Techniques to Tackle State Explosion in Global Predicate Detection*

Sridhar Alagar and S. Venkatesan
Department of Computer Science, EC 31
University of Texas at Dallas
Richardson, TX 75083
{sridhar,venky}@utdallas.edu

Abstract

Detecting properties about distributed program is an important problem in testing and debugging distributed programs. This problem is very hard due to the combinatorial explosion of the global state space. For a given execution, we consider the problem of detecting whether a predicate Φ is true at some global state of the system. First, we present a space efficient algorithm for detecting Φ. Next, we present a parallel algorithm to reduce the time taken to detect Φ. We then improve the performance of our algorithms, both in space and time, by increasing the granularity of the execution step from an event to a sequence of events in a process.

1 Introduction

Properties about programs such as safety and liveness can be expressed in terms of predicates. So, a technique to detect whether a predicate is true at a system state, which occurred in a particular execution, is important for testing and debugging distributed programs. However, due to the inherent asynchrony of the distributed systems, processes (by themselves) cannot ascertain that the global state (at which a given global predicate Φ is true) actually occurred during the execution. Cooper and Marzullo [4] describe two interpretations, *Possibly(Φ)* and *Definitely(Φ)*, for detecting Φ. The definitions of Possibly(Φ) and Definitely(Φ)are as follows [4].

Possibly(Φ): There is an execution of the program consistent with its observed behavior such that Φ was true at a point in that execution.

Definitely(Φ): For all executions of the program consistent with its observed behavior, Φ was true at some point.

Cooper and Marzullo [4] present centralized algorithms for detecting Possibly(Φ) and Definitely(Φ) for any arbitrary predicate Φ. In this paper, we refer to these algorithms as CM algorithms. The space and time complexities of CM algorithms are exponential in the number of processes. Several researchers have presented polynomial time algorithms for detecting global predicates by placing restrictions on the type of predicates [1, 6, 8, 9, 13]. The CM algorithms are important because (1) existing polynomial time algorithms are for restricted forms of predicates, and (2) the polynomial time algorithms are different for different predicates. In contrast, the CM algorithms may be used for any arbitrary predicate. It appears that detecting an arbitrary global predicate involves exhaustive checking of every global state.

A natural question to ask now is, "can we reduce the space complexities of the algorithms for detecting Possibly(Φ) and Definitely(Φ) for an arbitrary predicate?" In this paper, we present an algorithm for Possibly(Φ) that uses $O(mn)$ space where m is the total number of events in the computation and n is the number of processes in the system. We assume that the computation terminates eventually. It remains to be seen whether one can provide a space efficient algorithm for Definitely(Φ) also.

We then parallelize our space efficient algorithm for detecting Possibly(Φ). Considering the explosive nature of global state space, a parallel algorithm can significantly reduce the time taken to detect Possibly(Φ). A salient feature of our parallel algorithm is that the processors that test global states do not communicate (through messages or shared memory) to determine who has to test a particular global state. Our experimental results show that the speedup of our algorithm is close to the optimal value for the experiments per-

*This research was supported in part by NSF under Grant No. CCR-9110177 and by the Texas Advanced Technology Program under Grant No. 9741-036.

0-8186-6555-6/94 $04.00 © 1994 IEEE

formed.

We further improve the performance, both in time and space, of our algorithms by increasing the granularity of an execution step from an event to a sequence of events (*interval*). Instead of checking every global state, we check every *global interval*. (For the definition of global interval see Section 5.) When the value of the variables related to the global predicates are not changed "frequently," the number of global intervals can be substantially less than the number of global states, thereby reducing the space and time requirements of our algorithms.

2 Preliminaries

A distributed system is a collection of n processes labeled $P_1, \ldots, P_n$. Processes are connected by point to point logical channels. Processes and channels are asynchronous. We assume that processors (on which processes execute) and channels are fault free. Processes communicate by message passing only.

A process is a collection of events that form a total order. An event in a process is an action that changes the state of the process. An event may be a send event resulting in sending of a message to other processes, a receive event resulting in receipt of a message from a process, or an internal event. We use Lamport's partial order *happened before*, denoted by $\rightarrow$, to express the causality between any two events [7]. Let E be the set of all events in a particular execution. Then $(E, \rightarrow)$ is a partially ordered set. A run of a distributed program can be represented by a space-time diagram. The space-time diagram for a sample run is shown in Figure 1(a).

A *global state* of a distributed system is a *consistent* collection of local states of the processes [3]. A *consistent cut* C is a finite subset of E such that $e \in C$ and $e' \rightarrow e$ implies $e' \in C$. For every consistent cut there is a corresponding consistent global state (the state of the system after executing all the events in the cut).

The set of all global states of a run forms a *lattice* [11]. Two nodes (global states) of a lattice are connected by an edge, if the system can proceed from one state to the other by executing one event. A path from the initial global state to the final global state in the lattice is an observable computation of the system. The lattice for the sample run of Figure 1(a) is shown in Figure 1(b).

A global state S is represented as $< k_1, \ldots, k_n >$ such that, for process P_i, its state after executing k_i events is included in S.
For a global state $S =< k_1, \ldots, k_n >$,

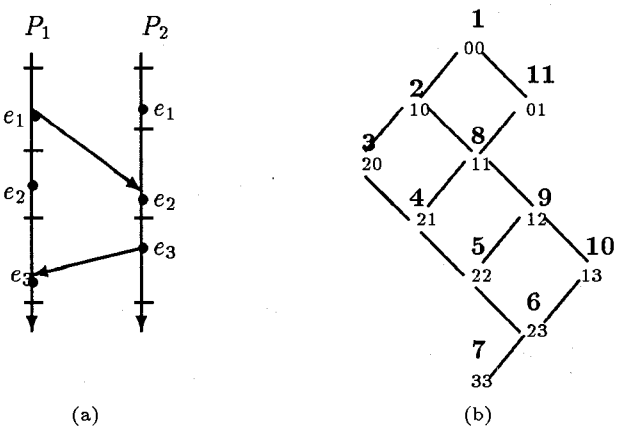

Figure 1: (a): A sample execution. (b): The lattice for the sample execution.

$$value(S) = (k_1, k_2, \ldots, k_n)$$

For a set of global states GS,

$$value(GS) = \{value(S) \mid S \in GS\}$$

For any two global states $S =< k_1, \ldots, k_n >$ and $S' =< k'_1, \ldots, k'_n >$, $value(S) > value(S')$ if there exists a j such that $k_j > k'_j$ and $k_i = k'_i$ for all $i : 1 \leq i < j$.
Function *pred*, the predecessor function, for a global state S is defined below.

$pred(S) = \{S' \mid$ system may change from global state S' to S by executing one event$\}$

Function *succ*, the successor function, for a global state S is defined below.

$succ(S) = \{S' \mid$ system may change from global state S to S' by executing one event$\}$

For a global state S, its *pred* and *succ* can be computed easily by using vector clocks [5, 11]. Every process maintains a vector clock consisting of n components. Let V_i be the vector clock of process P_i. Process P_i increments the i^{th} component of its vector clock, $V_i[i]$, whenever P_i executes an event. When a process sends a message, it timestamps the message with the current value of its vector clock. When P_i receives a message with timestamp T, $V_i[j] = max(V_i[j], T[j])$ for all j. The timestamp of an event e is denoted as $TS(e)$. For an event e in P_i, $TS(e)$ is the value of the updated vector clock V_i when e is executed. We say that two events e in P_i and e' in P_j are *consistent* if $TS(e)[j] \leq TS(e')[j]$ and $TS(e')[i] \leq TS(e)[i]$. A

global state $< k_1, \ldots, k_n >$ is consistent if, for all i, j, k_i^{th} event in process P_i and k_j^{th} event in process P_j are consistent.

3 Algorithm for Possibly(Φ)

The CM algorithm for Possibly(Φ) constructs the lattice of the execution. The number of nodes in the lattice can be exponential in the number of processes. The depth of the lattice is the total number of events in the run. The average breadth of the lattice can be exponential in the number of processes, even when the total number of events in an execution is polynomial in the number of processes. The *level* of a global state $< k_1, \ldots, k_n >$ in a lattice is the sum $k_1 + k_2 + \ldots + k_n$. The CM algorithm for Possibly(Φ) traverses the lattice in a breadth first fashion. It computes all the global states in one level, stores and checks them and then proceeds to the next level. Hence the space required (to store nodes of one level) in the worst case can be exponential in n.

In contrast, our algorithm traverses the lattice in a depth first fashion. We do not store the nodes that are already visited as this will require exponential space. Also, we do not visit a node more than once. We perform some computation to decide whether a node has already been visited or not.

Main Idea: Assume that we are in global state S in the lattice. Let S' be a successor of S in the lattice. Now we have to decide whether S' can be visited (tested) from S. Global state S' has at most n predecessors in the lattice. All predecessors of S' can be ordered according to their values. If the value of S is the maximum among values all the predecessors of S', then we visit S' from S. The *key rule* for testing a global state exactly once is to visit the global state only from its predecessor that has the maximum value among all its predecessors.

Our algorithm for Possibly(Φ) is executed by a *monitor* process. During the computation, all the processes, after executing an event, send the timestamp of the event and the value of the local variables related to Φ to the monitor. A formal description of the algorithm for Possibly(Φ) appears in Figure 2.

Algorithm Possibly sends a global state S. First, Φ is tested at S. If Φ holds, then the algorithm returns true. Otherwise $succ(S)$ is computed. For every successor S' of S such that S is the predecessor of S' with the maximum value, algorithm Possibly(S') is recursively invoked. Initially, the monitor invokes the algorithm with the initial global state, $< 0, \ldots, 0 >$.

```
Algorithm Possibly(S)
(1)  if Φ is true at S then
(2)    return(true);
(3)  Let S =< k_1, ..., k_i, ..., k_n >
(4)  i := 1;
(5)  while (i ≤ n) do
     begin
(6)    S' =< k_1, ..., k_i + 1, ..., k_n >
(7)    if S' is a consistent global state then
(8)      if value(S) = max(value(pred(S'))) then
(9)        if (Possibly(S') = true) then
(10)         return(true);
(11)   i = i + 1;
     end;
(12) return(false);
```

Figure 2: Algorithm for detecting Possibly(Φ)

Before invoking the algorithm Possibly with a global state S, if the monitor has not received the values of the variables corresponding to the state S, it waits for them and then invokes Possibly with S. If the algorithm returns true, then Possibly(Φ) is true.

We illustrate the working of our algorithm with an example shown in Figure 1(b). The numbers in bold show the order in which the lattice is traversed. The algorithm is invoked with the initial global state $< 0, 0 >$. The value of i is 1. After step 6 $S' =< 1, 0 >$. Since, $< 1, 0 >$ is a consistent global state and $max(value(pred(< 1, 0 >)))$ is $< 0, 0 >$, Possibly($< 1, 0 >$) is invoked. To understand the algorithm further, assume that algorithm Possibly is invoked with state $< 0, 1 >$. After executing step 6 once, $S' =< 1, 1 >$. The set $pred(< 1, 1 >)$ is $\{< 1, 0 >, < 0, 1 >\}$, and $max(value(pred(< 1, 1 >)))$ is $< 1, 0 >$ and not $< 0, 1 >$ (step 8). So, Possibly $< 1, 1 >$ is not invoked from $< 0, 1 >$. When step 6 is executed for the second time, S' becomes $< 1, 2 >$. But $< 1, 2 >$ is not a consistent global state, and since there are no more successors of $< 0, 1 >$, the algorithm returns.

3.1 Correctness and Analysis

Theorem 1 *Possibly(Φ) is true if and only if algorithm Possibly returns true when invoked with the initial global state.*

Theorem 2 *The space complexity of Algorithm Possibly is $O(mn)$ where m is the total number of events*

in the given run and n is the number of processes.

The space complexity of our algorithm can further be reduced to $O(m)$ without using vector clocks. For details refer to [2].

We performed some experimental studies and their detailed results are in [2]. With 12 processes and 1200 events (depth) the maximum size of breadth is 875201. So a depth first approach offers significant savings in storage than the breadth first approach. Also when using the breadth first approach, before inserting a global state in the queue, the queue has to be searched whether the global state is already present in the queue. Maintaining and searching the queue requires significant amount of time when the breadth of the lattice is large. One advantage of breadth first approach is that Possibly(Φ) can be detected early if Φ is true at a global state at the top of the lattice. However if Φ is true at a global state that occurs near the bottom of the lattice, depth first approach detects Φ early as breadth first approach will be spending significant time at the lower levels.

4 A Parallel Algorithm

In this section, we parallelize the algorithm presented in the previous section to reduce the time taken to detect Possibly(Φ). We assume that a parallel machine consisting of t processors is available. Our algorithm is independent of the architecture of the parallel machine. In fact the distributed system on which the computation was performed can also be used.

To ease the description of the algorithm, we assume that the input (timestamps of all the events and the value of variables related to Φ after every event) is available at all the processors. Later, we discuss various ways of providing input to all the processors.

In our parallel algorithm, every global state is tested exactly by one processor. A salient feature of our algorithm is that processors do not exchange messages to decide whether they can test a particular global state or not. We achieve this by using the same idea used in the previous section to reduce the space complexity of algorithm Possibly.

A global state will be tested only by a processor that tested the largest predecessor (which is unique) of the global state. If a processor, say $\mathcal{P}r_i$, finds some idle processor, $\mathcal{P}r_i$ will delegate one of the global states that it has to test to the idle processor.

One of the processors will also act as a processor *allocator*. A processor that becomes idle will register itself with the allocator. When a processor has more than one global state to test, it will request the allocator for an idle processor. The allocator will serve the requests in the first come first serve basis. Initially, processor $\mathcal{P}r_0$ is assigned the initial global state, and all other processors are idle.

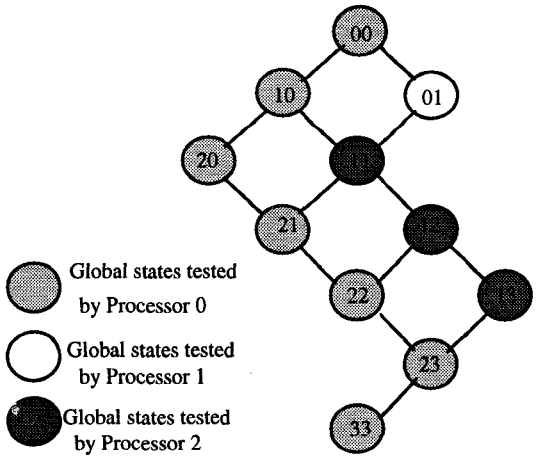

Figure 3: No of processors = 3

Now we illustrate our algorithm with the example shown in Figure 3. The number of processors used is 3. Processor $\mathcal{P}r_0$ is assigned the global state $< 0,0 >$ initially. Since the global state $< 0,0 >$ is the largest predecessor of $< 1,0 >$ and $< 0,1 >$ processor $\mathcal{P}r_0$ has to test them. However, processors $\mathcal{P}r_1$ and $\mathcal{P}r_2$ are idle. So, $\mathcal{P}r_0$ assigns state $< 0,1 >$ to $\mathcal{P}r_1$ and tests state $< 1,0 >$. $\mathcal{P}r_0$ after testing $< 1,0 >$ considers the states $< 2,0 >$ and $< 1,1 >$. Since $< 1,0 >$ is the largest predecessor of the states $< 2,0 >$ and $< 1,1 >$, it is processor $\mathcal{P}r_0$'s responsibility to test them. At this stage, $\mathcal{P}r_2$ is idle, so state $< 1,1 >$ is assigned to $\mathcal{P}r_2$. $\mathcal{P}r_2$ tests $< 2,0 >$.

$\mathcal{P}r_1$ after testing $< 0,1 >$ looks at $< 1,1 >$ and finds that $< 0,1 >$ is not the largest predecessor of $< 1,1 >$, so it does not test $< 1,1 >$. $\mathcal{P}r_2$ informs the allocator that it is idle as it does not have any other global state to test. Proceeding in a similar fashion, $\mathcal{P}r_0$ tests states $< 2,1 >$, $< 2,2 >$, $< 2,3 >$ and $\mathcal{P}r_2$ tests $< 1,2 >$, and $< 1,3 >$.

A formal description of the algorithm is shown in Figure 4.

An idle processor, say $\mathcal{P}r_i$, begins when it receives a *start*(S) message. (S is a global state.) $\mathcal{P}r_i$ first tests Φ at S. If Φ is true at S, it notifies all the processors and exits. Otherwise $\mathcal{P}r_i$ finds the list of all the successors of S for which S is the largest predecessor and stores them in a FIFO queue Q_i. Processor $\mathcal{P}r_i$

On receiving $start(GS)$, execute the following steps.
$S = GS$;
$done = false$;
repeat
count = 0;
if Φ is true at S, notify all the processors and exit;
for all S' such that S' is a successor of S
if $value(S) = max(value(pred(S')))$ then
add S' to the tail of Q_i;
$count = count + 1$;
if Q_i is empty **then** $done = true$
else
S = head of Q_i;
send $request(count - 1)$ message to the allocator;
delete the first element of Q_i;
until($done$)
inform the *allocator* that $\mathcal{P}r_i$ is idle.

On receiving $allocated(\mathcal{P}r_j)$ from the allocator, execute the following steps.
S' = head of Q_i;
delete head of Q_i;
Send $start(S')$ to processor $\mathcal{P}r_j$;

Figure 4: Algorithm executed by processor $\mathcal{P}r_i$

then requests the allocator for any idle processor by sending $request(x)$ message to the allocator where x is the number of processors that $\mathcal{P}r_i$ needs. $\mathcal{P}r_i$ then takes the first element in Q_i and repeats the steps in the loop.

In the meanwhile, if $\mathcal{P}r_i$ receives an $allocated(\mathcal{P}r_j)$ message, $\mathcal{P}r_i$ sends $start(S')$ to $\mathcal{P}r_j$ where S' is the first element in Q_i. When Q_i becomes empty, $\mathcal{P}r_i$ informs the allocator that is idle and waits for a *start* message.

If a parallel machine is dedicated for testing Possibly(Φ), one way to distribute the input is to send the input to all the processors. Another approach is to write the input in a shared memory and the processors can get the input from the shared memory when they require it. If the existing distributed system itself is being used for testing, the input can be provided separately to all the processors or the input can be stored in a distributed shared memory.

We evaluated the performance of our parallel algorithm by a simulation. The speedup achieved is close to the optimal value for the experiments we have performed. For details of the results refer to [2].

5 Increasing the granularity of execution step

The performance of our algorithms can be improved by considering a sequence of consecutive events instead of a single event when detecting Φ. The value of a local variable related to Φ may not change during every event. A consecutive sequence of states in a process in which the value of the local variables related to Φ remains unchanged can be considered as identical states with respect to Φ. An *interval* is a maximal sequence of events such that the values of the local variables related to Φ are the same after the occurrence of every event in the sequence. A process *begins a new interval* if an event changes the value of a local variable related to Φ.

Process P_i maintains an *interval clock* V_i consisting of n components. Process P_i increments the i^{th} component of V_i whenever it begins a new interval. When a process sends a message, it timestamps the message with the current value of its interval clock. When P_i receives a message with timestamp T, $V_i[j] = max(V_i[j], T[j])$ for all j. The timestamp of interval I_i is denoted by $TS(I_i)$. For an interval I_i in P_i, $TS(I_i)$ is the value of the updated interval clock V_i when the first event of the interval I_i occurred.

We say that two intervals I_i and I_j are *consistent* if $TS(I_i)[j] \leq TS(I_j)[j]$ and $TS(I_j)[i] \leq TS(I_i)[i]$. A *global interval* is a collection of intervals with one interval from every process such that the intervals are pairwise consistent. A global interval $< I_1, \ldots, I_n >$ is consistent if I_i and I_j are consistent for all i, j. The set of all global intervals forms a lattice. A node in a lattice is a global interval, and there is an edge between global intervals I_i to I_j if the computation can proceed from one global interval to another by executing a sequence of events (interval) in a process. We claim that it is sufficient to check all the global intervals instead of all the global states to detect Possibly(Φ).

Theorem 3 *There exist a consistent global interval at which Φ is true if and only if there exists a consistent cut (global state) at which Φ is true.*

When a process begins a new interval, it sends the timestamp of the interval and the value of the local variable related to Φ to the monitor process. Algorithm Possibly described in Figure 2 can be used to detect Possibly(Φ). The *successor* and *predecessor* functions can be computed using the timestamps of the intervals. In a process, the number of intervals may be considerably less than the number of events if every event does not change the value of the local

variable. Therefore, the total number of global intervals can be substantially less than the total number of global states, improving the performance of the algorithm to detect Possibly(Φ) both in space and time.

Our concept of interval clock is an extension of weak vector clock used by Marzullo and Neiger [10]. A process P_i increments its i^{th} component (terminates an interval) either when it executes an event that potentially changes Φ, or when it executes an receive event through which it perceives that another process has potentially changed Φ [10]. But in our technique, a process terminates an interval only when it executes an event that changes a value of a variable related to Φ. A process P_i may not change the variable related to Φ but it may receive messages from a process P_j that frequently changes its variable related to Φ. Hence, the number of global intervals in our case may be considerably less than the number of global states obtained by using weak vector clocks.

6 Conclusion

The concept of *event occurrence* condition introduced by Spezialetti [12] was formulated as Possibly(Φ) by Cooper and Marzullo [4]. Several researchers – Manabe and Imase [9], Venkatesan and Dathan [13], Garg and Waldecker [6], and Alagar and Venkatesan [1] – present efficient algorithms for detecting Possibly(Φ) by restricting Φ.

In this paper, we have presented a space efficient algorithm for detecting Possibly(Φ) for any arbitrary predicate Φ. The space complexity of our algorithm is $O(mn)$ where m is the total number of events in the run and n is the number of processes. We have then presented a parallel algorithm to detect Possibly(Φ). Also, we have improved the performance of our algorithm, both in space and time, by increasing the granularity of an execution step from an event to a sequence of events.

References

[1] ALAGAR, S., AND VENKATESAN, S. Hierarchy in testing distributed programs. In *Proceedings of First International Workshop on Automated and Algorithmic Debugging* (1993), Springer Verlag.

[2] ALAGAR, S., AND VENKATESAN, S. Techniques to tackle state explosion in global predicate detection. Computer science technical report, The University of Texas at Dallas, 1994.

[3] CHANDY, K., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (1985), 63–75.

[4] COOPER, R., AND MARZULLO, K. Consistent detection of global predicates. *Sigplan Notices* (1991), 167–174.

[5] FIDGE, J. Timestamps in message passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference* (1988), pp. 55–66.

[6] GARG, V., AND WALDECKER, B. Detection of unstable predicates in distributed programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science* (1992), Springer Verlag.

[7] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[8] LI, H. F., AND DASH, B. Detection of safety violations in distributed systems. In *Proceedings of 1992 International Conference on Parallel and Distributed Systems* (1992), pp. 275–282.

[9] MANABE, Y., AND IMASE, M. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing* (1992), 62–69.

[10] MARZULLO, K., AND NEIGER, G. Detection of global state predicates. In *Distributed Algorithms Proceedings of 3rd International Workshop*. Springer-Verlag, 1991, pp. 254–272.

[11] MATTERN, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. Cosnard et. al., Ed. Elsevier Science Publishers B. V., 1989, pp. 215–226.

[12] SPEZIALETTI, M. *A generalized approach to monitoring distributed computations for event occurrences.* PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania, 1989.

[13] VENKATESAN, S., AND DATHAN, B. Testing and debugging distributed programs distributively. In *Proceedings of Thirtieth Annual Allerton Conference on Communication, Control and Computing* (1992).