Parallel Algorithms for Predicate Detection

Vijay K. Garg
Electrical and Computer Engineering
The University of Texas,
Austin, TX 78712.
Email: garg@ece.utexas.edu

Rohan Garg
Electrical and Computer Engineering
The University of Texas,
Austin, TX 78712.
Email: rohanvgarg@utexas.edu

***Abstract*—Given a trace of a distributed computation and a desired predicate, the predicate detection problem is to find a consistent global state that satisfies the given predicate. The predicate detection problem has many applications in the testing and runtime verification of parallel and distributed systems. We show that many problems related to predicate detection are in the parallel complexity class NC. Given a computation on n processes with at most m events per process, our parallel algorithm to detect a given conjunctive predicate takes $O(\log mn)$ time and $O(m^3n^3 \log mn)$ work. The sequential algorithm takes $O(mn^2)$ time. For data race detection, we give a parallel algorithm that takes $O(\log mn \log n)$ time, also placing that problem in NC. This is the first work, to the best of our knowledge, that places the parallel complexity of such predicate detection problems in the class NC.**

1. Introduction

Debugging and testing multithreaded software is widely acknowledged to be a hard task. Sometimes it takes a programmer days to locate a single bug, especially when the bug appears in one thread schedule but not in others. The current debugging and testing method for multithreaded programs is as follows. The programmer tries the program with multiple inputs in the hope of finding a faulty execution. However, the behavior of a multithreaded program depends not only on the external user input, but also the thread schedule and the order in which locks are obtained by the program. It is easy for the testing process to miss a bug that arises with an alternate schedule. One of the fundamental problems in debugging these systems is to check if the user-specified condition exists in *any* global state of the system that can be reached by an alternative thread schedule. This problem, called *predicate detection*, takes a concurrent computation (in an online or offline fashion) and a condition that denotes a bug (for example, violation of a safety constraint), and outputs a schedule of threads that exhibits the bug if possible. Predicate detection is predictive because it generates inferred reachable global states from the computation; an inferred reachable global state might not be observed during the execution of the program, but is possible if the program is executed in a different thread interleaving.

The technique of predicate detection was introduced by Cooper and Marzullo [5] and Garg and Waldecker [10] for distributed debugging. Later, jPredictor [4] applied the technique to multithreaded programs. Current methods for predictive predicate detection are based upon modeling a computation as a poset, a set of events partially ordered by Lamport's happened-before relation [17]. A boolean predicate B is true in a computation iff there exists a consistent global state that satisfies B. The poset model has been widely used for concurrent and distributed debugging [3], [4], [7], [11], [18], [23], [27]. The predicate detection problem has many applications besides the testing of distributed and concurrent systems. For example, classic problems in distributed computing such as termination detection, deadlock detection, and mutual exclusion can be modeled as predicate detection. Similarly, classic problems in parallel computing such as mutual exclusion violation, data races, and atomicity violation can also be modeled as predicate detection.

Detection of a general predicate is NP-complete [3] and therefore researchers have explored special classes of predicates. In this paper, we present parallel algorithms for two classes of predicates — conjunctive predicates and data race predicates. Detection of conjunctive predicates was discussed by Garg and Waldecker in [11]. Distributed on-line algorithms for detecting conjunctive predicates were presented in Garg and Chase [8]. Observer-independent predicates were introduced by Charron-Bost, Delporte-Gallet, and Fauconnier [1]. Hurfin, Mizuno, Raynal and Singhal [15] gave a distributed algorithm that does not use any additional messages for predicate detection. Distributed algorithms for off-line evaluation of global predicates are also discussed in Venkatesan and Dathan [28]. Stoller and Schneider [26] have shown how Cooper and Marzullo's algorithm can be integrated with that of Garg and Waldecker's to detect conjunction of global predicates.

While there has been extensive work in online and offline distributed algorithms for conjunctive predicate detection, we are not aware of any work that explores the parallel complexity of problems in predicate detection. In this paper, our main result for the parallel complexity of

conjunctive predicate detection is as follows.

Theorem 1. The conjunctive predicate detection problem on n processes with at most m states can be solved in $O(\log mn)$ time using $O(m^3n^3 \log mn)$ operations on the common CRCW PRAM.

Applying this result, we show that any boolean expression B in disjunctive normal form with r disjuncts can be detected in $O(\log mn)$ time using $O(rm^3n^3 \log mn)$ operations on the CRCW PRAM. We also give an algorithm to compute the slice of a computation [21] with respect to any conjunctive boolean predicate in $O(\log mn)$ time. A slice of a computation concisely represents *all* the consistent cuts of the computation that satisfies the predicate.

The problem of data race detection has also attracted wide attention, for example, it has been studied in [19], [22], [24], [25], [29]. None of these papers explore the parallel complexity of the problem. Our main result for data race predicate detection is

Theorem 2. Consider the execution trace on n processes and q objects with at most m events per process.

1) There exists a parallel algorithm that detects the data race predicate in $O(1)$ time and $O(m^2n^2)$ work using $O(m^2n^2)$ processors on CREW PRAM.
2) There exists a parallel algorithm that detects the data race predicate in $O(\log m)$ time and $O(mn(n+q)\log m)$ work using $O(mn^2)$ processors on CREW PRAM.
3) There exists a parallel algorithm that detects the data race predicate in $O(\log mn \log n)$ time and $O(mnq \log mn \log n)$ work on CREW PRAM.

The paper is organized as follows. Section 2 gives our model of a computation and defines the problem of predicate detection. Section 3 presents the parallel algorithm for conjunctive predicates. Section 4 describes three algorithms for data race detection. Finally, Section 5 gives the conclusions and future work.

2. Our Model

This section presents the concepts of local and global predicates. The reader is referred to [12] for more comprehensive background material. We assume a loosely-coupled message-passing system without shared memory or a global clock. A distributed program consists of a set of n processes denoted by $P_1, P_2, ..., P_n$ communicating solely via asynchronous messages. We assume that no messages are lost, altered or spuriously introduced. We do not make any assumptions about a FIFO nature of the channels. In this paper, we will be concerned with a single run of a distributed program.

Each process P_i in that run generates a single execution trace which is a finite sequence of local states, or simply *states*. The state of a process is defined by the values of all its variables including its program counter. The state transition occurs in any process due to an internal action, the send of a message, or the receive of a message. Let S be the set of all states in the computation. We define the usual happened-before relation ($\rightarrow$) on the states (similar to Lamport's happened-before relation between events) as follows. If state s occurs before t in the same process, then $s \rightarrow t$. If the event following s is a send of a message and the event preceding t is the receive of that message, then $s \rightarrow t$. Finally, if there is a state u such that $s \rightarrow u$ and $u \rightarrow t$, then $s \rightarrow t$. A *computation* is simply the poset given by $(S, \rightarrow)$.

It is useful to define *consistent global states* of a computation. Let $s||t$ denote that states s and t are incomparable, i.e., $s||t \equiv s \not\rightarrow t \wedge t \not\rightarrow s$. States s and t are called *concurrent*. A *consistent global state* G is an array of states such that $G[i]$ is the state on P_i and $G[i]||G[j]$ for all i, j. Consistent global states are also referred to as *consistent cuts* in the literature. Consistent global states model *possible* global states in a parallel or a distributed computation.

We assume that there is a vector clock algorithm [6], [20] running with the computation which is used to track the happened-before relation. A vector clock algorithm assigns a vector $s.v$ to every state s such that $s \rightarrow t$ iff $s.v < t.v$. The vectors $s.v$ and $t.v$ are called the *vector clocks* at s and t. Fig. 1 shows an example of an execution trace with vector clocks.

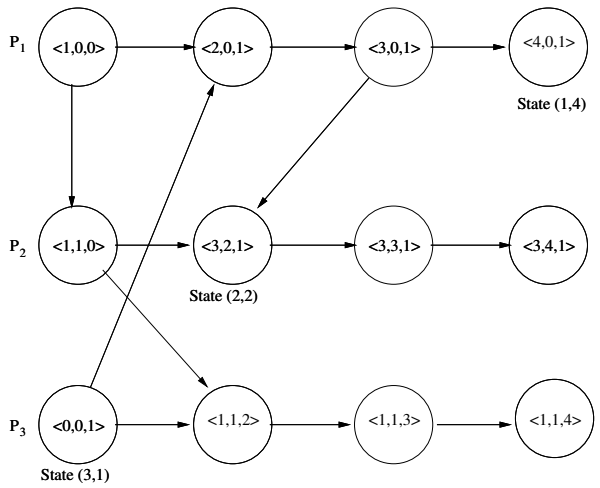

Figure 1. State-Based Model of a Distributed Computation

A *local predicate* is defined as any boolean-valued formula on a local state. For example, the predicate "P_i is in the critical section" is a local predicate. It only depends on the state of P_i, and P_i can obviously detect that local predicate on its own. A *global predicate* is a boolean-valued formula on a global state. For example, the predicate (P_1 is in the critical section)$\wedge$(P_2 is in the critical section) is a global predicate. It depends upon the states of multiple processes. Given a computation $(S, \rightarrow)$, and a boolean predicate B, the *predicate detection* problem is to determine if there exists a consistent global state G in the computation such that B evaluates to true on G.

We discuss two types of predicates in this paper. First, a global predicate formed only by the conjunction of local predicates is called a Weak Conjunctive Predicate (WCP)

[11], or simply, a conjunctive predicate. Thus, a global predicate B is a conjunctive predicate if it can be written as $l_1 \wedge l_2 \wedge \ldots \wedge l_n$, where each l_i is a predicate local to P_i. We restrict our consideration to conjunctive predicates because any boolean expression of local predicates can be detected using an algorithm that detects conjunctive predicates as follows. We convert the boolean expression into its disjunctive normal form. Now each of the disjuncts is a pure conjunction of local predicates and can be detected using a conjunctive predicate algorithm.

Second, we discuss data race predicates. A data race exists when there are two concurrent accesses to a shared object such that at least one of them is a write. Both of these classes of predicates model a large number of possible bugs.

3. Detecting Conjunctive Predicates

In this section we describe a parallel algorithm to detect a conjunctive predicate $B = l_1 \wedge l_2 \wedge \ldots \wedge l_n$. To detect B, we need to determine if there exists a consistent global state G such that B is true in G. Note that given a computation on n processes each with m states, there can be as many as m^n possible consistent global states. Therefore enumerating and checking the condition B for all consistent global states is not feasible. Since B is conjunctive, it is easy to show [11] that B is true iff there exists a set of states $s_1, s_2, ..., s_n$ such that (1) for all i, s_i is a state on P_i, (2) for all i, l_i is true on s_i and (3) for all i, j: $s_i \| s_j$. Our detection algorithm will either output such local states or guarantee that it is not possible to find them in the computation. When the global predicate B is true, there may be multiple G such that B holds in G. For conjunctive predicates B, it is known that there is a unique minimum global state G that satisfies B whenever B is true in a computation [3]. We are interested in algorithms that return the minimum G that satisfies B because G corresponds to the smallest counter-example to programmer's understanding since B typically represents the violation of a safety constraint.

Our parallel algorithm is based on the setting where the execution traces for all processes have been collected at one process. For example, in the centralized algorithm for conjunctive predicate detection, one process serves as a checker and collects the traces. All other processes involved in detecting the WCP, referred to as application processes, check for local predicates during the computation. Each process P_i also maintains the vector clock algorithm. Whenever the local predicate of a process becomes true for the *first* time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process [11].

The checker process uses queues of incoming messages to hold incoming local snapshots from application processes. We require that messages from an individual process be received in FIFO order. At the end of the computation, the checker process has a sequence of local states from each process where its local predicate is true. We now describe a sequential and a parallel algorithm to detect B on these traces. The sequential algorithm is an adaptation of the algorithm from [11]. We include it here because it is instrumental in understanding the parallel algorithm. Moreover, the correctness of the parallel algorithm is shown by assuming the correctness of the sequential algorithm.

3.1. A Sequential Algorithm

The sequential algorithm in Fig. 2 takes as input n traces each of size m. The variable $states[i][j]$ stores the vector clock of j^{th} local state on P_i. In step 1, we create an array cut that maintains the current global state that is being checked for consistency. It is initialized with the first local state at each process. The variable $current[i]$ keeps track of the index in the trace for that process. In step 2, we give a color to each of the local states in the cut. The color of a state is either red or green and indicates whether the state has been eliminated in the current cut or not. A state is green only if it is concurrent with all other green states. A state is red only if it cannot be part of a consistent cut that satisfies the WCP. In step 2, we mark all local states that are less than some other local states in the cut as red.

In step 3, we advance on the process which has a red state. Suppose we get the new state from P_i. The color of $cut[i]$ is temporarily set to green. It may be necessary to change some green states to red to preserve the property that all green states are mutually concurrent. Hence, we must compare the vector clock of $cut[i]$ to each of the other green states. Whenever the states are comparable, the smaller of the two is painted red.

We now discuss the time complexity of the algorithm. Note that it takes only two comparisons to check whether two vector clock timestamps are concurrent if we know the processes that generated these vector clocks [20]. Hence, the *for* loop in step 3 requires at most n comparisons. This *for* loop is called at most once for each state, and there are at most mn states. Therefore, at most $O(n^2m)$ comparisons are required by the algorithm.

3.2. The ParallelCut Algorithm

We now consider parallelization of the sequential algorithm. The *for* loop in the sequential algorithm can be trivially parallelized; however, the *while* loop appears to be inherently sequential. A different approach is required to exploit parallelism. Our approach is based on computing a *state rejection graph* for the trace. The state rejection graph is a directed graph with all local states as vertices of the graph. Let state j on process i be denoted as (i, j). The state rejection graph puts a rejection edge from the state (i, j) to (i', j') if the rejection of state (i, j) as a possible component of the consistent cut implies that the state (i', j') will also be rejected.

In Fig. 4 we show the state rejection graph of the computation in Fig. 1. The first state in P_1 given be the vector clock $\langle 1, 0, 0\rangle$ will be rejected by the sequential algorithm because it happened before $\langle 1, 1, 1\rangle$. The sequential algorithm will then move P_1 to the second state $\langle 2, 0, 1\rangle$

```
function SequentialCut()
Input: states : array[1...n][1...m] of vectorClock;
// sequence of local states given by vector clocks
Output: Consistent Global State as array cut[1...n]

Step 1: Create cut: set of initial states
var cut : array[1...n] of vectorClock;
for i := 1 to n do
    current[i] := 1 ;
    cut[i] := states[i, current[i]] ;

Step 2: Create color: array[1..n] of {red, green};
var color : array[1...n] of {red, green} initially green
for i := 1 to n do
    if ∃j : cut[i] → cut[j] then
        color[i] := red;

Step 3: Advance cut
while (∃i : (color[i] = red)) do
    if cut[i] is the last state then
        output("No satisfying Consistent Cut");
    else
        current[i] := current[i] + 1; // advance the cut
        cut[i] := states[i, current[i]];
        cut[i].color := green;
        for j := 1 to n do
            if (cut[j].color = green) then
                if (cut[i].v < cut[j].v) then cut[i].color := red;
                if (cut[j].v < cut[i].v) then cut[j].color := red;
        endfor
endwhile;

return ConsistentCut := cut ;
```

Figure 2. Sequential Conjunctive Predicate Detection Algorithm.

(successor of the first state in P_1). However, this would imply that the state $\langle 0,0,1\rangle$ will be rejected because $\langle 0,0,1\rangle$ happened before $\langle 2,0,1\rangle$. Hence, there is a rejection edge from $\langle 1,0,0\rangle$ to $\langle 0,0,1\rangle$. Similarly, the rejection of $\langle 0,0,1\rangle$ implies that P_3 will move to $\langle 1,1,2\rangle$. However, that move will result in rejection of the state $\langle 1,1,0\rangle$. Therefore, we put a rejection edge from $\langle 0,0,1\rangle$ to $\langle 1,1,0\rangle$. Finally, the rejection of $\langle 1,1,0\rangle$ will result in P_2 moving to $\langle 3,2,1\rangle$ which will result in the rejection of $\langle 2,0,1\rangle$ and $\langle 3,0,1\rangle$. All the rejection edges are shown in dashed arrows in Fig. 4. We show how such a graph can be constructed efficiently in parallel. The next step in the algorithm is to compute the transitive closure of this graph. Finally, the algorithm determines the least local state at each process which has not been rejected. In this example, it is the fourth state on P_1, the second state on P_2 and the third state on P_3.

Our parallel algorithm is presented in Fig. 3. The input to the algorithm is same as that of the sequential algorithm: a two-dimensional array of vector clocks.

We now explain the steps of the algorithm.

Step 1: We first create F, the set of all initially rejected states. Let I be the global state consisting of each processor's first local state, i.e., $I = \{(i,1) \mid i \in 1..n\}$. If there are no dependencies between any of these states, we have

```
function ParallelCut()
Input: states : array[1...n][1...m] of vectorClock
// Sequence of local states at each process
Output: Consistent Global State as array cut[1...n]

Step 1: Create F: set of states rejected in the first round
    var F : array[1...n] of 0...1 initially 0
    for all (i ∈ 1...n, j ∈ 1...n) in parallel do
            if ((i,1) → (j,1)) then
                F[i] := 1

Step 2: Create R: State Rejection Graph
// Represented as Adjacency Matrix
    var R : [(1...n, 1...m), (1...n, 1...m)] of 0...1
    for all (i ∈ 1...n, j ∈ 1...m) in parallel do
        R[(i,j),(i,j)] := 1
    for all (i ∈ 1...n, j ∈ 1...m − 1, i' ∈ 1...n, j' ∈ 1...m)
            such that i ≠ i' in parallel do
            if ((i',j') → (i,j+1)) then
                R[(i,j),(i',j')] := 1
            else
                R[(i,j),(i',j')] := 0

Step 3: Create RT: transitive closure of R
    var RT : array[(1...n, 1...m), (1...n, 1...m)] of 0...1
    RT := TransitiveClosure(R);

Step 4: Create valid: replace invalid states by 0
    var valid : array[[1...n][1...m] of 0...1
    for all (i ∈ 1...n, j ∈ 1...m) in parallel do
        valid[i][j] := 1
    for all (i ∈ 1...n, i' ∈ 1...n, j' ∈ 1...m) in parallel do
        if (F[i] = 1) ∧ (RT[(i,1),(i',j')] = 1) then
            valid[i'][j'] := 0


Step 5: Create cut: First Consistent Global State
    var cut : array[1...n] of 0...m initially 0
    for all (i ∈ 1...n, j ∈ 1...m) in parallel do
            if (valid[i][j] ≠ 0) then
                if (j = 1) ∨ ((j > 1) ∧ (valid[i][j−1] = 0) then
                    cut[i] := j
    for all (i ∈ 1...n) in parallel do
            if (cut[i] = 0) then
                output("No satisfying Consistent Cut")

    return ConsistentCut := cut ;
```

Figure 3. The ParallelCut algorithm to find the first consistent cut.

already reached the first consistent global state. Else, if there is a dependency from one of these states to another, we reject whichever state happened-before the other and add it to F. We represent the set F by a boolean bit array of size n that is indexed by processor. Thus, $F[i] = 1$ iff $(i,1)$ is in F. We initialize the bit array with zeroes, i.e., F is initially empty. Then, we set $F[i]$ to 1 whenever there exists a state $(j,1)$ such that $(j,1) \rightarrow (i,1)$. Formally,

$$\forall i \in 1 \ldots n : F[i] = 1 \equiv \exists j : (i,1) \rightarrow (j,1)$$

This step can be done in $O(1)$ time in parallel with

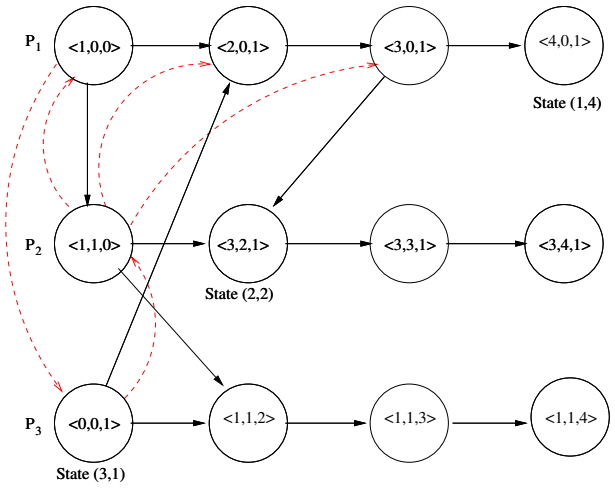

Figure 4. State Rejection Graph of a computation shown in dashed arrows

$O(n^2)$ work by using a separate processor for each value of i and j.

In Fig. 1, $I = \{\langle 1,0,0\rangle, \langle 1,1,0\rangle, \langle 0,0,1\rangle\}$. Among these elements, $\langle 1,0,0\rangle \rightarrow \langle 1,1,0\rangle$. Thus, $\langle 1,0,0\rangle \in F$.

Step 2: In step two, we create the state rejection graph represented as an adjacency matrix. We define a new two-dimensional array called R, which is of size $mn \times mn$, where each row and each column represents a different state. In this directed graph, there is an edge from state (i,j) to another state (i',j') only if we know that once state (i,j) is rejected, state $(i'j')'$ will also be rejected. In the adjacency matrix R, this is represented as $R[(i,j),(i',j')] = 1$. Additionally, we make the diagonal of the matrix all 1's. This means that the relation is reflexive once we create the matrix. We show that creating this boolean matrix can be done in constant time. First, setting the diagonal to all 1's in R takes constant time. We now discuss how off-diagonal entries are set. This is a crucial step in our algorithm. Suppose that a state (i,j) is rejected. We know that the processor will advance to the next state. Then, the next choice for that processor is $(i, j+1)$. Thus, the rejection of (i,j) would lead to the rejection of all states (i',j') where $(i',j') \rightarrow (i,j+1)$. Formally,

$$R[(i,j),(i',j')] = 1 \equiv (i',j') \rightarrow (i,j+1)$$

By using a separate processor for each tuple (i,j,i',j') we can set R in $O(1)$ time and $O(m^2n^2)$ work. We represent the state rejection graph as a boolean matrix so that we can compute the transitive closure of this graph by doing matrix multiplications.

Step 3: In step three, we take the transitive closure of R. The transitive closure of R is also represented as an adjacency matrix RT. Formally, for all $i,i' \in 1 \ldots n$ and $j,j' \in 1 \ldots m$: $RT[(i,j)(i',j')] = 1$ iff there exists a $k \in 1 \ldots mn : R^k[(i,j)(i',j')] = 1$.

It is well known that the transitive closure for any directed graph with $|V|$ vertices can be computed in $O(\log |V|)$ time using $O(|V|^3 \log |V|)$ work on the common CRCW PRAM [16]. Since our graph has $O(mn)$ vertices, this step takes $O(\log mn)$ time using $O(n^3m^3 \log mn)$ operations on the CRCW PRAM. This is the only step in our algorithm that takes more than $O(1)$ time.

Step 4: In step four, we use both F and RT to determine which states will not be part of the first consistent global state. To do this, we create a new two-dimensional array called $valid[[1 \ldots n][1 \ldots m]]$ where each entry is a value of $0 \ldots 1$. We initialize every entry in $valid$ to 1 in $O(1)$ time with $O(n^2)$ work. The algorithm sets rejected states in $valid$ with a 0. We use F and RT for this purpose. For all possible values of i,j,i', and j', in parallel, we check to see if a state (i,j) is an element of F and if there is an edge from (i,j) to another state (i',j'). If the two states (i,j) and (i',j') fit this criteria, then we set $valid[i'][j']$ to 0. In other words, if a state (i,j) is initially rejected, and there is an edge from (i,j) to (i',j') in RT, then we know the state (i',j') will also be rejected. Formally, for all $i \in 1 \ldots n, j \in 1 \ldots m$:

$$valid[i][j] = 0 \equiv \exists i' : F[(i') = 1 \wedge RT[(i',1),(i,j)] = 1$$

This step can also be done in $O(1)$ time and $O(m^2n^2)$ work. In our example, we compute the states reachable by $\langle 1,0,0\rangle$. In our example, states $\{\langle 0,0,1\rangle, \langle 1,1,0\rangle, \langle 2,0,1\rangle, \langle 3,0,1\rangle\}$ are all reachable by $\langle 1,0,0\rangle$ by following one or more rejection edges. Thus, we mark the states $(1,1),(2,1),(3,1),(1,2),(1,3)$ with $0's$ in $valid$.

Step 5: In step five, we traverse $valid$ and construct the set of states that form the consistent global state in a new array called cut where $cut[i] = j$ signifies that the tuple (i,j) is a part of the consistent global state. To do this, for every process, in parallel, we simply search for the first entry which is nonzero and either it is the first entry or the entry prior to it is zero.

Formally, $\forall i,j : (valid[i][j] \neq 0) \wedge ((j = 1) \vee (valid[i][j-1] = 0)) \equiv (cut[i] = valid[i][j])$.

In our example, we can easily see that the first consistent global state is the set $cut = \{(1,4),(2,2),(3,2)\}$. This step can be done in $O(1)$ time and $O(n^2)$ work.

Table 5 summarizes the time and work complexity of each step. Thus, the algorithm takes $O(\log mn)$ time and $O(m^3n^3 \log mn)$ work on a common CRCW PRAM.

Remark: Note that the algorithm to detect a conjunctive predicate can be used to detect a global predicate in Disjunctive Normal Form. A predicate is in Disjunctive Normal Form (DNF) if it is expressed as a disjunction of k pure conjunctions. To detect a predicate in this form it is sufficient to detect each conjunction in parallel.

3.3. Proof of Correctness

We now prove our algorithm always produces the first consistent global state, should it exist, that satisfies the given conjunctive predicate B. We prove that our algorithm rejects all states for which there does not exist a consistent global state. Once we have removed all these states, our algorithm

Step	Variable	Time on CRCW	Work on CRCW
Step 1	F	$O(1)$	$O(n^2)$
Step 2	R	$O(1)$	$O(m^2n^2)$
Step 3	RT	$O(\log mn)$	$O(m^3n^3 \log mn)$
Step 4	$valid$	$O(1)$	$O(n^2m)$
Step 5	cut	$O(1)$	$O(mn)$
Total		$O(\log mn)$	$O(m^3n^3 \log mn)$

Figure 5. Time and Work Complexity on Common CRCW PRAM.

simply picks the first state from each process that can be part of a consistent global state. Lemma 1 shows the property of F. Lemma 2 shows the property of states that are reachable from F in one or more steps. Finally, Theorem 3 shows the correctness invoking Lemma 1, Lemma 2 and properties of the SequentialCut algorithm.

Lemma 1. . For any of the initally rejected states $s \in F$, there does not exist a consistent global state G, where $s \in G$.

Proof: Let $s = (i, 1)$. From the definition of F, there exists j such that $(i, 1) \rightarrow (j, 1)$. Since $(j, 1) \rightarrow (j, k)$ for all $k > 1$, from transitivity of $\rightarrow$, we get that $(i, 1) \rightarrow (j, k)$ for all k. Thus, there does not exist a state along process j that is concurrent with state s. Hence, there does not exist a consistent global state G containing s. ■

We define a set $Rejected(t)$ for $t = 1..mn$ that captures the states that are rejected in t steps or less starting from the initial rejected states. Since there are mn states in all, there cannot be a path longer than mn. Formally,

$$(i, j) \in Rejected(t) \equiv \exists (i', j') \in F : R^t[(i', j'), (i, j)] = 1.$$

We now show that if a state s' on $P_{i'}$ is rejected by ParallelCut in any round then there does not exist a consistent global state containing s' or any state prior to s' on $P_{i'}$.

Lemma 2. For all $t \geq 1$, if any state $(i, j) \in Rejected(t)$, then for all $k \leq j$, there does not exist any consistent global state containing (i, k).

Proof: The proof is by induction on t which corresponds to the length of a rejection path from a state in F to s'. When $t = 1$, i.e., there is a direct rejection edge from some state $s \in F$ to s'. Let $s = (i, j)$ and $s' = (i', j')$. From the definition of R, we get that $s' \rightarrow (i, j + 1)$. Therefore, s' cannot be concurrent with any state (i, k) for $k > j$. Since $s = (i, j)$ is in F, we have that any state (i, k) for $k \leq j$ is not in any consistent global state. Hence, s' is not in any consistent global state because it cannot be concurrent with any state on process P_i that is part of a consistent global state. Now consider any state u prior to s'. Due to transitivity, $u \rightarrow (i, j+1)$. Therefore, by the same reasoning u cannot be part of a consistent global state.

Consider any state $s' \in Rejected(t + 1)$ for $t \geq 1$. By definition of $Rejected(t + 1)$, there is a path of length at most $t + 1$ from some state $s \in F$ to s'. Let $s_t = (i, j)$ and $s' = s_{t+1} = (i', j')$. Using the inductive hypothesis, we know that all the states $s_0, s_1, \ldots s_t$ are not in any consistent global state. For there to be a rejection edge in RT from s_t to s_{t+1} it must be that $s_{t+1} \rightarrow succ(s_t)$. From induction, we know that there is no consistent global state containing state s_t or any state prior to s_t. The least state from P_i that can be in a consistent state is $(i, j + 1)$.

Consider the set $L(s') = \{(i', k') | k' \leq j'\}$. Pick any state u in $L(s')$. We claim that state u is not concurrent with any state on P_i that can be in a consistent global state. It cannot be in a consistent global state with any state (i, k) on P_i for $k \leq j$ because s_t is in $Rejected(t)$ and by induction any state prior to (i, k) cannot be part of a consistent global state. Furthermore, u is not concurrent with any state (i, k) on P_i for $k > j$ because $u \rightarrow (i, j + 1)$. From these two assertions, we get that u is not part of any consistent global state. ■

We now show the correctness of the ParallelCut algorithm.

Theorem 3. The *ParallelCut* algorithm returns the least consistent global state satisfying B if there exists one. Otherwise, it returns "no consistent global state".

Proof: We show that the states rejected by the *ParallelCut* are identical to the states rejected by the *SequentialCut* algorithm. Then, the correctness of the sequential algorithm will imply the correctness of the ParallelCut algorithm.

Suppose that a state s is rejected by the sequential cut algorithm. This implies that there is a sequence of the steps of some length, say t, that colors s as red. We do induction on t. When t equals 0, the state s must be an initial state and it must happen before another initial state for it to be colored red in step 2 of the SequentialCut algorithm. By our construction of F, state s will be added to F in step 1 of the ParallelCut algorithm. Now assume that the claim is true for $t = k$ steps of the sequential algorithm. If s is colored red in step $k+1$, then there exists a state u such that $s \rightarrow u$ from step 3 of SequentialCut. If u is an initial state, then s will be rejected in step 1 of the ParallelCut algorithm. Otherwise, there exists a state u' such that the successor of u' is u and u' is colored red at some earlier step. We have that the successor of u' is u and $s \rightarrow u$. Therefore, by our construction of R, there is a rejection edge from u' to s. From induction, u' is rejected by the ParallelCut algorithm, and there is a rejection edge from u' to s. Hence, s is also rejected by the ParallelCut algorithm.

We now show that if a state is not rejected by SequentialCut, then it is not rejected by ParallelCut. We show the contrapositive. If a state is rejected by the ParallelCut algorithm, it is reachable from F in some number of steps. If a state s is part of F, then Lemma 1 shows that s cannot be part of any consistent cut and therefore also rejected by the SequentialCut. Now, consider a state s that is reachable from some state in F by one or more rejection edges. From Lemma 2, if a state s is rejected by the ParallelCut, then that state and all the states before s on that process cannot be part of a consistent cut. Since the SequentialCut algorithm is correct, it would color all those states red and therefore also reject them. ■

Remark: We note that the work complexity of the algorithm is cubic in m. One method to reduce average work complexity is as follows. Instead of starting with the entire trace, one can start with a small prefix of the computation (i.e. smaller value of m). If the algorithm succeeds in finding the satisfying cut, we are done. Otherwise, we double the size of the prefix. This strategy will add a factor of $O(\log m)$ to the time complexity but the algorithm will have a lower work complexity on average if there exists a satisfying cut in some initial prefix.

3.4. Parallel Slicing Algorithm

In many applications, it is not sufficient to find just the least consistent global state that satisfies the given predicate. For example, the detection of temporal logic predicates on a distributed computation requires capturing all consistent global states that satisfy the given predicate. When the given predicate is *regular*, i.e., the set of consistent global states satisfying B form a sublattice, then the set can be precisely characterized by the *slice* of the computation with respect to B. A slice of a computation P with respect to a predicate B is a graph such that the consistent global states of the graph includes all consistent global states that satisfy B and when B is regular, it includes only those. The reader is referred to [13], [21] for discussions of slicing. The algorithm requires the computation of $J(B, e)$ for all $e \in E$ where $J(B, e)$ is the least consistent global state that satisfies B and includes e. Since the computation of $J(B, e)$ is in **NC**, the computation of a slice is also in **NC** because the following algorithm computes the slice in $O(\log mn)$ time.

```
graph function computeSlice(
        B: regular_predicate, P: graph)
var R: graph initialized to P;
begin
    for every element e in P in parallel do
        let J(B,e) be the least global state of P
            that satisfies B and includes e;
        for every f ∈ J(B,e) in parallel do;
            add edge (f,e) to R;
    return R;
    end;
```

Figure 6. An efficient parallel algorithm to compute the slice for a regular predicate B

4. Data Race Predicate Detection

As mentioned earlier, the detection of data races has received wide attention for shared-memory based programs. We assume that such a program has n threads that share q objects. Each thread makes at most m accesses. Given two accesses one can establish a happened-before relation between them as follows. If a thread executes an operation e and then later executes another operation f, then e happened-before f. All operations done by a single thread are therefore totally ordered. If an operation e releases a lock and another operation f acquires that lock, then e happened-before f. Similarly, if any operation e results in sending a signal and another operation f is executed on receiving that signal, then e happened-before f. The data race detection problem can be defined as follows.

Data Race Predicate Detection: Given a multithreaded computation $(E, \rightarrow)$, is there any instance of a

1) *read-write conflict:* Are there two events e and f in E such that e is a read on some object, f is a write on the same object, and e is concurrent with f.
2) *write-write conflict:* Are there two events e and f in E such that both e and f are writes on the same object and e is concurrent with f.

For ease of exposition, we first consider a special case of the problem when we are concerned only with a single object and all accesses need to be mutually exclusive. We then extend mutual exclusion violation algorithms to the general case of the data race predicate.

4.1. Special Case: Mutual Exclusion Violation Predicate

In this section, we discuss detecting the violation of mutual exclusion. Mutual exclusion is used when there is shared data or there is a section of program called the *critical section*, that must be executed by at most one thread (or a process in a distributed system) at any given time. Our model is the same as that for conjunctive predicates detection. There are n processes and each process has a sequential trace of all accesses to the critical section. We assume that there are at most m accesses per process. Each access to the critical section is given by a vector timestamp of size n. The computation has a violation of mutual exclusion if there exists two concurrent accesses to the critical section. The optimal sequential algorithm requires $O(mn \log n)$ time and is based on merging n sorted lists of vector clocks. A straightforward parallel algorithm with $O(1)$ time complexity is shown in Fig. 7. This algorithm requires $O(m^2n^2)$ work and $O(1)$ time with $O(m^2n^2)$ processors. It does not require any additional space besides the traces themselves. Note that we have used the notation $[n]$ to mean $\{1 \ldots n\}$.

We can reduce the work complexity as follows. Let $trace[i]$ be the sorted array of all m vectors on P_i. Given any vector u we can determine if it is comparable to all the vectors in $trace[i]$ using $O(\log m)$ time. The algorithm is shown in Fig. 7. This algorithm requires $O(mn^2 \log m)$ work and $O(\log m)$ time with $O(mn^2)$ processors.

We can improve the work complexity further by using a parallel merge [16] algorithm and additional space as follows. At the beginning of the algorithm, we have n sorted traces each of size m. At each round, we will reduce the number of lists by a factor of two and increase the size of lists by a factor of two. If in any round, we

Mutex Violation Algorithm 1:
Time: $O(1)$, **Additional Space**: $O(1)$
Work: $O(m^2n^2)$

for all $(i \in [n], j \in [n], k \in [m], l \in [m])$
with $i < j$ in parallel do
if $(v[i][k]||v[j][l])$ **return** "mutex violation"
endfor;
return "no violation of mutual exclusion"

Mutex Violation Algorithm 2:
Time: $O(\log m)$, **Additional Space**:$O(1)$
Work: $O(mn^2 \log m)$

for all $(i \in [n], j \in [n], k \in [m])$ with $i < j$ in parallel do
binary search $v[i][k]$ in $trace[j]$
if (incomparable vector found)
return "mutex violation"
endfor;
return "no violation of mutual exclusion"

Figure 7. Brute force parallel algorithms to detect mutex violation

do not succeed because of any incomparable vectors, we have managed to find a violation of mutual exclusion. We can implement each round in parallel time $O(\log mn)$ as follows. For simplicity, we have assumed that n is a power of 2. The goal is to merge trace 0 with trace 1, trace 2 with trace 3 and so on. Assume that there is a thread for each vector. It can determine the rank of the vector in the merged trace by computing the number of elements smaller than itself in its list and the list to be merged. The ranks can be computed in $O(\log mn)$ time by performing binary search. There are $O(\log n)$ rounds giving us the time complexity of $O(\log mn \log n)$. The algorithm is shown in Fig. 8. This algorithm requires $O(mn \log mn \log n)$ work, $O(\log mn \log n)$ time and $O(mn^2)$ space.

4.2. Data Race Detection

In this section, we discuss detecting data races when concurrent reads are allowed but read-write and write-write conflicts must be detected. We use the same model as that of the violation of mutual exclusion, i.e., we have traces of read accesses and write accesses of any object given by the vector timestamps. Associated with each vector is a field $object$ that gives the identifier of the object and a bit op that indicates whether the access is a *read* or a *write*. In data race detection, it is more traditional to use the event based model and we assume that the trace consists of vector clocks for events representing accesses to shared objects instead of vector clocks of states. The $O(1)$ algorithm for mutual exclusion with $O(m^2n^2)$ processors is easily generalized to data race detection. It is shown in Fig. 9. When we compare any event with any other event, we flag violation only if at least one of the accesses is *write* and only if they are on the same object.

We now consider the generalization of algorithm 2 for mutual exclusion. In algorithm 2 for mutual exclusion, we

Mutex Violation Algorithm 3:
procedure ME();
Time: $O(\log mn \log n)$, **Space**:$O(mn)$
Work: $O(mn \log mn \log n)$

$L :=$ set of n traces each with m vectors;
$numTraces := n$; // assume n is a power of 2
for $r := 1 \ldots \log n$ do // in sequence
// for all $j := 0$ to $numTraces/2$ in parallel
// merge trace $2j$ with trace $2j+1$
for all u in trace $2j$ and $2j+1$ in parallel do
// rank of u in $trace[2j]$
$rank1$:= binary search u in $trace[2j]$;
// rank of u in $trace[2j+1]$
$rank2$:= binary search u in $trace[2j+1]$;
if (binary search finds incomparable vector)
return "violation of mutual exclusion"
else
write u at $rank1 + rank2$ in the merged trace;
$numTraces := numTraces/2$;
endfor;
return "no violation of mutual exclusion"

Figure 8. An almost work optimal parallel algorithm to detect mutex violation

Data Race Detection Algorithm 1:
Time: $O(1)$, **Additional Space**:$O(1)$
Work: $O(m^2n^2)$

for all $(i \in [n], j \in [n], k \in [m], l \in [m])$
with $i < j$ in parallel do
if $((v[i][k]||v[j][l])$
$\wedge((v[i][k].op = write) \vee (v[j][l].op = write))$
$\wedge(v[i][k].object = v[j][l].object))$
return "data race"
endfor;
return"no data race"

Figure 9. A Data Race Detection Algorithm with $O(1)$ time and $O(m^2n^2)$ work.

employed binary search to reduce the work complexity. In that algorithm, given any vector u and $trace[i]$, vector u was comparable with all vectors in the $trace[i]$ unless there is a violation of mutual exclusion. For data races, this is not the case since the vector u may be on a different object than a vector in $trace[i]$ or both u and the vector in $trace[i]$ may be reads. However, if we take the projection of trace i for each object and only focus on vectors u that correspond to writes, we can still employ binary search.

The second algorithm starts with taking projections on the traces for individual objects. These projections can be computed in parallel for all traces and all objects. The method $projection$ takes the trace $inTrace$ for any process and an object q as input and returns its projection of the trace on the object q. First, we compute an array loc of the size equal to the size of the trace as follows. We set, for the k^{th} event, $loc[k]$ to be 1 if that event is an operation

```
Data Race Detection Algorithm 2:
Time: O(log m), Additional Space:O(mn),
Work: O(mn(n + q) log m)

Step 1: Compute projections for all objects
for all (i ∈ [n], obj ∈ [q]) in parallel do
  objectTrace[i][obj] := projection(trace[i], obj);

Step 2: Do binary search for each event and process
for all (i ∈ [n], j ∈ [n], k ∈ [m]) in parallel do
  if (v[i][k].op = write)
      obj := v[i][k].object;
      binary search v[i][k] in objectTrace[i][obj]
      if (found incomparable vector)
      return "data race exists" for v[i][k].object;
endfor;
return "no data race"

function projection(inTrace : array[1..m] of
      (vector, object, op), obj: object);
  var
      loc : array[1..m] of integer;
  for all (k ∈ [m]) in parallel do
      if (inTrace[k].object = obj) then
          loc[k] := 1
      else
          loc[k] := 0;
  loc := parallelPrefixSum(loc);
  int s := loc[m]; // size of outTrace;
  outTrace := new array[1..s] of (vector, object, op);
  for all (k ∈ [m]) in parallel do
      if (inTrace[k].object = obj) then
          outTrace[loc[k]] := inTrace[k]
  return outTrace;
end function
```

Figure 10. Parallel algorithm 2 to detect data races

on object q. The parallel-prefix sum operation calculates the output address of each such event in the output trace. Finally, all events that are related to the object q are written in the appropriate locations in $outTrace$. This method takes $O(\log m)$ time due to parallel prefix-sum. Summing up all the work and time in step 1, we get $O(mnq \log m)$ work and $O(\log m)$ time.

In step 2, we perform binary search for each local event and each process. This step takes $O(\log m)$ time and $O(mn^2 \log m)$ work.

This algorithm requires no additional space beyond the traces.

Theorem 4. Consider the execution trace on n processes and q objects with at most m events per process.

1) The first parallel algorithm in Fig. 9 detects data race in $O(1)$ time and $O(m^2n^2)$ work on CREW PRAM.
2) The second algorithm in Fig. 10 detects data race in $O(\log m)$ time and $O(mn(n + q) \log m)$ work on CREW PRAM.

The second algorithm reduces the work but is quadratic in n. We now give a parallel algorithm for data races that reduces the dependence on n to $O(n \log n)$ by generalizing algorithm 3 for mutual exclusion. We check for data race in two steps. In the first step, we take the projection of each object trace with respect to write accesses. Now we invoke the algorithm ME to merge all the write accesses. If we find incomparable vectors, we have found a write-write conflict. Computing the projection of all traces requires $O(\log m)$ time and $O(mnq \log m)$ work. Merging these traces requires $O(\log mn \log n)$ time and $O(mnq \log mn \log n)$ work. In the second step, we check that no read access is concurrent to any write in the merged object trace. This can be done using binary search for all read operations. This step requires $O(\log mn)$ time with $O(mn \log mn)$ work.

```
Data Race Detection Algorithm 3:
Time: O(log mn log n), Additional Space:O(mn),
Work: O(mnq log mn log n)

// Step 1: Merge traces for all write operations for every object
  L := set of n traces each with m vectors;
  numTraces := n; // assume n is a power of 2
  for obj ∈ 1 . . . q in parallel do
    L_obj := L projected on obj and write operations;
    mergedTrace_obj := Algorithm ME applied to L_obj;
    if (incomparable vectors found)
      return "write-write data race";
  endfor;

// Step 2: Do binary search for all read operations
  for all (i ∈ [n], k ∈ [m]) in parallel do
    if (v[i][k].op = read) ∧ (v[i][k].object = obj);
      binary search v[i][k] in mergedTrace_obj
      if (incomparable vectors found)
        return "read-write data race";
  endfor;
  return "no data race"
```

Figure 11. Parallel algorithm 3 to detect data races

Theorem 5. Consider the execution trace on n processes and q objects with at most m events per process. The parallel algorithm in Fig. 11 detects a data race in $O(\log mn \log n)$ time and $O(mnq \log mn \log n)$ work on CREW PRAM.

5. Conclusions and Future Work

We have shown that a conjunctive global predicate in a system with n processes and a maximum of m local states can be found in $O(\log mn)$ parallel time on a CRCW PRAM. This also allows the parallel detection of a disjunctive normal form predicate. Furthermore, the complexity of computing a slice of any conjunctive predicate is also in **NC**.

Our parallel algorithm has optimal time complexity but high work complexity. We do not know if there exists an algorithm with lower work complexity. Linear predicates [3]

and regular predicates [9] generalize conjunctive predicates. Finding the parallel complexity of detecting these predicates is a future work. Recently, it was shown that the problem of stable marriage can be formulated as detecting a *linear* predicate in a computation [14]. Clearly, if detecting any linear predicate is in **NC**, then the stable marriage problem would also be in **NC**. The problem of finding the parallel complexity of the stable marriage problem has been open for many decades and any progress in finding the parallel complexity of linear predicates is important.

We have given parallel algorithms for data race detection with different time complexities and work complexities. Finding work-optimal parallel algorithms is a future work.

Another class of predicates for which there exists efficient polynomial time algorithms are relational predicates [2]. These predicates are of the form $\sum_i x_i \geq k$ where each x_i is on a different process. We do not know if the complexity of detecting relational predicates is in **NC**.

References

[1] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, January 1995.

[2] C. Chase and V. K. Garg. On Techniques and their Limitations for the Global Predicate Detection Problem. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[3] C. Chase and V. K. Garg. Efficient detection of global predicates in a distributed system. *Distributed Computing*, 11(4), 1998.

[4] 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.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

[6] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[7] 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.

[8] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, BC, Canada, June 1995.

[9] V. K. Garg and N. Mittal. On slicing a distributed computation. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)*, pages 322–329, Washington - Brussels - Tokyo, April 2001. IEEE.

[10] V. K. Garg and B. Waldecker. Detection of unstable predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991. ACM/ONR.

[11] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[12] Vijay K. Garg. *Elements of distributed computing*. Wiley, 2002.

[13] Vijay K Garg. *Lattice Theory with Computer Science Applications*. Wiley, New York, NY, 2015.

[14] Vijay K. Garg. Brief announcement: Application of predicate detection to the stable marriage problem. In *International Symposium on DIStributed Computing (DISC'17)*, 2017.

[15] Michael Hurfin, Masaaki Mizuno, Michel Raynal, and Mukesh Singhal. Efficient distributed detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, August 1998.

[16] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[19] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1542476.1542491, doi:10.1145/1542476.1542491.

[20] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[21] N. Mittal and V. K. Garg. Slicing a distributed computation: Techniques and theory. In *5th International Symposium on DIStributed Computing (DISC'01)*, October 2001.

[22] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/781498.781528, doi:10.1145/781498.781528.

[23] Vinit A. Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, pages 420–434, 2007.

[24] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/781498.781529, doi:10.1145/781498.781529.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997. URL: http://doi.acm.org/10.1145/265924.265927, doi:10.1145/265924.265927.

[26] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th International Workshop on Distributed Algorithms*, pages 318–332, Le Mont-Saint-Michel, France, September 1995. Springer-Verlag.

[27] A. I. Tomlinson and V. K. Garg. Monitoring Functions on Global States of Distributed Programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, March 1997.

[28] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *Thirtieth Annual Allerton Conference on Communication, Control and Computing*, pages 137–146, Allerton, IL, October 1992.

[29] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1095810.1095832, doi:10.1145/1095810.1095832.