,8]. The details of these benchmarks are shown in first four columns of Table 1. Benchmarks *d-100, d-300* and *d-500* are randomly generated posets for modeling distributed computations. The benchmarks *bank*, and *hedc* are computations obtained from real-world concurrent programs that are used by [11,14,31] for evaluating their predicate detection algorithms. The benchmark *bank* contains a typical error pattern in concurrent programs, and *hedc* is a web-crawler. Benchmarks *w-4* and *w-8* have 480 events distributed over 4 and 8 processes respectively, and help to highlight the influence of degree of parallelism on the performance of enumeration algorithms. We conduct two sets of experiments: (a) complete traversal of lattice of consistent cuts (of the computation) in BFS manner, and (b) traversal of cuts of specific ranks. We conduct all the experiments on a Linux machine with an Intel Core i7 3.4 GHz CPU, with L1, L2 and L3 caches of size 32 KB, 256 KB, and 8192 KB respectively. We compile and run the programs on Oracle Java 1.7, and limit the maximum heap size for Java virtual machine (JVM) to 2 GB. For each run of our traversal algorithm, we use the online partition algorithm (see Appendix B in [9]) to find the uniflow chain partition of the poset. The runtimes and space reported for our uniflow traversal implementation include the time and space needed for finding and storing the uniflow chain partition of the poset.

Table 1. Benchmark details, heap-space consumed (in MB) and runtimes (in seconds) for two BFS implementations to traverse the full lattice of consistent cuts. T_{part}= time (seconds) to find uniflow partition; × = out-of-memory error

Name	n	$\|E\|$	Approx. # of cuts	n_u	T_{part}	Traditional BFS		Uniflow BFS	
						Space	Time	Space	Time
d-100	10	100	1.2×10^6	26	0.030	108	0.48	31	0.37
d-300	10	300	4.3×10^7	68	0.031	842	16.84	33	46.20
d-500	10	500	4.9×10^9	112	0.033	893	108.07	34	607.55
bank	8	96	8.2×10^8	8	0.023	×	×	59	73.2
hedc	12	216	4.5×10^9	26	0.028	×	×	56	1129
w-4	4	480	9.3×10^6	121	0.036	258	0.99	25	8.59
w-8	8	480	7.3×10^9	63	0.032	×	×	40	1445.57

Table 1 compares the size of JVM heap and runtimes for traditional BFS and our uniflow based BFS traversal of lattice of consistent cuts of the benchmarks. The traditional BFS implementations runs out of memory on *hedc, bank*, and *w-8*. Our implementation requires significantly less memory, and even though it is slower, it enables us to do BFS traversal on large computations — something that is impossible with traditional BFS due to its memory requirement.

Table 2 highlights the strength of our algorithm in traversing consistent cuts of specific ranks. We compare our implementation with traditional BFS as well

Table 2. Runtimes (in seconds) for tbfs: Traditional BFS, lex: Lexical, and uni: Uniflow BFS implementations to traverse cuts of given ranks

Name	$r = \frac{\lvert E\rvert}{4}$			$r = \frac{\lvert E\rvert}{2}$			$r = \frac{3\lvert E\rvert}{4}$			$r \leq 32$		
	tbfs	lex	uni	tbfs	lex	uni	tbfs	lex	uni	tbfs	lex	uni
d-100	0.12	0.10	0.04	0.22	0.11	0.05	0.20	0.89	0.04	0.19	0.93	0.12
d-300	0.39	1.23	0.05	2.70	1.15	0.07	6.33	1.25	0.13	0.20	1.22	0.14
d-500	2.29	5.73	0.11	7.83	6.52	0.33	67.59	6.86	1.48	0.19	4.93	0.19
bank	3.36	16.80	0.27	×	16.34	3.07	×	17.02	0.32	45.43	16.87	5.70
hedc	4.72	16.50	0.40	×	152.76	15.70	×	153.54	0.51	0.23	128.60	0.12
w-4	0.09	0.18	0.07	0.53	0.18	0.10	0.93	0.19	0.09	0.01	0.13	0.05
w-8	26.39	143.08	0.72	×	171.23	120.27	×	169.21	3.09	0.02	196.21	0.05

as the implementation of Lexical traversal. For traversing consistent cuts of three specified ranks (equal to quarter, half, and three-quarter of number of events) our algorithm is consistently and significantly faster than both traditional BFS, as well as Lex algorithm. Thus, it can be extremely helpful in quickly analyzing traces when the programmer has knowledge of the conditions when an error/bug occurs. In addition, there are many cases when we are not interested in checking all consistent cuts of a computation. It has been argued that most concurrency related bugs can be found relatively early in execution traces [4,25]. We also perform well in visiting all consistent cuts of rank less than or equal to 32. Hence, our implementation is faster on most benchmarks for smaller ranks, and requires much less memory (memory consumption details for this experiment are given in the extended version of the paper at [9]). These results emphasize that our algorithm is useful for practical debugging tasks while consuming less resources.

7 Future Work and Conclusion

Algorithm 1 can perform the BFS traversal without regenerating the vector clocks for uniflow chain partitions. This is particularly beneficial for the computations in which $|E| >> n$, and hence the $\mathcal{O}(|E|^2)$ space needed to regenerate the vector clocks is expensive. Observe that any chain partition, including a uniflow chain partition, of a computation is only an arrangement of its graph. Hence, we can implement Algorithm 1 without regenerating new vector clocks, and by only finding the positions of the events in the uniflow chain partition. To do so, we assign a unique id to each event, and then place this event id on its corresponding uniflow chain. We also store a mapping of original vector clocks against the event ids. The space requirement for our algorithm will reduce to $\mathcal{O}(n_u \cdot n)$ as we do not regenerate vector clock, and computation of projections can be performed using $n_u \times n$ space instead of $n_u \times n_u$ space. As a future work, we plan to implement and evaluate this strategy.

As Algorithm 1 traverses cuts of rank $r+1$ independently of those of rank r, we can parallelize rank traversals using a parallel-for loop at line 3 of Algorithm 1. We intend to implement this parallel approach and compare its performance against parallel traversal algorithms such as Paramount [8].

For verification and analysis of parallel programs, breadth-first-search based traversal of global states is a crucial routine. We have reduced the space complexity of this routine from exponential to quadratic in the size of input computation. This reduction in space complexity allows us to analyze computation with high degree of parallelism with relatively small memory footprint — a task that is practically impossible with traditional BFS implementations.

References

1. Alagappan, R., Ganesan, A., Patel, Y., Pillai, T.S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Correlated crash vulnerabilities. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), GA, pp. 151–167. USENIX Association (2016)
2. Alagar, S., Venkatesan, S.: Hierarchy in testing distributed programs. In: Fritzson, P.A. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 101–116. Springer, Heidelberg (1993). doi:10.1007/BFb0019404
3. Alagar, S., Venkatesan, S.: Techniques to tackle state explosion in global predicate detection. IEEE Trans. Softw. Eng. **27**, 412–417 (2001)
4. Ball, T., Burckhardt, S., Coons, K.E., Musuvathi, M., Qadeer, S.: Preemption sealing for efficient concurrency testing. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 420–434. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12002-2_35
5. Bianco, L., Dell Olmo, P., Giordani, S.: An optimal algorithm to find the jump number of partially ordered sets. Comput. Optim. Appl. **8**(2), 197–210 (1997)
6. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
7. Chang, Y., Garg, V.K.: Quicklex: a fast algorithm for consistent global states enumeration of distributed computations. In: 19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14–17, 2015, Rennes, France, pp. 25:1–25:17 (2015)
8. Chang, Y.-J., Garg, V.K.: A parallel algorithm for global states enumeration in concurrent systems. In: ACM SIGPLAN Notices, vol. 50, pp. 140–149. ACM (2015)
9. Chauhan, H., Garg, V.K.: Space efficient breadth-first and level traversals of consistent global states of parallel programs (extended version). https://arxiv.org/abs/1707.07788
10. Chein, M., Habib, M.: The jump number of dags and posets: an introduction. Ann. Discrete Math. **9**, 189–194 (1980)
11. Chen, F., Serbanuta, T.F., Roşu, G.: jPredictor: a predictive runtime analysis tool for java. In: Proceedings of the International Conference on Software Engineering, pp. 221–230 (2008)
12. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proceedings of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 163–173, May 1991

13. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial-ordering. In: Raymond, K. (ed.) Proceedings of the 11th Australian Computer Science Conference (ACSC), pp. 56–66, February 1988
14. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the Conference on Programming Language Design and Implementation, pp. 121–133 (2009)
15. Ganter, B.: Two basic algorithms in concept analysis. In: Kwuida, L., Sertkaya, B. (eds.) ICFCA 2010. LNCS, vol. 5986, pp. 312–340. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11928-6_22
16. Garg, V.K.: Enumerating global states of a distributed computation. In: Proceedings of the International Conference on Parallel and Distributed Computing Systems, pp. 134–139 (2003)
17. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **5**(3), 299–307 (1994)
18. Habib, M., Medina, R., Nourine, L., Steiner, G.: Efficient algorithms on distributive lattices. Discrete Appl. Math. **110**(2–3), 169–187 (2001)
19. Huang, J., Zhang, C.: Persuasive prediction of concurrency access anomalies. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 144–154 (2011)
20. Jegou, R., Medina, R., Nourine, L.: Linear space algorithm for on-line detection of global predicates. In: Proceedings of the International Workshop on Structures in Concurrency Theory, pp. 175–189 (1995)
21. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM (CACM) **21**(7), 558–565 (1978)
22. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
23. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 37–48 (2006)
24. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG), pp. 215–226 (1989)
25. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In Proceedings of Conference on Programming Language Design and Implementation, pp. 446–455 (2007)
26. Pruesse, G., Ruskey, F.: Gray codes from antimatroids. Order **10**, 239–252 (1993)
27. Song, W., Gkountouvas, T., Birman, K., Chen, Q., Xiao, Z.: The freeze-frame file system. In: ACM Symposium on Cloud Computing (SOCC) (2016)
28. Squire, M.B.: Enumerating the ideals of a poset. In: Ph.D. Dissertation, Department of Computer Science, North Carolina State University (1995)
29. Steiner, G.: An algorithm to generate the ideals of a partial order. Oper. Res. Lett. **5**(6), 317–320 (1986)
30. Sysło, M.M.: Minimizing the jump number for partially ordered sets: a graph-theoretic approach. Order **1**(1), 7–19 (1984)
31. von Praun, C., Gross, T.R.: Object race detection. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 70–82 (2001)