# Distributed Algorithms for Detecting Conjunctive Predicates

*Vijay K. Garg and Craig M. Chase*

**TR TR-PDS-1995-008 1995**

**Parallel & Distributed Systems group**
**Department of Electrical & Computer Engineering**
**University of Texas at Austin**
**Austin, Texas 78712**

# Distributed Algorithms for Detecting Conjunctive Predicates*

Vijay K. Garg †  Craig M. Chase ‡

Parallel and Distributed Systems Laboratory
The University of Texas at Austin
pdslab@ece.utexas.edu

## Abstract

*This paper discusses efficient distributed detection of global conjunctive predicates in a distributed program. Our methods correctly detect the first consistent cut in which the predicate is true, even if the predicate is unstable. Previous work in detection of such predicates is based on a centralized checker process.*

*In this paper, we introduce algorithms which distribute the computation and space requirements of the detection procedure. Two algorithms are presented. The first algorithm requires $O(n^2m)$ time and space where $m$ is the number of messages sent by any process and $n$ is the number of processes over which the predicate is defined. This algorithm has identical time complexity to the original centralized algorithm. However computation, space and message requirements are distributed evenly over the $n$ processes. The second algorithm requires $O(Nm)$ 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.*

*Parallelism can be introduced into either distributed algorithm, reducing the average case time complexity. We show that the worst-case time complexity can not be improved beyond $O(mn)$ with any on-line detection algorithm.*

## 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, 7, 10, 8, 13, 14]. See [1, 17] for surveys of stable and unstable predicate detection.

This paper is most closely related to [10], [8], and [3]. Garg and Waldecker [10] 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 [10], 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. [10] also presents a method of decentralizing the algorithm. The main idea is that the 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.

Garg, Chase, Mitchell and Kilgore [8] 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 [8]. 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 described in Section 3 and is based on vector clocks similar to [10]. 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. 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 and is therefore useful when $n^2$ is greater

*This paper appears in the International Conference on Distributed Computing Systems, June 1995.

†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 NSF Grant CCR-9409736 and the Jack Kilby/Texas Instruments Endowed Faculty Fellowship in Computer Engineering

than $N$. Section 4 describes this algorithm.

Before describing the distributed algorithms, we provide a brief description of our model for distributed computation and global predicate in Section 2. We establish $\Omega(mn)$ as a lower bound on the worst-case time complexity for any on-line WCP detection algorithm in Section 5.

## 2 Our Model

This section presents the concepts of local and global predicates. The reader is referred to [10, 8] 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 [11]. The happened-before relation can be formally stated as: $\alpha \rightarrow \beta$ iff: $\alpha$ occurs before $\beta$ in the same process, or the action following $\alpha$ is a send of a message and the action preceding $\beta$ 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, $\parallel$, 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, $\alpha$, 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 [10].

### 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. We introduce a new set of $N$ *monitor processes*. One monitor process is mated to each application process. The monitor processes observe the states of the application processes and collectively use these observations to construct a view of the global progress of the application. Monitor processes usually do not communicate to the application processes. The exception to this rule is when a predicate is detected, the monitor processes suspend further execution of the application. Note that at this point some form of check pointing or replay may be necessary to recover the desired state of the application process. Ideally, execution of the application processes is neither halted or slowed until the predicate has been actually detected (in practice some probe effect is inevitable). Thus, at the time the monitor process has realized that a global predicate has become true, the application process may have long since progressed beyond that state. See [5, 16] for a discussion of how this restoration might be accomplished.

Our model is illustrated in Figure 1. This model is similar to that used by Gerstel et al. in [15]. 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").

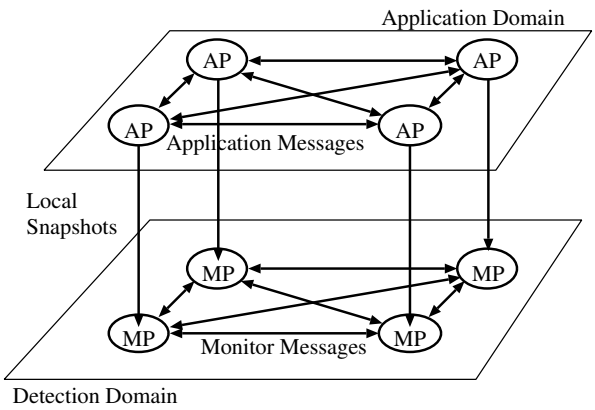


Figure 1: Application and Monitor Processes

## 3 A Vector Clock Based Distributed 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 $O(mn)$ 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 [12, 6]. We use the notation $(i, k)$ to represent the $k$th 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.

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

Each application process 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.

```
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

### 3.2 Monitor Processes for Token Algorithm

The monitor process has the responsibility to enqueue all of the local snapshots received from the application process. The application process is allowed to run at its normal rate (subject to the obvious imposition implied by Figure 2). Monitor processes consider each snapshot in sequence until either the WCP is detected, or until all local snapshots have been exhausted and the application process has terminated. In this latter case the monitor processes simply abort the predicate detection algorithm.

The detection algorithm works by considering a sequence of global cuts. Although there are $O(m^n)$ possible cuts, at most $mn$ cuts are considered by our algorithm. Beginning with the first state from each application process, each cut is checked for consistency. Since the application processes only send local snapshots of states in which the local predicates are true, any consistent cut made of these states will satisfy the WCP. If the cut is not consistent, then the state with the smallest vector clock is eliminated and its successor is used to form the next candidate cut. This method is also used in the centralized algorithm, see [10] for a discussion and proof.

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

on receiving the token (G,color)
    while (color[i] = red) do
        receive candidate from 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 P_j
    else detect := true;
```

Figure 3: Monitor Process Algorithm

The current global cut is identified by a unique token which is passed between the monitor processes. The token contains two vectors. The first vector is labeled *G* and holds logical clock values. If *G[i]* has the value $k$, then state $k$ from process $P_i$ is part of the current cut. The token is initialized with $\forall i : G[i] = 0$ (recall that application processes initialize their logical clocks to the value 1). The second vector is labeled *color*. Loosely speaking, the color indicates which states in the cut are not concurrent. More precisely, the color of a state can be either red or green. If *color[i]* = red then the state $(i, G[i])$ and all its predecessors 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 concurrency with this new candidate. This activity is repeated until the candidate state from $P_i$ did not happen before any other state in the cut. The color[i] is then set to green and G[i] is set to the logical clock value from this candidate state. Next, if $M_i$ 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, 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. $color[i] = green \Rightarrow \forall j : (i, G[i]) \not\rightarrow (j, G[j])$
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 $G[i] \leq candidate[j]$. 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, $\exists j : G[i] \rightarrow G[j] \Rightarrow color[i] = red$. 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 $j$. Let $G[i] \rightarrow G[j]$. This implies that the vector $M_j.candidate$ which satisfies $M_j.candidate[j] = G[j]$ also satisfies $M_j.candidate[i] \geq G[i]$ (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 $G[j]$. Since all states preceding $G[j]$ are red, it follows from the induction hypothesis that G[i] can not be used to satisfy the WCP. □

**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.

We now argue that if there exists a cut that satisfies the WCP, then the algorithm will eventually halt with detect set to true. Note that a process can perform only a finite amount of work each time it receives the token. Furthermore, every time the token is received by a process, at least one state is eliminated (a red state is thrown away and a new candidate is received from the application process). Since there are a finite number of states, the token can be received only a finite number of times. Therefore, the algorithm must eventually terminate. We now use Lemma 3.1 part 4 to argue that the detection algorithm can not bypass the cut that satisfied the WCP (the algorithm only advances by finding a successor to a red state). Thus, when the algorithm terminates, all states must be green, and detect will have been set to true.

□

### 3.4 Analysis of Token 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 [10]. The main difference between the algorithms is that the work for any process in the distributed algorithm is at most $O(mn)$.

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 $O(n \log m)$ bits. The factor of $\log m$ is based on the number of bits necessary to encode a logical clock value. Therefore, the total number

of bits communicated is $O(n^2 m \log m)$. In practice, however, a single integer (typically of fixed size and 32-bits or larger) is large enough to hold any clock value, suggesting message traffic of $O(n^2 m)$ integers.

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

### 3.5 Introducing Parallelism

Although the work of predicate detection is distributed by this algorithm, it is still sequential. A monitor process is active only if it has the token. However, parallelism can be introduced 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. In the worst case, tokens are returned to $P_0$ $O(mn)$ times, requiring $P_0$ to perform $O(n^2 m)$ work.

## 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 $O(mN)$ 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. 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 | Direct Dependence |
|---|---|
| token.*color*[i] | $M_i$.*color* |
| token.*G*[i] | $M_i$.*G* |

Table 1: Distribution of Token Data Structures

A process is active whenever its *color* is red. Figure 4 shows the activity taken. 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 dependence message is sent to $M_j$ where $P_j$ is the source of the dependence. The dependence message consists of a single integer, the clock value for the dependence. All dependences are processed in this manner and then color is set to green.

Upon receiving a dependence message, the monitor process functions as shown in Figure 5. 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.

The predicate is detected when all dependence messages have been received, and all monitor processes are inactive (i.e., their *color* is green). Hence, if termination occurs the predicate has been detected. We use Dijkstra and Scholten's algorithm for termination detection on diffusing computations [4]. To make our algorithm satisfy the requirement to be a diffusing computation we arbitrarily appoint one monitor process (say $M_0$) as the originating process. Dijkstra and Scholten's algorithm can then be easily

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

upon color becoming red
deplist := ∅;
repeat
    receive candidate from application process
    deplist := deplist ∪ candidate.deplist;
until (candidate.clock > G)

/* process dependent list*/
for (each (i, k) ∈ deplist)
    send dependence(k) to M_i endfor

color := green
```

Figure 4: Monitor Process for Direct Dependence Algorithm

```
upon receiving dependence message from M_i:
if (dependence.clock ≥ G) then
    color := red;
    G := dependence.clock;
endif
```

Figure 5: Upon Receiving a Dependence Message

superimposed on the monitor process's and will result in $M_0$ being informed of termination when it occurs.

### 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 note 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 (see [19]). 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 $t$.

**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 Theorem 7 of [19].
□

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 $i$ : $M_i.G \neq 0 \wedge M_i.color = red \Rightarrow \exists j : (i, M_i.G) \rightarrow (j, M_j.G)$*
*2. When all dependence messages have been received $M_i.color = green \wedge M_j.color = green \Rightarrow (i, M_j.G) \not\rightarrow_d (j, M_j.G) \wedge (j, M_j.G) \not\rightarrow_d (i, M_i.G)$.*
*3. If $(M_i.color = red)$ then there is no global cut satisfying the WCP that includes $(i, M_i.G)$.*

**Proof:** 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 as a response to receiving a dependence message. That message can have been sent only if there exists some state $(j, k)$ such that $(i, M_i.G) \rightarrow_d (j, k)$. Furthermore, $(j, k) \rightarrow_d (j, M_j.G)$ (notice both states are from the same process, $P_j$). From this it follows that $(i, M_i.G) \rightarrow (j, M_j.G)$.

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 $(i, M_i.G)$. This message must generate a dependence message. Once that dependence message is received, the $M_i.color$ will be set to red. Since the color can not be changed to green without receiving a new state (and thus changing $M_i.G$) we have a contradiction.

3. The proof of this part is analogous to Lemma 3.1 part 4.
□

**Theorem 4.3** *Termination of the direct dependence algorithm occurs iff the cut $G$ defined by all variables $M_i.G$ is the first consistent cut that satisfies the WCP.*

**Proof:** We first show that termination implies that $G$ is the correct cut. Note that termination occurs only if all colors are set to green and no dependence messages are in route. 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 3 of Lemma 4.2 tells us that we can not bypass any cut that satisfies the WCP. Hence when termination occurs, $G$ is the first such cut to satisfy the WCP.

We now argue that termination always occurs. The total number of dependence messages which can be sent is bounded by the number of messages sent by application processes. A finite number of messages (at most $m$) can be sent by any application process prior to the WCP becoming true. Thus, a finite number of dependence messages can be sent. Since each monitor process does a finite amount of work as a result of receiving a dependence message, the algorithm will eventually terminate. □

### 4.4 Analysis of the Direct Dependence Algorithm

At most $mN$ dependences exist in the system. Hence, at most $mN$ dependence messages can be sent. Dijkstra and

Scholten show that termination can be detected with merely doubling the number of messages sent. At most $mN$ local snapshots can be sent by application processes. Hence, the total number of messages related to predicate detection is $O(mN)$.

A dependence message contains one integer. We again note that in practice integers are constant size. An absolute limit on the size of an integer in this algorithm is the maximum of $\log m$ and $\log N$ bits. The messages sent for termination detection carry no actual information, and thus can be arbitrarily small. Therefore, total number of bits sent by monitor processes is $O(mN)$ integers. Local snapshots are variable sized. However the total number of dependences for any one process can not exceed $m$. Since a dependence can be represented by a pair of integers a total of $O(mN)$ integers 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 and sending a dependence message. Since there can be no more than $O(m)$ dependences for any one process, the total work performed by all monitor processes is $O(mN)$.

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 $O(m)$ integers are required on any process.

### 4.5 Introducing Parallelism

The direct-dependence algorithm is inherently parallel, there is no additional structure required to achieve concurrent activity of the monitor processes. Monitor processes become active whenever they receive a dependence message, and inactive once they've identified an acceptable candidate state. However, it must be noted that in the worst case only a single monitor process will be active (or no processes are active and a single dependence message is in route). In the following section we give a lower bound which explains why parallelism can not be expected in the worst case.

## 5 Lower Bound for Predicate Detection

We argue in this section that $\Omega(mn)$ 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 $\Omega(mn)$ steps to determine the correct answer.*

**Proof:** 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 $n$. □

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 $\Omega(mn)$ 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 $O(mn)$. 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 $\Omega(mn)$ 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 $O(mn)$ 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.

### Acknowledgements

We thank Richard Kilgore and the anonymous reviewers for their careful reading of this paper and valuable suggestions.

## 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] Edsger W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol 11, No. 1, pp. 1–4, August, 1980.

[5] S. Feldman and C. Brown, "IGOR: A System for Program Debugging via Reversible Execution", *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 112–123, May, "1988".

[6] 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.

[7] 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*, Pennsylvania State University, August 1994.

[8] 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*, vol. 2, pp. 232–241, January, 1995.

[9] V. K. Garg, and A. Tomlinson, "Using Induction to Prove Properties of Distributed Programs," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1993.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] O. Gerstel, M. Hurfin, N. Plouzeau, M. Raynal and S. Zaks, "On-the-fly Replay: A Practical Paradigm and its Implementation for Distributed Debugging", *Proceedings Symposium on Parallel and Distributed Processing*, Dallas, TX, October, 1994.

[16] D. Z. Pan and M. A. Linton, "Supporting Reverse Execution of Parallel Programs", *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 112–123, May, "1988".

[17] 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.

[18] 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.

[19] Yi-Min Wang, Andy Lowry and W. Kent Fuchs, "Consistent Global Checkpoints Based on Direct Dependency Tracking", *Information Processing Letters*, pp. 223-230, 1994.