Distributed Algorithms for Detecting Conjunctive Predicates

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Vijay K. Garg* Craig M. Chase†

September 30, 1994

Abstract

This paper discusses efficient distributed detection of global conjunctive predicates in a distributed program. Previous work in detection of such predicates is based on a checker process. The checker process requires $O(n^2m)$ time and space where m is the number of messages sent or received by any process and n is the number of processes over which the predicate is defined. In this paper, we introduce token-based algorithms which distribute the computation and space requirements of the detection procedure. The distributed algorithm has $O(n^2m)$ time, space and message complexity, distributed such that each process performs $O(nm)$ work. We describe another distributed algorithm with $O(Nm)$ total work, where N is the total number of processes in the system. The relative values of n and N determine which algorithm is more efficient for a specific application.

1 Introduction

Detection of a global predicate is a fundamental problem in distributed computing. This problem arises in many contexts such as designing, testing and debugging of distributed programs. Previous work has described algorithms for detecting stable and unstable global predicates [2, 3, 5, 7, 6, 10, 11, 13]. See [1, 12] for surveys of stable and unstable predicate detection.

This paper is most closely related to [7], [6], and [3]. Garg and Waldecker [7] present a method for detecting weak conjunctive predicates (WCP). A WCP is a global predicate formed as the conjunction of n local predicates. Many problems in distributed systems can be viewed as special cases of WCP detection. In [7], each process checks for its own local predicate and sends messages to a global predicate checker that detects when all predicates are true in a consistent cut. [7] also presents a method of decentralizing the algorithm. The main idea is that the

*supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

†supported in part by the Texas Instruments/Jack Kilby Faculty Fellowship

set of processes are divided into groups. The checker process for the group is responsible for sending to the overall checker process the set of all global states which are consistent within the group. The overall checker process then checks consistency across groups. This technique suffers from the disadvantage that the group checker process may have to send an exponential number (exponential in the number of processes in the group) of global states to the overall checker process. The algorithm presented in this paper avoids this problem.

Garg, Chase, Mitchell and Kilgore [6] extend the algorithm for detecting a WCP to include predicates based on the state of communication channels. This larger class of predicates are termed Generalized Conjunctive Predicates (GCP). A centralized checker process is also used in [6].

Cooper and Marzullo [3] discuss detection of any global predicate. Their method relies on constructing the lattice of global states and searching it to see if the global predicate is satisfied by one of the states. This method also requires a centralized checker process.

In this paper, we present distributed algorithms to detect conjunctive predicates. We present two algorithms. The first algorithm is based on vector clocks similar to [7]. However, instead of a centralized process we use a token which carries in it the candidate global state and information sufficient to determine if the global state satisfies the WCP. This algorithm requires $O(n^2m)$ total work and $O(nm)$ work per process where m is the number of messages sent or received by any process and n is the number of processes over which the predicate is defined. We also discuss how this algorithm can be generalized to a parallel algorithm for WCP with g tokens. The second algorithm does not use vector clocks. Instead it is based on finding all direct dependences between processes. This algorithm requires $O(Nm)$ work where N is the total number of processes in the system. The relative values of n and N determine which algorithm is more efficient for a specific application.

This paper is organized as follows. Section 2 describes our model of distributed computation and monitoring of a global predicate. In Section 3, we present the vector clock based algorithm, and in Section 4, we discuss the direct dependence based algorithm. Section 5 provides a lower bound for the problem of detection of a conjunctive predicate. In this section, we show that any parallel or distributed algorithm takes at least $\Omega(nm)$ steps to determine if a conjunctive predicate became true in a distributed computation.

2 Our Model

This section presents the concepts of local and global predicates. The reader is referred to [7, 6] for more comprehensive background material.

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of N processes denoted by $\{P_1,P_2,...,P_N\}$ communicating via asynchronous messages. In this paper, we will be concerned with a single run of a distributed program. We assume that no messages are lost, altered or spuriously introduced. We do not make any assumptions about a FIFO nature of the channels. We use a happened-before relation, '$\rightarrow$,' between states similar to that of Lamport's happened-before relation between events [8]. The happened-before relation can be formally stated as: $\alpha \rightarrow \beta$ iff: α occurs before β in the same process, or the action following α is a send of a message and the action preceding β is a receive of that message, or $\exists\gamma : \alpha \rightarrow \gamma \wedge \gamma \rightarrow \beta$. Two states for which the happened-before relation does not hold in either direction are said to be concurrent. The concurrency relation, $\|$, can be formally stated as:

$$\alpha \parallel \beta \Leftrightarrow (\alpha \not\rightarrow \beta \ \wedge \ \beta \not\rightarrow \alpha)$$

A set of states is called a *consistent cut* if all states are pairwise concurrent.

A *local state* is the value of all program variables and processor registers (including the program counter) for a single process. The execution of a process can thus be viewed as a sequence of local states. A *local predicate* is defined as any boolean-valued formula on a local state. For any process, represented by P_i, a local predicate is written as l_i. $l_i(\alpha)$ is used to represent the predicate being true in a particular state, α, of P_i. A process can obviously detect a local predicate on its own.

A global state is a collection of local states such that exactly one state is included from each process. A global predicate is a boolean-valued formula on a global state. Global predicates can only be true if the global state is a consistent cut. A global predicate formed only by the conjunction of local predicates is called a Weak Conjunctive Predicate (WCP). We restrict our consideration to conjunctive predicates because any boolean predicate can be detected using an algorithm that detects conjunctive predicates [7].

Following are some examples of the WCP formulas:

1. Suppose we are developing a mutual exclusion algorithm for two processes. Let CS_i represent the local predicate that the process P_i is in critical section. Then, detecting $(CS_1 \wedge CS_2)$ is

equivalent to detecting violation of mutual exclusion for a particular run:

2. Assume that in a database application, serializability is enforced using a two phase locking scheme Further assume that there are two types of locks: *read* and *write*. Then, detecting $(P_1\ has\ read\ lock) \wedge (P_2\ has\ write\ lock)$ is useful in identifying an error in implementation.

2.1 Application and Monitor Processes

A distributed program consists of a set of N interacting *application processes*. Collectively, these processes perform some useful function, e.g. implement a distributed database. Our goal is to detect the occurrence of global conditions of these processes. We introduce a new set of N *monitor processes*. One monitor process is mated to each application process. Our model is illustrated in Figure 1. The distributed program is illustrated by the plane labeled "Application Domain". The application processes (labeled "AP") interact according to the distributed application. In addition, the application processes send local snapshots (described below) to monitor processes (labeled "MP"). The monitor processes interact with each other, but do not communicate to the application processes.

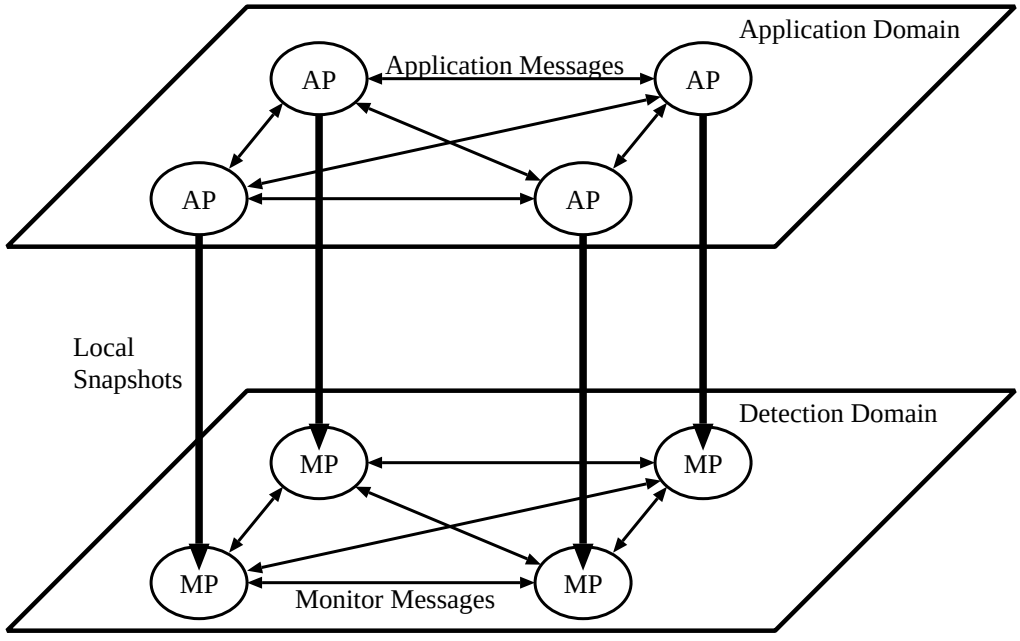

Figure 1: Application and Monitor Processes

3 A Vector Clock Based WCP Detection Algorithm

In this section we describe a vector clock based, distributed algorithm for detecting a WCP. The algorithm has $O(n^2m)$ time, space and message complexity where the WCP is defined over n ($n \leq N$) application processes and each application process sends at most m messages. We describe the behavior of both application and monitor processes. Each process is required to perform at most $O(nm)$ work.

3.1 Application Processes

Since a WCP requires that local predicates are true on every process, we can disregard all local states in which the local predicate is false. To uniquely identify a state, we use a vector clock [9, 4]. We use the notation (i, k) to represent the kth state on application process P_i. We will also use simply k to represent the this state when the identity of process P_i is obvious from context.

The following properties of vector clocks are easily verified.

1. $\alpha \rightarrow \beta$ iff $\alpha.v < \beta.v$, where α and β are states in processes P_i and P_j and $\alpha.v$ and $\beta.v$ are the respective vector clocks at these states.
2. Let v be a vector on P_i. Then, for any j different from i, $(j, v[j]) \rightarrow (i, v[i])$.

Each application processes checks for its local predicate. It sends a message, called a *local snapshot*, to its monitor process whenever the local predicate becomes true for the first time since the last program message was sent or received. The message contains *vclock*, the current vector clock. An algorithm for this process is given in Figure 2. Note that we require communication between an application process and a monitor process be FIFO.

3.2 Monitor Processes for single token algorithm

The distributed WCP detection algorithm uses a unique token. The token contains two vectors. The first vector is labeled G. This vector defines the current candidate cut. If *G[i]* has the value k, then state k from process P_i is part of the current candidate cut. Note that all states on the candidate cut satisfy local predicates. However, the states may not be mutually concurrent (i.e. the candidate cut may not be a consistent cut). The token is initialized with $\forall i : G[i] = 0$.

The second vector is labeled *color*, where *color[i]* indicates the color for the candidate state from application process P_i. The color of a state can be either red or green. If *color[i]* = red

```
P_i:
var
vclock: array [1..n] of integer
        init ∀j : j ≠ i :vclock[j] = 0; vclock[i] = 1;
firstflag : boolean init true;

For sending msg do
        msg.vclock := vclock;
        send msg;
        vclock[i]++ ;
        firstflag:=true;

Upon receive msg do
        ∀j :vclock[j]:=max(vclock[j], msg.vclock[j]);
        vclock[i]++ ;
        firstflag:=true;

Upon (local_pred = true)∧ firstflag do
        firstflag := false;
        send (vclock) to monitor process M_i;
```

Figure 2: Application Process Algorithm

then the state $(i, G[i])$ and all its predecessors have been eliminated and can never satisfy the WCP. If $color[i] = green$, then there is no state in G such that $(i, G[i])$ happened before that state. The token is initialized with $\forall i : color[i] = red$.

The token is sent to monitor process M_i only when *color[i]* = red. When it receives the token, M_i waits to receive a new candidate state from P_i and then checks for violations of consistency conditions with this new candidate. This activity is repeated until the candidate state did not happen before any other state on the candidate cut (i.e. the candidate can be labeled green). Next, M_i examines the token to see if any other states violate concurrency. If it finds any j such that $(j, G[j])$ happened before $(i, G[i])$, then it makes $color[j]$ red. Finally, if all states in G are green, that is the cut, G, is consistent, then M_i has detected the WCP. Otherwise, M_i sends the token to a process whose color is red. This behavior is described in Figure 3. Note that the token can start on any process. Since the entire color vector is initialized to red, it must eventually visit every process at least once.

3.3 Correctness of Vector Clock Based Algorithm

We now show that our algorithm correctly detects the first cut that satisfies a WCP.

Lemma 3.1 *For any i,*

1. $G[i] \neq 0 \wedge color[i] = red \Rightarrow \exists j : j \neq i : (i, G[i]) \rightarrow (j, G[j])$;

2. $color[i] = green \Rightarrow \forall k : (i, G[i]) \not\rightarrow (k, G[k])$;

```
var
candidate:array[1..n] of integer; /* vector clock from candidate state */

on receiving the token (G,color)
      while (color[i] = red) do
            receive candidate from application process P_i
            if (candidate[i] > G[i]) then
                  G[i] := candidate[i]; color[i]:=green;
      endwhile
      for j ≠ i:
            if (candidate[j] ≥ G[j]) then
                  G[j] := candidate[j];
                  color[j]:=red;
            endif
      endfor
      if (∃ j: color[j] = red) then send token to M_j
      else detect := true;
```

Figure 3: Monitor Process Algorithm

3. $(color[i] = green) \wedge (color[j] = green) \Rightarrow (i, G[i]) \| (j, G[j])$.

4. If $(color[i] = red)$*, then there is no global cut satisfying the WCP which includes* $(i, G[i])$.

Proof: 1. Initially, $G[i] = 0$, so the lemma holds. For any i, $color[i]$ is set to red only when some process P_j has the token and $G[i] \leq candidate[i]$. By the property of vector clocks, for any $candidate$ in P_j, $candidate[i] \rightarrow candidate[j]$. Therefore, $G[i] \rightarrow G[j]$. Whenever a new candidate is selected for P_i, the color of P_i is set to green, hence the lemma holds.

2. We show the contrapositive of the statement. That is, $\exists k : G[i] \rightarrow G[k] \Rightarrow color[i] = red$. Initially $color[i]$ is red and the lemma holds. Assume that the lemma holds for any cut. We show that on advancing the cut, the lemma continues to hold. The antecedent can be made true only by advancing the cut on process k. Let $G[i] \rightarrow G[k]$. This implies that the vector $M_k.candidate$ which satisfies $M_k.candidate[k] = G[k]$ also satisfies $M_k.candidate[i] \geq G[i]$ (by the property of vector clocks). Therefore, $color[i]$ is changed to red.

3. Follows easily from part 2.

4. We show that there is no global cut satisfying the WCP with $G[i]$. The proof is by induction. Initially $G[i]$ is 0 and the assertion holds since there cannot be any consistent global cut including $(i, 0)$. From part 1, $(G[i] \neq 0) \wedge (color[i] = red)$ implies that it happened before some state $G[j]$. This implies that $G[i]$ cannot be consistent with $G[j]$ or any state in P_j after $G[j]$. By the induction hypothesis, it follows that G[i] cannot be consistent with any state preceding $G[j]$ because states preceding $G[j]$ are also colored red. Thus, there is no global cut which includes a state in P_j and is consistent with $G[i]$. □

Theorem 3.2 *The flag* detect *is true with G iff G is the first cut that satisfies the WCP.*

Proof: The condition *detect* implies that $\forall j : color[j] = green$. It follows from part 3 of Lemma 3.1 that $\forall i, j : i \neq j : G[i] \| G[j]$. We know that $l_i(G[i])$ always holds because only states where l_i is true can be candidates. Hence, the WCP holds for G. We now show that G is the first such cut. Note that for any process P_i, $G[i]$ is changed only when $color[i]$ is red. From lemma 3.1, part 4 it follows that no predecessor of any state, $G[i]$, can satisfy the WCP. Therefore, whenever detect is set to true, G is the first cut to satisfy the WCP.

Conversely, let H be the first cut for which the WCP is true. We first show that if G is the current candidate for the WCP and G is before H, then at least one of the states in G will be eliminated. That is, the detection algorithm will continue to make progress until it reaches H. It is easy to see that for any cut G before H, there exists $j : color[j] = red$. Since only process P_j can change $color[j]$ from red to green, the condition to send the token to P_j will stay true until P_j gets the token. Further, the token eventually is sent to P_j because when a token moves at least one state is removed from consideration and there are finite number of states in other processes after G and before H. Once the token reaches P_j, the state in P_j is advanced.

We now show that any state in H will not be eliminated. A state along P_i is eliminated only if it is red. From part 4 of Lemma 3.1 it follows that any eliminated state cannot be part of a consistent cut satisfying the WCP. Therefore, no state in H is eliminated. Eventually, all states from H will be part of the candidate cut and the algorithm will halt with detect true. □

3.4 Analysis of Vector Clock Based Algorithm

We first analyze the time complexity for computation. It is easy to see that whenever a process receives the token, it deletes at least one local state (i.e., it receives at least one message from the application process). Every time a state is eliminated, $O(n)$ work is performed by the process with the token. There are at most mn states; therefore, the total computation time for all processes is $O(n^2m)$. Note that this requirement is same as that for the centralized algorithm [7]. The main difference between the algorithms is that the work for any process in the distributed algorithm is at most $O(nm)$.

We now analyze the message complexity. Since there are at most mn states, the token is sent at most mn times. In addition, each monitor process receives at most m messages from its application process. Hence the total number of messages in the system is $2mn$. The size

of both the token and the candidate messages is $O(n)$. Therefore, the total number of bits communicated is $O(n^2m)$.

The main space requirements are the buffer for holding messages from the application process (the token is unique and only $O(n)$ in size). There are at most m such messages each of size $O(n)$. Therefore, $O(nm)$ space is required by the algorithm for every monitor process. We again observe that the checker-based algorithm requires $O(n^2m)$ space for the checker process.

3.5 Introducing Parallelism to the Vector Clock Based Algorithm

The main drawback of the single-token WCP detection algorithm is that it has no concurrency. A monitor process is active only if it has the token. This can be remedied by using multiple tokens in the system. We partition the set of monitor processes into g groups and use one token for each group. Let *group(i)* denote the group to which M_i belongs. The monitor processes in each group run the single-token algorithm, except that the token is not allowed to be sent to a process in a different group. Once there are no longer any red states from processes within the group, the token is returned to some pre-determined *leader* process (say P_0).

When P_0 has received all the tokens, it merges the information in the g tokens to identify a new global cut. Some processes may not satisfy the consistency condition for this new cut. If so, a token is sent into each group containing such a process. If P_0 determines that all groups satisfy the consistency condition, then the WCP is detected.

4 A Direct Dependence Based WCP Detection Algorithm

In this section we describe a WCP detection algorithm that does not use vector clocks. The algorithm is based upon satisfying only direct dependences between states. Hence, it is necessary for all N processes to participate in the algorithm. Recall that the vector clock based algorithm required participation by only the n processes for which local predicates are defined. The direct dependence based algorithm has $O(Nm)$ time, space and message complexity and is more efficient when n^2 is large relative to N.

4.1 Application Process

Each application process uses a logical counter to uniquely identify candidate states. The counter is incremented on each send or receive performed by the application process. The

counter is attached to each message sent between application processes. This contrasts with the previous algorithm, which attached a vector of n clock values to each message.

The format of a local snapshot is also changed from the previous algorithm. When application process P_i receives a message from process P_j that is tagged with clock k, it records (j, k) as a dependence. In other words, all successive states on process P_i depend on state k from process P_j. We refer to this dependence as a direct dependence since it is implied by a single message. A linked list of all direct dependences is constructed as messages are received. When a candidate state for the WCP is reached, a local snapshot is created from this dependence list and the current value of the local clock. The dependence list is reinitialized to be empty after generating the local snapshot.

4.2 Monitor Processes

We now describe the direct dependence algorithm for monitor processes. As before, the monitor processes interact by exchanging a token. However, in this algorithm the token is empty. All data structures, including the candidate cut are distributed. Each monitor process must keep track of the color for its candidate state (the variable *color)* and the clock value for this state (the variable *G)*. These variables are analogous to those from the vector clock based algorithm as shown in Table 1.

Vector Clock Algorithm	Direct Dependence Algorithm
token.*color*[i]	$M_i.color$
token.G[i]	$M_i.G$

Table 1: Distribution of Token Data Structures

In addition, the monitor processes maintain a pointer variable, called *next_red*, which is used to construct a null-terminated linked list that includes all monitor processes whose *color* is red. We call this list the red chain. We maintain the invariant that the process with the token is at the head of the red chain.

Monitor processes are active only when:

1. they have the token, or

2. they are polled by the process that has the token

```
var
color : { red, green } init red;
G : integer init 0;
deplist : list of (process id, clock);

upon receiving the token
deplist := ∅;
repeat
      receive candidate from application process
      deplist := deplist ∪ candidate.deplist;
until (candidate.clock > G)
color := green

/* add processes to red chain */
for (each (i, k) ∈ deplist) do
      send poll(k, next_red) to M_i;
      receive response from M_i;
      if (response = became_red) next_red := i;
endfor

if (next_red = NULL) detect := true;
else send token to M_next_red;
```

Figure 4: Monitor Process for Direct Dependence Algorithm

The monitor process with the token operates as shown in Figure 4. Note that only a monitor process whose *color* is red will have the token. The process receives new candidate states from the application process until a state is found with a local clock larger than G. As each new candidate state is received, the dependences are collected into a list. When a candidate has been found that has a clock larger than G, the dependence list is traversed as follows: For each dependence in the list, a poll message is sent to M_j where P_j is the source of the dependence. The poll message consists of two integers, the clock value for the dependence and *next_red*. If the response to the poll message is "became red", then M_i sets *next_red* to be the value j, adding M_j to the red chain. If the response is "no change" then the dependence did not cause M_j to become red. All dependences are processed in this manner. Then M_i sends the token to the process corresponding to *next_red*. If *next_red* is NULL, then there are no red processes and *detect* is set to true.

Upon receiving a poll message, the monitor process performs two functions as shown in Figure 5. First it updates the G and *color* variables as follows. If the dependence contained in the poll message has a value equal to or larger than G, then G is set to this value, and *color* is set to red. Second, the monitor process generates a response message as follows. If in the first step, *color* was not changed from its previous value, then reply with the message "no change". Otherwise set the local variable *next_red* to the value of *next_red* from the poll message and

```
upon receiving poll message from M_i:
old_color := color;
if (poll.clock ≥ G) then
      color := red;
      G := poll.clock;
endif
if (color = red ∧ old_color = green) then
      next_red := poll.next_red;
      send "became red" to M_i;
else
      send "no change" to M_i;
endif
```

Figure 5: Responding to Poll Messages

reply with the message "became red".

4.3 Correctness of the Direct Dependence Algorithm

In this section we show that the direct dependence algorithm detects the first cut for which the WCP is true. We first show that finding a global consistent cut with respect to indirect dependence is equivalent to finding a global consistent cut with respect to direct dependence when all processes are involved. Direct dependence, denoted $s \rightarrow_d t$, means that s and t are on the same process and s happens before t, or there exists a single message sent after s and received before t. Now, we can state the result that consistency can be checked with respect to direct dependence.

Lemma 4.1 *Let G be any global cut. Then, G is a consistent cut (i.e. $\forall i, j : G[i] \not\rightarrow G[j]$) iff $\forall i, j : G[i] \not\rightarrow_d G[j]$.*

Proof: See Appendix. □

It should be observed that we have used the fact that G includes a component from each of the N processes. Thus, this algorithm requires that all processes participate in the algorithm.

Lemma 4.2

1. *For any i : $M_i.G \neq 0 \wedge M_i.color = red \Rightarrow \exists j : (i, M_i.G) \rightarrow_d (j, M_j.G)$*
2. *$M_i.color = green \wedge M_j.color = green \Rightarrow (i, M_j.G) \not\rightarrow_d (j, M_j.G) \wedge (j, M_j.G) \not\rightarrow_d (i, M_i.G)$.*
3. *$M_i.color$ is red, iff M_i is on the red chain.*

Proof: See Appendix. □

Theorem 4.3 *If detect is set to true, then the cut defined by $M_i.G$ on all monitor processes M_i is the first consistent cut that satisfies the WCP.*

Proof: If detect is set to true then the red chain is empty. This is because only the process with the token can set detect. Since the process with the token is always the head of the red chain, this chain must be empty before detect can be made true. Therefore, by part 3 of Lemma 4.2 all of the *color* variables must be green. Hence, by part 2 of Lemma 4.2 the G variables form a consistent cut with respect to direct dependence. By Lemma 4.1, this cut satisfies the WCP. We now argue that this is the first such cut. Since a monitor process can only change G when *color* is red, Part 1 of Lemma 4.2 tells us that we cannot bypass any cut that satisfies the WCP. Hence if detect is set to true then this must be the first such cut. □

Theorem 4.4 *If cut H is the first consistent cut that satisfies the WCP, then detect will be set to true with H defined by the* G *variables on all M_i.*

Proof:

Initially, all M_i are at a state that is a predecessor to H. We must now show that they continue to make progress until they reach H. This requires two parts. First we must show that all red candidates are eventually replaced with green candidates. By Part 3 of Lemma 4.2 we can assert that the token will eventually visit any process with a red candidate. The token will remain on this process until a green candidate can be found. Second, we must show that any state which precedes H is correctly painted red. Assume (i, k) is a state that precedes H. For this to be true, there must have been some message sent after state k by P_i that has been received prior to H. Since all messages sent prior to any green state are included in the dependence list, any such predecessor will eventually be painted red. Hence, the monitor processes will eventually advance the cut to H. When this occurs, all processes must be green. By Lemma 4.2 part 3, the red chain must be empty and detect will be set to true. □

4.4 Analysis of the Direct Dependence Algorithm

At most mN dependences exist in the system. Hence, at most mN poll messages and replies can be sent. The token never visits processes with green candidates. Every time the token visits a process, at least one state is eliminated. Hence the token can be sent at most mN times.

Therefore, the total number of messages sent by monitor processes is $3mN$. At most mN local snapshots can be sent by application processes.

All monitor messages are of constant size. The token carries no actual information, and poll messages contain only two integers. Poll responses are 1 bit. Therefore, total number of bits sent by monitor processes is $O(Nm)$. Local snapshots are variable sized. However the total number of dependences for any one process cannot exceed m. Since a dependence can be represented by a pair of integers, $O(Nm)$ bits are sent by application processes.

Although a variable amount of work occurs each time the token visits a processor, the total work is bounded by $O(m)$ for each process. This is clearly seen by noting that a constant amount of work is performed for each dependence. This work includes adding and then removing the dependence from a list, sending a single poll message and waiting for a response. Since there can be no more than $O(m)$ dependences for any one process, the total work performed by all monitor processes is $O(Nm)$.

The amount of space required for each monitor process or application is determined by the space required for the dependence list. All other variables are of constant size. Since the dependence list is at most m elements long on any process, at most $O(m)$ space is required on any process.

4.5 Introducing Parallelism to the Direct Dependence Based Algorithm

We can introduce concurrent activity into the direct dependence based algorithm by noting that any red process can safely search for a new candidate state. As it discovers new dependences, the process may cause other processes to become red. Since the process itself is red, it has a pointer into some part of the red chain. It can use this pointer to add new processes to the chain without waiting for the token to arrive. Since poll messages must be acknowledged, there is no possibility of a single process being inserted into the list twice.

Note that the parallel variation of the algorithm still requires that the token visit a process before that process can be removed from the red chain. This restriction ensures that the chain is never broken, and all red processes will remain on the chain in spite of the parallel activity along it.

5 Lower Bound for Predicate Detection

We argue in this section that $\Omega(nm)$ is the lower bound on the time complexity of WCP detection. This bound holds for both serial and parallel algorithms. We assume that local states must be considered in order. This restriction models the online nature of the detection algorithm. Since the predicate is possibly unstable, it is not acceptable to skip any local states. Since the application processes may continue executing for an arbitrary time even after the predicate has become true, it is not acceptable to delay the detection process until after the processes have terminated. Hence, identifying a global state that satisfies a global predicate reduces to the problem of eliminating all local states that precede this global state.

Formally, the predicate detection problem is defined as follows. Let $(S, <)$ be any partially ordered finite set of size mn. We are given a decomposition of S into n sets $P_1, ...P_n$ such that P_i is a chain of size m. Each chain is accessed through a queue. Only the head element of each queue may be examined at any instant. When the head element has been removed from the queue, it is lost from further consideration. Let the steps of a parallel algorithm consist of any of the following:

S1. compare all heads of the queues in parallel

S2. delete the heads of any number of queues in parallel.

Theorem 5.1 *Let A be any parallel algorithm which determines if there exists an anti-chain of size n in such a poset of size mn. If A is restricted to steps (S1) and (S2), then it takes at least $\Omega(nm)$ steps to determine the correct answer.*

Proof: See Appendix □

It follows from the above result that any online parallel detection algorithm based on comparison of vectors requires at least mn time. It is also easy to see that in the worst case $\Omega(nm)$ space is also necessary.

6 Conclusion

We have presented distributed algorithms for detecting weak conjunctive predicates. The first of these algorithms reduces the space requirements on any single process from $O(n^2m)$ to $O(nm)$. Further, the distribution of the workload is also more equitable than the the centralized algorithm. We achieve this distribution without increasing the total number of messages, or

increasing (except possibly by a constant factor) the total amount of work performed. The second algorithm reduces time and message complexity in the case n (the number of processes over which the predicate is defined) is large with respect to N (the total number of processes). This algorithm has $O(m)$ time, space, and message complexity on each of the N processes. The algorithms rely on the ability to efficiently eliminate a state from any global cut that does not satisfy the predicate. The happened before relation provides this ability for weak conjunctive predicates.

We establish that any parallel algorithm for detecting a conjunctive predicate requires $\Omega(nm)$ steps when the algorithm is limited to eliminating at most one state from each of n processes. Although it is not possible to improve upon $O(nm)$ steps in the worst case, in the average case faster detection may be possible. We describe methods of introducing parallelism into our algorithms to improve average case performance.

References

[1] Ö Babaoğlu, and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," *Distributed Systems*, 2nd Edition, editor Sape Mullender, Addison Wesley, New York, NY. 1994, pp. 55-96.

[2] K. Chandy, and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, no. 1, pp. 63-75, February 1985.

[3] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates", *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, pp. 163 – 173, May 1991.

[4] C. J. Fidge, "Partial Orders for Parallel Debugging," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* also *SIGPLAN Notices*, Vol. 24. No. 1. January, 1989. pp. 183-194.

[5] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations," *Proceedings of the 23rd Int. Conference on Parallel Processing*, St. Charles, Illinois, pp. 2: 73-76, August 1994.

[6] V. K. Garg, C. Chase, J. R. Mitchell, R. Kilgore, "Detecting Conjunctive Channel Predicates in a Distributed Programming Environment," *Proc. Hawaii International Conference on System Sciences*, Hawaii, 1995, (to appear). Also as University of Texas at Austin, Technical Report TR-PDS-94-02, June 1994.

[7] V. K. Garg, and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, March 1994, pp. 299-307.

[8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications fo the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[9] F. Mattern, "Virtual time and global states of distributed systems", *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V., 1989, pp. 215–226.

[10] Y. Manabe, and M. Imase, "Global Conditions in Debugging Distributed Programs," *Journal of Parallel and Distributed Computing*, Vol. 15, pp. 62-69, 1992.

[11] B. P. Miller and J. Choi, "Breakpoints and Halting in Distributed Programs", *Proceedings of the 8-th International Conference on Distributed Computing Systems*, San Jose, California, June 1988, pp. 316–323.

[12] R. Schwartz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", *Distributed Computing*, 7(3), 1994, pp. 149–174.

[13] A.I. Tomlinson, V. K. Garg, "Detecting Relational Global Predicates in Distributed Systems," *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 1993, pp. 21–31.

7 Appendix

Proof of Lemma 4.1:

($\Rightarrow$) Since, $s \rightarrow_d t$ implies that $s \rightarrow t$, it follows that $s \not\rightarrow t$ implies $s \not\rightarrow_d t$. Thus, the right hand side follows.

($\Leftarrow$) It is sufficient to show that $(\exists i, j : M_i.G \rightarrow M_j.G)$ implies $(\exists k, l : M_k.G \rightarrow_d M_l.G)$. The proof is using induction on the number of processes in the causality chain from M_i.G to M_j.G, that is, on the level of indirect dependence from M_i.G to M_j.G. The base case is trivially true since if there are no processes in the causality chain, then $M_i.G \rightarrow_d M_j.G$ and we are done.

Now consider the case when the indirect dependence goes through K processes. Let the first process in the chain be $P_{i'}$. That is there exists a message sent after M_i.G which is received by $P_{i'}$. There are two cases. If this message is received before $M_{i'}$.G, then we are done since $M_i.G \rightarrow_d M_{i'}.G$ and the right hand side is true. The second case is when this message is received after $M_{i'}.G$. However, this implies that $M_{i'}.G \rightarrow M_j.G$ and from the induction hypothesis the right hand side is true again.

Proof of Lemma 4.2

Part 1. Initially this lemma is true, since M_i.G is initialized to zero. During the detection

algorithm, *color* can be set to red only when a dependence has been found and sent in a poll message. Furthermore, the poll can be sent only when the relevant message was received by the application process prior to the current state in the process which sent the poll message. The proof then follows by induction.

Part 2. This lemma is easily proven by contradiction. Assume that $(i, M_i.G)$ precedes $(j, M_j.G)$. Therefore, a message has been received prior to $(j, M_j.G)$, that was sent after $(i, M_i.G)$. Since a state can be painted green only after all poll messages have been received, any such dependence would have caused $(i, M_i.G)$ to have been painted red, a contradiction.

Part 3. Initially this lemma is true, since all processes are on the red chain, and all processes are red. We note that a process can only be removed from the red chain when the token visits the process. The token can leave a process when *color* is green. Hence, all processes with a red state will remain on the red chain. Further, any process that turns red is added to the chain. Therefore, all red processes are on the red chain. Only red processes can be added to the chain, and a process can become green only when sending the token (simultaneously removing itself from the chain). Therefore, only red processes can be on the red chain.

Proof of Theorem 5.1

An adversary can exploit the fact that the algorithm can only compare heads of the queues. Further, the algorithm can delete only those head elements which are smaller than some other head element. Otherwise, the adversary can produce a consistent cut which includes the deleted head. Thus, if the adversary can ensure that at most one state is deleted in one step, then it has succeeded because there are mn states in all. For the first application of S1, the adversary returns that all heads are concurrent except one which is smaller than exactly one other. Thus, algorithm A can delete only one state in this iteration. Assume that the algorithm deleted a state from P_i in the last iteration. Of the remaining queues, let P_j be the queue with the largest number of elements. In the next iteration, the adversary returns that all heads are concurrent except that head of P_j is smaller than the head of P_i. Thus, only the head of P_j can be deleted in the next iteration. By repeating this procedure it is clear that the adversary can force the algorithm to delete one state at a time until some queue becomes empty. At this point, no queue can have two or more elements otherwise that queue would have been chosen in the last iteration. Hence, at least $mn - n$ states are deleted sequentially before the algorithm can answer

"no". Note that if the algorithm returns "no" any point before this, the adversary can produce another poset which is consistent with all its answers and has an anti-chain of size n.