Distributed Algorithms for Detecting Conjunctive Predicates

Vijay. K. Garg Craig M. Chase

TR ECE-PDS-94-03 June 1994

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

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†

June 16, 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 centralized checker process. The checker process 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 in the system. In this paper, we discuss token-based algorithms which distribute the computation and space requirements of the detection procedure. The total number of messages required by the system is the same as in the centralized algorithm, $O(mn)$. We also establish $\Omega(mn)$ as a lower bound on the time complexity for predicate detection.

1 Introduction

Detection of a global predicate, that is, a condition that depends on the state of the entire system, 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, 8, 6, 9, 10, 11, 12, 13, 15]. See [1, 14] for surveys of stable and unstable predicate detection.

This paper is most closely related to [8] and [6]. Garg and Waldecker [8] present a method for detecting weak conjunctive predicates (WCP). A WCP is a global predicate formed as the conjunction of n local predicates. In [8], each of the n processes in the system check for their own local predicate and send messages to a global predicate checker that detects when all predicates are true in a consistent cut. Garg, Chase, Mitchell and Kilgore [6] extend this algorithm to include predicates based also on the state of communication channels. This larger class of predicates are termed Generalized Conjunctive Predicates (GCP). A centralized checker process was also used in [6].

In this paper, we present distributed algorithms to detect conjunctive predicates. Our algorithms are based on tokens. Our exposition of algorithms is as follows. We first describe a single token algorithm for WCP. Next, we show how this algorithm can be generalized to an algorithm for WCP with g tokens. We also show how a single-token algorithm can be generalized

*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

to include channel predicates. It is then a simple task to combine these two extensions to get a g-token algorithm for GCP.

The main idea in detecting a WCP using a single token is as follows. The token carries in it a candidate global cut in which the desired global predicate could be true. A global cut consists of states from each of the process such that local predicates are true in those states. If such a global state is also consistent, then the WCP is true. This check of consistency is performed by the process holding the token. If the process finds that the cut is consistent, then the global predicate is detected; otherwise the token is sent to the process which violates the consistency condition on the global cut. This process identifies a new candidate cut by replacing the state from this process with a local successor. The process will then check for consistency conditions again.

To permit greater concurrency in the algorithm, multiple tokens are used. The set of processes are partitioned into g groups. A token is associated with each group of processes and independently finds a cut which is consistent with respect to that group. Once all tokens have determined their cuts, a single process collects all these tokens and determines a unique global cut which is consistent with all tokens. It then checks whether consistency across groups is satisfied in that global cut. If not, groups which violate the consistency conditions get new tokens and continue searching for a consistent cut.

In presence of channel predicates, the process holding a token is required not only to check the consistency condition, but also whether channels incident to that process satisfy their predicates. This requires the history of sends and receives along those channels. As in [6], we require channel predicates be monotonic. This restriction allows the algorithm to eliminate at least one state when a channel predicate is found to be false.

As argued in [8, 6], many problems in distributed systems can be viewed as special cases of WCP or GCP detection. Both time and space complexity of our algorithm is $O(mn)$ per process where m is the maximum number of application messages generated by a process and n is the number of processes. The total number of messages generated by our detection algorithms is also $O(mn)$.

The next section presents definitions which are necessary in understanding the method of detecting conjunctive predicates. Section 3 discusses the algorithm for application processes. Section 4 presents a distributed algorithm to detect a WCP. This algorithm is based on a token moving around the network. Section 5 presents a generalization of this algorithm to g tokens. Section 6 presents a token based algorithm for detecting a GCP. Section 7 discusses optimizations and lower bounds for these problems.

2 Our Model

This section presents the concepts of global, local, and channel predicates. The reader is referred to [8, 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 [10]. 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 $\exists\gamma : \alpha \rightarrow \gamma \wedge \gamma \rightarrow \beta$. The happened-before relation leads to the idea of concurrency, which is used in the detection of global predicates. Two states for which the happened-before relation does not hold in either direction are said to be concurrent. That is, there exists a run with local clocks running at particular speeds where two concurrent states can be made to occur at the same physical time. The symbol, $\|$, is used to represent concurrency. The relationship 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 predicate* is defined as any boolean formula on a local process. 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 *channel predicate* is any boolean function of the state of a channel. We define a channel to be a uni-directional connection between two processes — one process performs all send events and the other all receive events. Let α and β be states at P_i and P_j. We use $\alpha.Sent[j]$ to denote the sequence of all messages sent at or before state α from i to j, and $\beta.Rcvd[i]$ for sequence of all messages received at or before state β from i to j. Channels have no memory. Hence, the state of a channel is the difference between the set of messages sent and the set of messages received. A channel predicate can then be written as:

$$chanp(\alpha.Sent[j] - \beta.Rcvd[i])$$

or, more concisely:

$$chanp(\alpha, \beta)$$

For efficiency reasons the channel predicates must be monotonic. The requirement for monotonicity can be stated as:

Definition 1 *A channel predicate, chanp(S), is* monotonic *iff:*

$$\forall S :: \neg chanp(S) \Rightarrow (\forall S' :: \neg chanp(S \cup S')) \vee (\forall R :: \neg chanp(S - R))$$

That is, given any channel state, S, in which the predicate is false, then either sending more messages is guaranteed to leave the predicate false, or receiving more messages is guaranteed to leave the predicate false. We assume that when the channel predicate is evaluated in some state S, it is also known which of these two cases applies. To model this assumption, we define a monotonic predicate to be a 3-valued function — $\{T, F_s, F_r\}$. The predicate evaluates to T if it is true for the current channel state. It evaluates to F_s if it is false for the current channel state and will remain false in the presence of an arbitrary set of additional messages sent on the channel (in the absence of receives). It evaluates to F_r if it is false for the current channel state and will remain false in the presence of an arbitrary set of additional messages received.

Some examples of monotonic channel predicates are:

Example 1. *Detection of empty channels:* $chanp(S) \equiv (S = \emptyset)$: It is obvious that if this predicate is not currently true, sending more messages will not make it true.

Example 2. *Detection of exactly k messages in a channel:* $chanp(S) \equiv$ (length of $S = k$): In any state where there are more than k messages in the channel, the predicate cannot be made true by sending more messages. In any state when there are less than k messages in a channel, the predicate can not be made true by receiving more messages. The only other possible state is when the channel has exactly k messages in it, and in this state the predicate is true.

A predicate formed only by the conjunction of local predicates is called a Weak Conjunctive Predicate (WCP). A predicate formed by a conjunction of local and channel predicates is called a generalized conjunctive predicate (GCP). A WCP or GCP is true for a given run if and only if there exists a consistent global cut in that run in which all conjuncts are true.

We restrict our consideration to conjunctive predicates because any boolean predicate can be detected using an algorithm that detects conjunctive predicates [6].

3 Application Processes

A distributed program consists of a set of interacting application processes. The application processes exchange messages over a set of channels. Without loss of generality, we assume a fully connected network. To detect a global predicate we introduce a set of monitor processes. The monitor processes do not directly participate in the distributed computation. Their only function is to detect global predicates. One monitor process is associated with each application process.

The execution of each application process can be viewed as sequence of local states. Since both a WCP and a GCP require 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 [11, 4]. We use the notation (i, k) to represent the kth state on 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 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

- 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 also maintain information about the send and receive channel history for all channels incident to them. 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 is of the form:

$$(vclock, incsend, increcv)$$

where *vclock* is the current vector clock and *incsend* and *increcv* are the list of messages sent and received from the application process since the last message for predicate detection was sent. An algorithm for this process is given in Figure 1. Note that for WCP detection, the *incsend* and *increcv* components are not required.

4 Monitor Processes for WCP

In this section we describe the monitor processes in the distributed WCP detection algorithm. Monitor processes are responsible for searching for a consistent cut that satisfies the WCP. A candidate cut is stored in the token. The token also stores enough information to determine whether the candidate cut is consistent. If the process holding the token finds that the cut is consistent, then the global predicate is detected; otherwise the token is sent to a process which violates the consistency condition on the candidate cut.

```
var
incsend, increcv: vector of messages;
vclock: vector [1..n] of integer;
initially
        ∀j : j ≠ i :vclock[j] = 0;
        vclock[i] = 1;
                /* last causal msg rcvd from process 1 to n */
        firsflag = true;
        incsend = increcv = ∅;

for sending m do
        send (vclock, m);
        vclock[i]++ ;
        firstflag:=true;
        incsend:= incsend ⊕ m; /* concatenate */

on receive (vclock, m) do
        ∀j :vclock[j]:=max(vclock[j], msg_vclock[j]);
        vclock[i]++ ;
        firstflag:=true;
        increcv:= increcv ⊕ m; /* concatenate */

on (local_pred = true)∧ firstflag do
        firstflag := false;
        send (vclock, incsend, increcv) to monitor process;
        incsend:=increcv:=∅;
```

Figure 1: Application Process Algorithm for P_i

A token is sent to a process P_i only when the current cut is not consistent. Specifically, the token is sent to P_i only when current state from P_i happened before some other state in the candidate cut. Hence, the current state on P_i can be eliminated, since it can never be part of a consistent cut that satisfies the WCP. Once the monitor process for P_i has eliminated the current state, it receives a new state from the application process and then checks for consistency conditions again. This sequence of steps is repeated until all states are eliminated from some process P_i or until the WCP is detected.

A monitor process is active only if it has the token. The token consists of two vectors G and *color*. G is a global state vector, an array with all components initialized to 0. It represents the current global cut that is a candidate for satisfying a WCP. Specifically, $G[i] = k$ indicates that state (i, k) is part of the current cut. We maintain the invariant that $G[i] = k$ implies that any global cut C with $(i, s) \in C$ and $s < k$ cannot satisfy the WCP.

The second vector, *color*, indicates which states have been eliminated. The color of a state can be either red or green. A red state has been eliminated and can never satisfy the global predicate. If $color[i] = red$ then state $(i, G[i])$ and all predecessors to this state are red. If $color[i] = green$, then there is no state in G such that $(i, G[i])$ happened before that state.

The application process sends its vector clock to the monitor process each time the local predicate becomes true since the vector clock was last changed. We require that this communication be FIFO. The monitor process keeps in local memory the vector clock from the current candidate state. This variable is called *candidate*. When the monitor process P_i receives the token, it continues to advance the current cut by receiving messages from the application process until $candidate.vclock[i] > G[i]$. Once this condition is met, the monitor process compares the happened before relations of the new candidate to the current cut G. When for some j, $(j, G[j])$ happened before the candidate, we paint state $(j, G[j])$ red. If all states in G are green, that is

the cut, G, is consistent, then the process has detected the WCP. Otherwise, the process sends the token to a process whose color is red.

```
var
candidate:array[1..n] of integer;

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

Figure 2: Monitor Process Algorithm

4.1 Correctness of WCP Detection Algorithm

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

Lemma 4.1 *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. $color[i] = green \Rightarrow \forall k : (i, G[i]) \not\rightarrow (k, G[k])$;
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 $G[i] < candidate.vclock[j]$. By the property of vector clocks, for any $candidate$ in P_j, $candidate.vclock[i] \rightarrow candidate.vclock[j]$. 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 $P_k.candidate$ which satisfies $P_k.candidate.vclock[k] = G[k]$ also satisfies $P_k.candidate.vclock[i] \geq G[i]$ (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 $G[j]$. Also, by the induction hypothesis it follows that the same holds for all states preceding $G[j]$. Thus, there is no global cut which includes a state in P_j and is consistent with $G[i]$. □

Theorem 4.2 *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 4.1 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. For any process P_i, $G[i]$ is changed only when $color[i]$ is red. Again, from Lemma 4.1, part 1 it follows that $\exists j : G[i] \rightarrow G[j]$. Thus, $G[i]$ cannot be part of a global cut that satisfies 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. 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 4.1 it follows that any eliminated state can not be part of a consistent cut satisfying the WCP. Therefore, no state in H is eliminated. □

4.2 Analysis of Single-Token WCP 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. At least one state is eliminated every time a process receives 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 [8]. The main difference between the algorithms is that the work for any process in the distributed algorithm is at most $O(mn)$. Thus, this algorithm achieves distribution with almost no overhead.

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 $O(mn)$. 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 only $O(n)$ in size). There are at most m such messages each of size $O(n)$. Therefore, $O(mn)$ space is required by the algorithm for every process. We again observe that the checker-based algorithm requires $O(n^2m)$ space.

5 A g token algorithm for WCP

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 processes into g groups and use one token for each group. Let *group(i)* denote the group to which P_i belongs. The monitor processes in each group run the single-token algorithm, except that the process with the token checks for violations of the consistency condition only for the processes in its group. Once any process in

a group determines that the group satisfies the consistency condition, the token is returned to some pre-determined *leader* process (say P_0). The monitor process for the g-token algorithm is shown in Figure 3.

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. The algorithm for P_0 is given in Figure 4.

```
var
candidate:array[1..n] of integer;

on receiving the token (G,color)
      while (color[i] = red) do
            receive candidate from application process P_i
            if (candidate.vclock[i] > G[i]) then
                  G[i] := candidate.vclock[i];
                  color[i] := green;
            endif
      endwhile
      for j ≠ i:
      if (candidate.vclock[j] > G[j]) then
            G[j] := candidate.vclock[j];
            color[j] := red;
      endif
      if (∃ j ∈ group(i): color[j] = red) then send token to P_j
      else send token to P_0;
```

Figure 3: Monitor Process P_i for g-token WCP Algorithm

```
var
myG:array[1..n] of integer;
mycolor:array[1..n] of {green, red};
havetoken:array[1..g] of boolean initially true;
on ∀ i : havetoken[i]
      for (all j)
            if (color[j]= red ∧ havetoken[group(j)]) then
                  send token (myG, mycolor) to P_j
                  havetoken[group(j)] := false;
            endif
      if (∀ i : havetoken[i]) then detect := true;

on receiving the token (G,color) from P_j
      if (myG[i] < G[i]) then
            myG[i] := G[i];
            mycolor[i] := color[i];
      endif
      havetoken[group(j)] := true;
```

Figure 4: Leader Process P_0 for g-token WCP Algorithm

We observe that if there is a single group, then the role of P_0 is trivial and can be carried out by the process which has the token. In this case, this algorithm reduces to the single-token algorithm. At the other end of the spectrum, if there are n groups, then the algorithm reduces to a centralized algorithm with P_0 performing $O(n^2m)$ work.

5.1 Correctness of g-token WCP Detection Algorithm

We now show the correctness of the g-token algorithm.

Theorem 5.1 *The flag* detect *is true with G iff G is the first cut that satisfies the WCP.*
Proof: The condition *detect* implies that P_0 has token for all groups and $\forall j : mycolor[j] =$ green. Note that if the latter is false, then the token is returned to all groups which contain a red state. Using arguments similar to Theorem 4.2 it follows that the WCP is true in G and that G is the first such cut.

Conversely, let H be the first cut for which WCP is true. It is easy to see that for any cut G before H, there exist $i, j : G[i] \rightarrow G[j]$. If $group(i) = group(j)$, then by the arguments in 4.2, the token for that group will eventually reach P_i and the cut on P_i will be advanced. If $group(i)$ is different from $group(j)$, then eventually tokens for both the groups return to P_0. When the tokens are merged, P_0 detects that the consistency condition is false for P_i. Hence the token is sent to the group(i) and eventually the cut is advanced along P_i. □

5.2 Analysis of the g-token algorithm

Similar to the single-token version, each monitor process (other than P_0) can receive the token at most m times. Hence, a total of $O(mn)$ work can be done by each monitor process. In the worst case, the leader process, P_0, can issue only one token at each iteration of the algorithm. Merging tokens is an $O(n)$ operation. Hence P_0 performs $O(n^2m)$ work. However, this can be reduced to $O(mn)$ work on P_0 by a more clever implementation. In this implementation, instead of returning the complete token to P_0, a group sends only the components of the token which have changed. The changes to the token are limited to a total of $O(mn)$ over the duration of the computation.

The space complexity of the g-token algorithm is identical to that of the single-token version, $O(mn)$ space on each process.

We now consider the message complexity. The g-token algorithm has two kinds of messages — those received by P_0 and those received by other processes. For any message that is received by a process other than P_0 at least one state is advanced. Thus, there are at most mn messages of that kind. Further, the number of messages received by P_0 is equal to the number of messages sent by P_0. Any time a message is sent by P_0 at least one state is eliminated; therefore, there are at most mn messages sent or received by P_0. On summing up all the messages, we get that there are at most $2mn$ messages used by the algorithm.

6 A Distributed GCP Detection Algorithm

The distributed algorithm to detect WCPs can be modified to detect GCPs as well. In this paper, we describe only the single token version. This section presents an overview of the algorithm, a proof that the algorithm correctly detects the first cut in which the GCP is satisfied, and a complexity analysis. A multiple token version of this algorithm is also possible using similar techniques to those in Section 5.

6.1 Data Structures

Each monitor process P_i maintains a set of data structures. The total space required for this data is $O(mn)$. The data structures are:

- *S[j]* : vector of sets of message — for each j, the set of messages sent by P_i to P_j but not yet received according to the current cut.
- *R[j]* : vector of sets of message — for each j, the set of messages received by P_j but not yet sent by P_i according to the current cut.
- *cp[j]* : vector of { T, F_s, F_r, X } — for each j, the value of the channel predicate for the channel from P_i to P_j, or X if the value is not currently known.
- *IR[j]* : vector of sets of message — for each j, a subset of the messages received by P_i prior to the current cut. These messages may or may not have been sent by P_j according to the current cut. The *IR* vector is an incremental receive history. Periodically, an update message consisting of the contents of IR[j] will be sent to P_j. IR[j] contains only those messages from receive events not already sent in previous updates.
- *count* : integer — the number of updates received by P_i.
- *candidate* — the local snapshot (see Section 3) that corresponds to the current cut.

The token is similar to that used in the WCP algorithm. We add two vector fields, each vector has n elements. Hence, the space required for the token remains $O(n)$

- *updates* : vector of integer — the number of update messages sent to each process.
- *CP* : vector of {T, X} — The local satisfaction vector. If *CP*[i] is true, then all channel predicates are true for those channels on which process i can send.

6.2 Distributed Algorithm for GCP Detection

The algorithm's major component is shown in Figure 5. This code is executed by a monitor process whenever it receives the token. The code is similar to that used in the WCP algorithm. We describe only the differences.

In addition to passing the token, monitor processes exchange update messages. Only the process with the token is allowed to send update messages. When a process receives the token, it must wait until it receives all outstanding update messages before it attempts to evaluate channel predicates. We implement this by maintaining an update count vector in the token. When a process receives the token, it simply waits until its local count matches the update count in the token.

The subroutines *update-channels* and *eval-cps* in the GCP algorithm maintain the internal data structures that describe the channel states. The code for these subroutines is shown in figures 6 and 7 respectively. The *update-channels* routine updates the data structures in response to receiving a new local snapshot from the application process. The monitor process is responsible for maintaining the channel state for only those channels on which the application process can send messages. Hence, for message sends, the monitor process does not need to communicate with other monitor processes to update the channel state. It either adds the message to the appropriate list in the S array, or subtracts the message from the appropriate list in the R array. However, for receive events, the monitor process can not directly update the channel state. All receive events are eventually placed into update messages and sent to the remote monitor process that is responsible for the respective channel. Rather than sending the update messages immediately, we collect the messages in the *IR* array. The update messages are sent before the token leaves this process (see Figure 9). This optimization reduces the

```
on (P_i receiving the token) do
    /* Find a candidate local state */
    while (color[i] = red) do
        receive candidate from application process P_i
        update-channels(i);
        if (candidate.vclock[i] > G[i]) then
            G[i] := candidate.vclock[i];
            color[i]:=green;
        endif
    endwhile

    /* Do all candidates form a consistent cut? */
    for (j ≠ i) do
        if (candidate.vclock[j] > G[j]) then
            G[j] := candidate.vclock[j];
            color[j] := red;
        endif
    endfor
    if (∃ j: color[j] = red) then send token to P_j

    wait (updates[i] = count)
    eval-cps();
```

Figure 5: Monitor Process for GCP Algorithm

number of messages sent in the average case. When these update messages are received, the code in Figure 8 is executed.

```
update-channels()
    for (all j : candidate.incsend[j] ≠ ∅)
        S[j] := S[j] ⊕ (candidate.incsend[j] − R[j]); /* concatenate */
        R[j] := R[j] − candidate.incsend[j];
        if (cp[j] = T ∨ cp[j] = F_r) cp[j] := X;
        CP[i] := X;
    endfor
    for (all j : candidate.increcv[j] ≠ ∅)
        IR[j] := IR[j] ∪ candidate.increcv[j];
        CP[j] := X;
```

Figure 6: Update channels

The subroutine *eval-cps* evaluates channel predicates. The array *cp* is used to record the value of channel predicates. Since channel predicates can only change when there is activity on the channel, using the *cp* array allows us to avoid unnecessarily reevaluating channel predicates whose value has not changed.

There are four possible cases for the *eval-cps* subroutine. In the first case, at least one channel predicate is known to be both false and send monotonic for the current channel state. Hence, the state from the receiving process can be deleted since the GCP will never be satisfied until at least one new message is received on this channel. The second case is analogous, but

```
subroutine eval-cps:
    /* Evaluate channel predicates on the consistent cut */
    while (has_token(i) ∧ detect = false)
        if (∃ j : cp[j] = F_s)
            CP[i] = X;
            color[j] := red;
            send-token(j);
        else if (∃ j : cp[j] = F_r)
            CP[i] = X;
            color[j] := red;
            send-token(i);
        else if (∀ j : cp[j] = T)
            CP[i] = T;
            if (∃ j : CP[j] = X) send-token(j);
            else detect := true;
        else
            let j : cp[j] = X;
            cp[j] := chanp(S[j]);
    endwhile
```

Figure 7: Evaluating Channel Predicates

with a channel predicate that is receive monotonic for the current state. When a state is eliminated, the token is sent to that process.

In the third case, all channel predicates are true. It is important to note that *eval-cps* is only called when the current cut represents a consistent cut and that all local predicates are true in this global state. Hence, if all channel predicates are true, then the current cut satisfies the GCP. However, each monitor process has direct access to the state of only a subset of the channels. The token contains a vector, CP, for which element i is true only if all channel predicates for P_i are also true. If there exists some process for which the CP vector is not true, then *eval-cps* sends the token to that process. If the token's CP vector is all true, then the current cut satisfies the GCP, and the detect flag is set to true.

The fourth case is when no predicates are known to be false, and the value of at least one channel predicate is unknown. In this case, an unknown channel predicate is evaluated. Since the current cut is a consistent cut, and since all update messages have been received before this subroutine is called, the channel state is exactly the contents of the S list [6]. Evaluating channel predicates is given the lowest priority of the four cases. This policy ensures that the monitor process does not perform extra work by evaluating channel predicates whose values will not be required.

The remaining components of the distributed GCP detection algorithm are shown in Figures 8 and 9. Figure 8 shows the code that is executed whenever an update message is received by a monitor process. Figure 9 shows the code that is executed by a monitor process to send the token.

6.3 Correctness of the Distributed GCP Algorithm

We begin our analysis by describing several invariant properties of the algorithm. These properties can be easily verified by examining the code. The invariants are then used to establish the correctness of our algorithm.

on (P_i receiving an update (increcv) from P_j) **do**
    R[j] := R[j] ∪ (increcv - S[j]);
    S[j] := S[j] - increcv;

Figure 8: On Receiving an Update Message

send-token(k : integer)
for (all j : IR[j] ≠ ∅)
    updates[j] += 1;
    send IR[j] to P_j;
    IR[j] := ∅;
endfor
send token to P_k;

Figure 9: Procedure to Send the Token

The following invariant is based on the fact that before the token can leave a process, all incremental receive information has been placed into update messages:

Invariant 6.1 *For any process P_i, if the token is not on process P_i, then:*

$$\forall j : j \neq i : P_i.IR[j] = \emptyset$$

Observe that only the process currently possessing the token can add to its IR sets. Furthermore, before relinquishing the token, the process must empty all of its IR sets.

Based on Invariant 6.1 we can establish the validity of the S and R vectors. The centralized GCP-detection algorithm [6] used a matrix of lists to record the channel state along the current cut. These matrices were labeled *S[i,j]* and *R[i,j]*. The distributed algorithm assigns one row of these matrices to each process. In [6], it is shown that for any i and j such that P_i and P_j are concurrent on the current cut, *S[i,j]* is exactly the channel state on the current cut. Rather than demonstrate the same property for the distributed versions of S and R, we establish an invariant which describes how the distributed version of these variables relate to the centralized version:

Invariant 6.2 *S and R sets:*
$\forall i$: updates[i] = $P_i.count \Rightarrow$

$$\begin{aligned} \forall j : S[i,j] &= P_i.S[j] - P_j.IR[i] \\ \forall j : R[i,j] &= P_i.R[j] \cup (P_j.IR[i] - P_i.S[j])) \end{aligned}$$

Invariant 6.2 and [6] allows us to know that $P_i.S[j]$ is exactly the channel state for the channel between P_i and P_j whenever process i has the token, and P_i and P_j are concurrent. These conditions are always true when the subroutine *eval-cps* is invoked.

The next invariant describes how $P_i.cp[j]$ relates to the value of a channel predicate.

Invariant 6.3 *If the token is on process P_i and the current cut is a consistent cut, then the following holds whenever updates[i] = $P_i.count$:*

$$P_i.cp[j] \neq X \Rightarrow P_i.cp[j] = chanp(G[i], G[j]$$

Note that the conditions for this invariant always hold when the *eval-cps* subroutine is called. Finally, we present the invariant for the CP vector in the token.

Invariant 6.4

$$CP[i] = T \Rightarrow \forall j : chanp(G[i], G[j])$$

Theorem 6.1 (Soundness) *If the 1-token distributed GCP detection algorithm sets detect to true, then the current cut satisfies the GCP.*

Proof: Detect is set to true only in the subroutine eval-cps (Figure 7). This subroutine can only be invoked when the token is on process P_i, $\forall j$: color$[j]$ = red and P_i.count = updates$[i]$. Hence, the current cut is a consistent cut satisfying all local predicates and all update messages have been received. Furthermore, the detect flag is set to true only when: $\forall j : CP[j] = T$. Hence, by Invariant 6.4, all channel predicates must also be true. Therefore, the current cut satisfies the GCP. □

Theorem 6.2 (Completeness) *If H is the first cut that satisfies the GCP, then the algorithm will set detect to true with $G = H$.*

Proof: The proof is in two parts. First we show that if the current cut is a predecessor to H, then that cut will be advanced. Second, we show that if the current cut satisfies the GCP, then the detect flag will be set to true.

Assume that the current cut, G, is a predecessor to H. There are two cases (Note: if both cases apply, case 1 takes priority):

case 1 $\exists i, j : (i, G[i]) \rightarrow (j, G[j])$: That is, the cut is not a consistent cut. — By part 2 of Lemma 4.1, $color[i]$ is red. The token will eventually be sent to P_i which will advance the cut.

case 2 $\exists i, j$: chanp$(G[i], G[j])$: That is, at least one channel predicate is not satisfied. — By Invariant 6.3, $P_i.cp[j] \neq T$. By Invariant 6.4, $CP[i] = X$. Hence, the token will eventually be routed to P_i. The *eval-cps* subroutine evaluates channel predicates until it finds at least one false predicate, or until all predicates are true. When the false predicate is found, P_i will paint a state red and then send the token. The process receiving the token will advance the cut.

Hence, if G is a predecessor to H, then the cut will be advanced.

Now we show that when the current cut is exactly H, the detect flag will be set. By part 4 of Lemma 4.1, we know that $\forall i : color[i] = green$. Since no channel predicates are false, Invariant 6.3 ensures that $\forall i, j : P_i.cp[j]$ is neither F_s nor F_r. If the token is on P_i, eventually $\forall j : P_i.cp[j]$ will be set to T. Hence, eventually $CP[i]$ will be set to T. Eventually, the token will visit every P_i such that $CP[i]$ is X, and set that flag to T. When the token visits the last such process, detect will be set to true. □

6.4 Analysis of Distributed GCP Detection Algorithm

The complexity for the GCP detection algorithm is similar to that of the single-token WCP detection algorithm.

The only difference in the time complexity of the GCP detection algorithm and the single-token WCP detection algorithm are the subroutines *eval-cps*, *update-channels* and the additional work to send and receive update messages. First note that, as in the WCP detection

algorithm, a state can be painted red at most once. Since there are $O(mn)$ states, the total amount of work performed by all processes is $O(n^2m)$.

The work required to evaluate a single channel predicate is at most $O(m)$ since there can be no more than m messages in a channel. Note that many useful channel predicates can be evaluated in $O(1)$ time. Since each channel predicate can be evaluated at most $2m$ times (once each time a message is either sent or received on the channel). The total amount of work by any process evaluating channels is $O(m^2)$. A similar analysis establishes $O(m^2)$ as the upper bound of the work by any process updating the channel state.

We now analyze the message complexity. The token is again $O(n)$ in size and is sent $O(mn)$ times. There are at most m update messages sent by any process, hence at most $O(mn)$ update messages total. We assume that the size of a message is constant, hence the total message complexity is $O(mn)$ messages and $O(n^2m)$ bits.

The additional space requirements of GCP detection over WCP detection are the S, R and IR sets. Each of these sets holds at most m messages. Hence the space complexity for GCP detection is $O(mn)$ (as in WCP) for each process.

7 Discussion

As shown earlier, our algorithms take $O(mn)$ messages to detect a WCP. Further, if we assume that the computation time is negligible in comparison to communication time, then the algorithm also takes $O(mn)$ time in the worst case. We argue in this section that mn is also the lower bound on the time complexity of solving this problem. Formally, the 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. This restriction models the online nature of the detection algorithm. 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.

Proposition 7.1 *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. We now show that for many special cases, WCP and GCP can be detected with lower overhead. For example, if the WCP predicate is known to be stable and it is required that we detect any cut that satisfies the predicate (not necessarily the first), then it is sufficient to keep a buffer for a single snapshot at each process. On arrival of a new local snapshot, the old snapshot is deleted. This reduces the space overhead significantly.

As another example, assume that for a WCP, a snapshot includes not only the timestamp, but also the list of processes from which it has received messages since the last snapshot. Also, assume that all processes in the system are involved in the WCP. In this case, the monitor process needs to check only these processes to color their states red. Since there are $O(mn)$ receives and $O(mn)$ states in the entire system, it is easily shown that the total amount of work for WCP reduces to $O(mn)$ from $O(n^2m)$. Note that the lower bound of $O(n^2m)$ presented in [8] does not hold in this case because the algorithm uses the list of senders in addition to vectors.

8 Conclusions

We have presented distributed algorithms for detecting weak and generalized conjunctive predicates. These algorithms reduce 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 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, and monotonicity of channel predicates provides this ability for generalized 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 a method of using g tokens which permits up to g states to be eliminated at each step of the algorithm. In future work we plan to implement these algorithms as part of a distributed programming environment and analyze the speedup that can be achieved in the average case.

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*, Pennsylvania State University, August 1994.

[6] V. K. Garg, C. Chase, J. R. Mitchell, R. Kilgore, "Detecting Conjunctive Channel Predicates in a Distributed Programming Environment," University of Texas at Austin, Technical Report TR-PDS-94-02, June 1994.

[7] 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.

[8] 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.

[9] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems", *Proc. of the 21-st Intl. Conf. on System Sciences*, Vol. 2, Jan 1988, pp 166 – 175. 1990, pp. 134 – 141.

[10] 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.

[11] 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.

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

[13] 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.

[14] R. Schwartz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Germany, December 1992.

[15] 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.