Monitoring Functions on Global States of Distributed Programs *

Alexander I. Tomlinson
alext@pine.ece.utexas.edu

Vijay K. Garg
garg@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas 78712

June 10, 1994

Abstract

The domain of a global function is the set of all global states of an execution of a distributed program. We show how to monitor a program in order to determine if there exists a global state in which the sum $x_1 + x_2 + \ldots + x_N$ exceeds some constant K, where x_i is defined in process i. We examine the cases where x_i is an integer variable and where x_i is a boolean variable. For both cases we provide algorithms, prove their correctness and analyze their complexity.

1 Introduction

As a distributed program executes, each process proceeds through a sequence of local states. The set S of all local states is partially ordered by Lamport's *happens before* relation[Lam78], denoted by $\rightarrow$. A global state is a subset of S in which no two elements are ordered by $\rightarrow$. Given a global state c of some execution, it is impossible to determine if c actually occurred in an execution. However, it is known that c is consistent with some global state that did occur in the execution [CL85, Mat89]. In other words, it is possible that c occurred but there is no way to determine if it actually did.

A global function is a function whose domain is the set of all global states of a given execution. In this paper we present algorithms for monitoring the value of a global function in a distributed program while the program is executing. In particular, we show how to monitor a global function f in order to detect, if for any global state c in the underlying program execution, $f(c)$ exceeds some constant K. Without making restrictions on f, this problem is intractable. Therefore we will consider the case where f is a sum of local variables distributed among the processes in the computation.

Let x_i denote a variable at process P_i, $1 \leq i \leq N$, whose value is in domain D. We assume that x_i has a defined value in every state at P_i. The problem we consider in this paper is to determine if there exists a global state such that the sum $x_1 + x_2 + \ldots + x_N$ is greater than or equal to some constant $K \in D$. We consider two cases of this problem. The first case is for two integer variables and N processes; we refer to this case as "Two integer variables". The second case is for N boolean variables and N processes, which we refer to as "N Boolean variables".

*Research supported in part by NSF Grant CCR 9110605, TRW faculty assistantship award, a grant from IBM, and an MCD University Fellowship.

An example of the first case is $x_1 + x_2 > K$, where each variable x_i is an integer. For this case we present a decentralized and centralized algorithm which determines if there exists a global state (in a given execution) such that $x_1 + x_2 > K$. Following this, we generalize D, $+$, and $>$ so that we may use the same results for detecting any of the following for a given execution:

- $x_1 + x_2 > K$ is true in some global state ($x_i \in Reals$).
- $x_1 * x_2 > K$ is true in some global state ($x_i \in Reals$, $x_i \geq 1$).
- $x_1 \wedge x_2$ is true in all global states (x_i boolean).
- $x_1 \vee x_2$ is true in all global states (x_i boolean).
- $x_1 \wedge x_2$ is true in some global state (x_i boolean).

In the second case, N Boolean variables, we determine if there exists a global state such that at least K of these N variables are true. This has many applications, such as the K-mutual exclusion problem, which requires that no more than K processes ever have access to some resource. If boolean variable x_i is true when process P_i has access to the resource, then the global function $x_1 + x_2 + \ldots + x_N > K$ can be used to monitor an executing program to detect if there is a global state which violates K-mutual exclusion.

In section 2 we survey related work and describe how our research relates to other research in the field. In section 3 we present our model of distributed computation which is based on Lamport's happens before relation and partially ordered sets. The next two sections describe our research on each of the two cases of global functions that we consider in this paper: "Two Integer Variables" and "N Boolean Variables". For each case, we define the problem, give algorithms to solve the problem, prove the correctness of the algorithms and analyze their complexity.

2 Related Work

The analysis of the executions of distributed programs has been an active area of research for several years. A program can be observed during execution in order to detect a behavior which has been previously specified. A behavior specification evaluates to either true or false for a given program execution, hence they are often referred to as predicates, or breakpoints.

Before reviewing related research, we present a taxonomy of behavioral predicates. Behavior specifications can be divided into two categories depending on whether or not they are based on global states. This dichotomy of global-state based and non-global-state based specifications corresponds roughly to safety and progress properties. Those which are based on global states can be further divided into stable and unstable predicates. The work presented in this paper falls into the unstable global-state based category.

A global-state based behavioral predicate is one which is evaluated on the set of all global states in the execution. A good example is mutual exclusion, which can be stated as follows: there does not exist a global state such that two processes have access to the same resource (i.e., critical section). In fact, invariants are usually interpreted as global-state based. For example, a property P is invariant in an execution if it is true in all global states of the execution.

Non-global-state based behavioral predicates cannot be evaluated on global states. An example of this type is the linked predicate, which can be stated as follows: the property "P then Q" is true

in an execution if there exists two local states ordered by Lamport's happens before relation such that P is true in the first state and Q is true in the second. This cannot be evaluated on a global state because the two states are not concurrent.

Global-state based approaches can be divided further into stable and unstable predicates. A stable predicate stays true once it becomes true while an unstable predicate may oscillate between true and false. An example of a stable predicate is: "each process has received the token at least once". Clearly, in any execution, once this predicate becomes true, it stays true. An example of an unstable predicate which would be useful in analyzing an implementation of the two phase commit protocol is: "all processes are in the *ready* state".

2.1 Global-state based approaches

2.1.1 Stable predicates

Chandy and Lamport [CL85] invented the global snapshot, which is the basis for almost all subsequent research on detecting stable predicates. A global snapshot is a freeze-frame picture of a distributed computation. The picture corresponds to a global state of the computation with one caveat: the global state is not guaranteed to have occurred. However, it is guaranteed to be consistent with at least one global state that did occur.

Their approach to detecting stable predicates is conceptually simple: take repeated snapshots. Since the predicate is stable, eventually a snapshot will detect the predicate if it ever becomes true. Bouge [Bou87], and Spezialetti and Kearns [SK86] improved the efficiency of the global snapshot algorithm when used for repeated snapshots.

These approaches do not work for unstable predicates because the predicate may become true and then false again between two snapshots. An entirely different approach is required for unstable predicates.

2.1.2 Unstable predicates

The work presented in this paper falls into the category of unstable predicates, which is perhaps the least explored of the categories reviewed here.

Garg and Waldecker [GW94] define a class of unstable predicates called weak conjunctive predicates. A weak conjunctive predicate consists of a conjunction of local predicates such as $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, where p_i is evaluated in process i. It is defined to be true in an execution if there exists a global state in that execution such that the expression evaluates to true. Optimal algorithms for detecting weak conjunctive predicates appear in [GW94]. In [GCKM94], Garg and Chase extend weak conjunctive predicates to include predicates on the state of message channels.

Garg and Waldecker [GW92] also define strong conjunctive predicates which, like their weak counterparts, consist of a conjunction of local predicates, $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$. The predicate evaluates to true (for a particular execution) when every sequence of global states consistent with the execution contains a global state which satisfies the conjunctive boolean expression. Because strong conjunctive predicates depend on the relationship between global states in the computation, one can make an argument that they are not global-state based. However, we classify them as global state based because of their close relationship with weak conjunctive predicates. Optimal algorithms for detecting strong conjunctive predicates appear in [GW92].

2.2 Non-global-state based approaches

A general method for detecting non-global-state based predicates is to construct the lattice associated with a distributed execution [CM91, DJR93] and then analyze it to detect the behavior [BM93, BR94, JJJR94]. A node of this lattice represents a possible global state[1] of the distributed computation and an edge represents an event that changes the global state of the computation. This approach can detect a very broad class of behaviors, including global-state based predicates. The drawback is that the cost is high since it requires building a lattice, which has exponential time complexity in the number of local states in the computation.

To avoid the cost of using a lattice, many researchers have defined classes of behavioral predicates whose detection do not require building a lattice. One of the first of these was linked predicates, introduced by Miller and Choi [MC88]. Linked predicates describe a causal sequence of local states where each state in the sequence satisfies a specific local predicate. The behavior "an occurrence of local predicate p is causally followed by an occurrence of local predicate q" is an example of a linked predicate. Algorithms for linked predicates appear in [HPR93, MC88].

Hurfin et al. [HPR93] generalized linked predicates to a broader class called atomic sequences of predicates. In this class, occurrences of local predicates can be forbidden between adjacent predicates in a linked predicate. The example given above for linked predicates could be expanded to include: "q follows p and r never occurs in between" (note that p,q, and r could be evaluated in different processes).

In [FRGT94] we introduced regular patterns which are based upon regular expressions. A behavior is specified by a regular expression of local predicates. For example pq^*r is true in a computation if there exists a sequence of consecutive local states $(s_1, s_2, \ldots, s_n)$ such that p is true in s_1, q is true in $s_2, \ldots, s_{n-1}$, and r is true in s_n. Note that the states in the sequence need not belong to the same process – two states are consecutive if they are adjacent in the same process or one sends a message and the other receives it.

In [GTFR94] we extended the results of [FRGT94] to introduce a class of behaviors which include regular patterns as a special case. We designed a logic for expressing these properties and presented an efficient decentralized algorithm for detecting formulas in the logic. We also defined a class of algorithms called efficient passive detection algorithms (EPDA) and showed that other algorithms in this class cannot detect more than our algorithm. Briefly, an EPDA detects a property of some underlying computation. We use the term passive because the algorithms can only observe the computation (they cannot initiate or inhibit the sending or receiving of messages; and they cannot alter the control flow of the observed computation).

3 Model and Notation

We use the following notation for quantified expressions: (Op FreeVars : Range of FreeVars : Expr). Op can be any commutative associative operator (e.g., $min, \cup, +$). For example $(min\, i : i \in \mathcal{R} : f(i))$ is the minimum value of $f(i)$ for all i such that $i \in \mathcal{R}$.

Any distributed computation can be modeled as a decomposed partially ordered set (deposet) of process states [Fid89]. A deposet is a partially ordered set $(S, \leadsto)$ such that:

1. S is partitioned into N sets S_i, $1 \leq i \leq N$.

[1] In contrast with the poset representation, where a node represents a local state.

2. Each set S_i is a total order under some relation $\triangleright$, and $\triangleright$ does not relate two elements which are in different partitions.

3. Let $\rightarrow$ be the transitive closure of $\triangleright \cup \rightsquigarrow$. Then $(S, \rightarrow)$ is an irreflexive partial order.

An execution that consists of processes $P_1, P_2 \ldots P_N$ can be modeled by a deposet where S_i is the set of local states at P_i which are sequenced by $\triangleright$; the $\rightsquigarrow$ relation represents the ordering induced by messages; and $\rightarrow$ is Lamport's *happens before* relation[Lam78]. The concurrency relation on S is defined as $u \| v \; = \; (u \not\rightarrow v) \wedge (v \not\rightarrow u)$.

Given a local state $\sigma \in S$, we denote the value of a variable, say x, in state σ by $\sigma.x$. A global state is a subset $c \subseteq S$ such that no two elements of c are ordered by $\rightarrow$. We define C to be the set of all global states in $(S, \rightarrow)$. We also use the terms "cut" and "antichain" to refer to an element of C. For the remainder of this paper, the term "state" refers to a local state. A "chain" is a set of states which are totally ordered by $\rightarrow$. For example, each set S_i is a chain.

If $(u \rightarrow v)$ then $max(u, v) = v$ and $min(u, v) = u$. Since max and min are commutative and associative, the maximum and minimum element of any chain in $(S, \rightarrow)$ are well-defined. The unit elements of the max and min operators are $\perp$ and $\top$ respectively. Thus max applied to a zero length chain returns $\perp$. We require that $(\forall u : u \in S : \perp \rightarrow u \; \wedge \; u \rightarrow \top)$, and also that $\perp \rightarrow \perp$ and $\top \rightarrow \top$.

The predecessor and successor functions are defined as follows for $u \in S$ and $1 \leq i \leq N$:

$$pred.u.i = (\max v : v \in S_i \; \wedge \; v \rightarrow u : v)$$

$$succ.u.i = (\min v : v \in S_i \; \wedge \; u \rightarrow v : v)$$

Thus if $(pred.u.i = v)$ then v is the maximum element in $(S_i, \rightarrow)$ which happens before u, or $\perp$ if no element in S_i happens before u.

An external event is the sending or receiving of a message. The nth interval in P_i (denoted by (i, n)) is the subchain of $(S_i, \rightarrow)$ between the $(n-1)^{th}$ and n^{th} external events. For a given interval (i, n), if n is out of range then (i, n) refers to $\perp$ or $\top$. The notion of intervals is useful because the relation of two states belonging to the same interval is a congruence with respect to $\rightarrow$. Thus, for any two states s and s' in the same interval and any state u which is not in that interval: $(s \rightarrow u \iff s' \rightarrow u)$ and $(u \rightarrow s \iff u \rightarrow s')$. We take advantage of this in our algorithms by assigning a single timestamp to all states belonging to the same interval. Due to this congruence, the *pred* and *succ* functions and the $\|$ relation are well-defined on intervals.

3.1 Vector clocks

In a system with N processes, a vector clock [Mat89, Fid89, Ray92] is a function which associates a vector of N non-negative integers with each state of the poset. Vector clocks are useful because they implement the *pred* function, and they encode Lamport's happens before relation. Given a state σ, we denote its vector clock value by $\sigma.v$. The important property of vector clocks is

$$\sigma \rightarrow \sigma' \iff \sigma.v < \sigma'.v$$

where the $<$ relation on vectors u and v is defined as:

$$u < v \iff (\forall i :: u[i] \leq v[i]) \; \wedge \; (\exists i :: u[i] < v[i])$$

The algorithm for implementing vector clocks is quite simple. We describe it here for process P_i; the algorithm is the same for each process. Initially $v[i]$ equals 1 and all other components equal zero. Upon sending a message, the message is tagged with the current value of v and then $v[i]$ is incremented by one. Upon receiving a message tagged with u (all received messages will have a message tag) execute $v[j] := max(v[j], u[j])$ for all j and then increment $v[i]$. This is the entire algorithm. In order to compare two states, we need the value of v in that state.

The solution to "Two Integer variables" uses a 2×2 matrix clock which is built upon the ideas of vector clocks. The solution to "N Boolean variables" uses traditional vector clocks to compare individual elements of the poset, or equivalently, to determine if one state happens before another state.

4 Two Integer Variables

The case where each x_i is an integer variable is useful for detecting potential violations of a limited resource. For example, consider a server which can handle at most 5 connections at a time. Client processes P_1 and P_2 each have a variable x_1 and x_2 which indicates the number of connections it has with the server. The predicate $(x_1 + x_2 > 5)$ indicates a potential error.

A formal problem statement for this case is shown below. It says that there exists states σ_1, σ_2 (in processes P_1 and P_2 respectively) which are concurrent, and the sum of the values of x_1 in σ_1 and x_2 in σ_2 is greater than K. (Note that $\sigma_1.x$ refers to the value of x_1 in state σ_1.)

$$(\exists \sigma_1, \sigma_2 : \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2 \ \wedge \ \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x > K) \qquad \text{(PS1)}$$

In this section we present two algorithms for detecting PS1. The decentralized algorithm runs concurrently with the underlying program and can be used for online detection of the predicate. The centralized version is decoupled from the underlying program and can run concurrently with the underlying program or post-mortem (i.e., after the underlying program terminates). We formally prove that both algorithms are sound (if the predicate is detected, then it has occurred) and complete (if the predicate occurs, then it is detected). Before describing the centralized and decentralized algorithms, we describe the ideas behind them, describe and prove mechanisms used by the algorithms (i.e., 2×2 matrix clock), and present some results which are necessary for proving the algorithms. The work presented in this section is based on [TG93].

4.1 The main idea behind the algorithms

To detect PS1 we need to determine if there exists a global state in which $x_1 + x_2 > K$. First consider a situation in which P_1 and P_2 do not send or receive any messages. Then every state in S_1 is concurrent with every state in S_2. In this case we could evaluate $maxx_1(S_1) + maxx_2(S_2) > K$, where $maxx_i(S_i)$ is the maximum value of x_i in the set of states S_i. This expression would be true if and only if PS1 were true.

We use this basic idea in our algorithm, but there are complications resulting from message communication. Care must be take to ensure that $x_1 + x_2$ is evaluated in a global state if and only if the global state is valid. Each time a message m is received at P_1 or P_2, we compute $S_1^m \subseteq S_1$ and $S_2^m \subseteq S_2$ such that every state in S_1^m is concurrent with every state in S_2^m. Thus

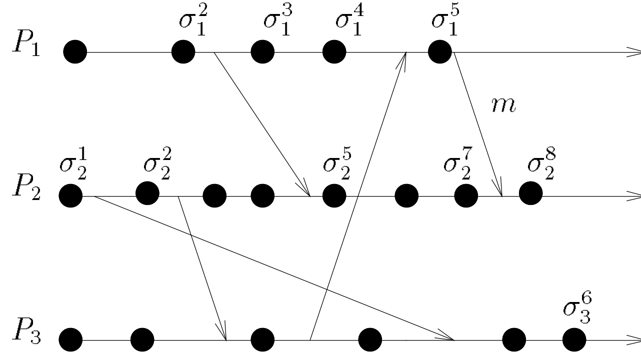

Figure 1: Demonstration of the *pred* function: $pred.\sigma_1^5.2 = \sigma_2^2$, $pred.\sigma_3^6.2 = \sigma_2^2$, $pred.\sigma_1^4.2 = \perp$.

if $maxx_1(S_1^m) + maxx_2(S_2^m) > K$, then we know that PS1 holds and our algorithm is sound. To demonstrate that this algorithm is complete we need to prove that $\sigma_1 \| \sigma_2$ if and only if there exists a message m received at P_1 or P_2 such that $\sigma_1 \in S_1^m$ and $\sigma_2 \in S_2^m$. We must also show how to determine S_1^m and S_2^m when m is received, and how to evaluate $maxx_1(S_2^m)$ and $maxx_2(S_2^m)$.

S_1^m is a sequence of states at P_1, $(\sigma_1^{lo_1+1}, \ldots, \sigma_1^{hi_1})$, and S_2^m is a sequence at P_2, $(\sigma_2^{lo_2+1}, \ldots, \sigma_2^{hi_2})$. Given a message m which has been received at P_1 or P_2 we need to be able to determine values for lo_1, hi_1, lo_2, hi_2. This is accomplished by keeping track of predecessors of states in P_1 and P_2. Consider a state $\sigma_i \in S_i$. The predecessor of σ_i in P_j, denoted $pred.\sigma_i.j$, is the latest state in P_j which happens before σ_i. Figure 1 shows an example. Notice that the predecessor or σ_i^n on P_i is just the previous state σ_i^{n-1}.

Suppose a message m is received in some state $\sigma \in S_1 \cup S_2$. Then we define lo_1, hi_1, lo_2, hi_2 as follows:

$$\begin{aligned} \sigma_1^{hi_1} &= pred.\sigma.1 & \sigma_2^{lo_2} &= pred.\sigma_1^{hi_1}.2 \\ \sigma_2^{hi_2} &= pred.\sigma.2 & \sigma_1^{lo_1} &= pred.\sigma_2^{hi_2}.1 \end{aligned}$$

Applying these definitions to figure 1, we see that $S_1^m = (\sigma_1^3, \sigma_1^4, \sigma_1^5)$, and $S_2^m = (\sigma_2^3, \sigma_2^4, \sigma_2^5, \sigma_2^6, \sigma_2^7)$, Thus, if we can evaluate the above *pred* functions, then we can determine S_1^m and S_2^m. We use a 2×2 matrix clock to enable us to evaluate these expressions.

4.2 Finding lo_1, hi_1, lo_2, hi_2 with a 2×2 matrix clock

Instead of associating values for lo_1, hi_1, lo_2, hi_2 with each state, we associate these values with intervals. Thus S_1^m and S_2^m now represent sequences of intervals, which themselves are sequences of states.

To evaluate the *pred* functions, we use a 2×2 matrix clock as described by Raynal [Ray92]. In this section we present an algorithm for maintaining the matrix clock and prove that from it we can determine lo_1, hi_1, lo_2, hi_2. Once we have these values, we can find S_1^m and S_2^m, and then determine if PS1 is satisfied.

The matrix clock algorithm is presented in figure 3. The algorithm is easier to understand by

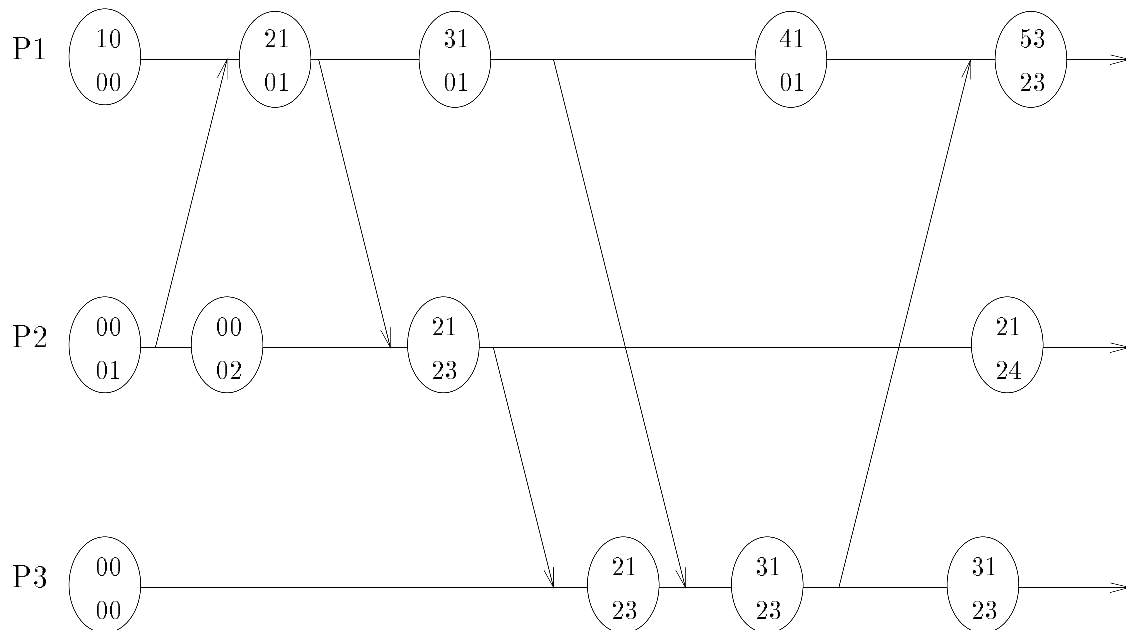

Figure 2: Matrix clock example.

noticing the vector clock algorithm embedded within it. If the row index is held constant, then it reduces to the vector clock algorithm. Figure 2 shows values of the matrix clock and message tags on an example run. Note that row 1 of P_1's matrix is a traditional vector clock restricted to indices 1 and 2, and row 2 equals the value of P_2's vector clock at a state in the "past" of P_1. Similar properties hold for P_2's matrix clock.

Let M_k^n denote the value of the matrix clock in interval (k, n). The following description applies to an $N \times N$ matrix clock in a system with N processes. The 2×2 matrix is the upper left submatrix of the $N \times N$ matrix. We describe the information contained in a matrix clock M_k^n at interval (k, n) in three steps. First we explain the meaning of the kth diagonal element. Then we explain the meaning of the kth row, followed by the entire matrix.

The kth diagonal element, $M_k^n[k, k]$, is an interval counter at P_k. Thus for any interval (k, n), the following is true: $M_k^n[k, k] = n$. Row k of M_k^n is equivalent to a traditional vector clock. In fact, when we say "P_k's vector clock" we are referring to row k of P_k's matrix clock. Since the vector clock implements the $pred$ function, row k of M_k^n can be used to find predecessors of $(k, n) = (k, M_k^n[k, k])$ as follows: the predecessor of $(k, M_k^n[k, k])$ on process j is the interval $(j, M_k^n[k, j])$. In fact, this applies to all rows: the predecessor of $(i, M_k^n[i, i])$ on process j is the interval $(j, M_k^n[i, j])$. Furthermore, row k of M_k^n equals the diagonal of M_k^n. Thus we can use row k to find the $pred.(k, n).j$ for $j \neq k$, and then use row j to find $pred.(pred.(k, n).j).i$ for $i \neq j$.

The matrix clock can be summarized in three simple rules. First, the kth diagonal element corresponds to some interval on P_k. Second, row i is the value of P_i's vector clock in interval $(i, M_k[i, i])$ and third, row k of M_k equals the diagonal. The meaning of a matrix clock is formally stated and proven in lemma 1.

To initialize:

$M_k[\cdot,\cdot] := 0;$
if $(k = 1 \ \vee \ k = 2)$ **then**
  $M_k[k,k] := M_k[k,k] + 1;$
end_if

To send a message:

Tag message with $M_k[\cdot,\cdot]$;
if $(k = 1 \ \vee \ k = 2)$ **then**
  $M_k[k,k] := M_k[k,k] + 1;$ *increment local clock*
end_if

Upon receipt of a message tagged with $W[\cdot,\cdot]$:

for $i := 1$ **to** 2 **do**
  if $(M_k[i,i] < W[i,i])$ **then**
    $M_k[i,\cdot] := W[i,\cdot];$ *copy vector clock for* $(i, W[i,i])$
  end_if
end_for
if $(k = 1 \ \vee \ k = 2)$ **then**
  $M_k[k,k] := M_k[k,k] + 1;$ *increment local clock*
  $M_k[k,\cdot] := diagonal(M_k);$
end_if

Figure 3: Algorithm for maintaining matrix clock $M_k[1..2, 1..2]$ at P_k, $1 \leq k \leq N$

Lemma 1 *(M1)* $M_k^n[k,k] = n$
(M2) $i \neq j \;\Rightarrow\; (j, M_k^n[i,j]) = pred.(i, M_k^n[i,i]).j$
(M3) $M_k^n[i,i] = M_k^n[k,i]$

Proof: Each part is proven by induction. In the base case, $n = 1$ and (k,n) is an initial interval. Induction applies to $n > 1$.

(M1) **Base** ($n = 1$): Initial value of $M_k^n[k,k] = 1$.
Induction ($n > 1$): We assume $M_k^n[k,k] = n$ and show that $M_k^{n+1}[k,k] = n+1$. P_k enters interval $(k, n+1)$ only after sending or receiving a message. From the program text it is clear the kth diagonal element is incremented by one. Thus $M_k^{n+1}[k,k] = M_k^n[k,k] + 1 = n+1$.

(M2) **Base** ($n = 1$): Since $n = 1$, M_k^n is the initial value of the matrix clock in P_k. Thus, $i \neq j$ implies that $M_k^n[i,j] = 0$, and hence $(j, M_k^n[i,j]) = (j, 0) = \perp$. Thus we need to show $pred.(i, M_k^n[i,i]).j = \perp$ also. Suppose $i \neq k$, then by initial value of M_k^n, we know $(i, M_k^n[i,i]) = (i, 0)$, thus $pred.(i, M_k^n[i,i]).j = pred.(i,0).j = \perp$. Now suppose $i = k$, then by initial value of $M_k^n[i,i]$ we know that $(i, M_k^n[i,i]) = (i,1)$. Since no interval precedes $(i,1)$, we know $pred.(i,1).j = \perp$.
Induction ($n > 1$): Assume true for (k,n) and every interval in the past of (k,n). Suppose the event initiating $(k, n+1)$ is a message send, then $M_k^{n+1} = M_k^n$ except for $M_k^{n+1}[k,k] = M_k^n[k,k]+1$. Since the event is a message send, (k,n) and $(k,n+1)$ have the exact same predecessors. Thus $pred.(i, M_k^{n+1}[i,i]).j$ equals $pred.(i, M_k^n[i,i]).j$, which by induction hypothesis, equals $(j, M_k^n[i,j])$, which is equal to $(j, M_k^{n+1}[i,j])$ since $i \neq j$ and only element $M_k^n[k,k]$ changes.

Suppose the event initiating $(k, n+1)$ is a message receive tagged with matrix clock W. By the induction hypothesis, W satisfies M2. Note also that M2 relates elements in a single row of M, thus by copying rows from W to M, M2 is not violated. Since all rows $i \neq k$ are either copied from W or not changed, then M2 holds for rows $i \neq k$ in M_k^{n+1}. It remains to show that M2 holds for row k of M_k^{n+1}. The predecessor of $(k, n+1)$ on P_j is the greater of $M_k^n[k,j]$ and $W_k^n[j,j]$. This is exactly the value assigned to $M_k^{n+1}[k,j]$. Thus $(k, M_k^{n+1}[k,j]) = pred.(k, M_k^{n+1}[k,k]).j$ and M2 holds for row k.

(M3) **Base** ($n = 1$): True by initial assignment to matrix.
Induction ($n > 1$): Assume $diag(M_k^n) = M_k^n[k,\cdot]$, show $diag(M_k^{n+1}) = M_k^{n+1}[k,\cdot]$. P_k enters interval $(k, n+1)$ only after sending or receiving a message. In the case of a send, $M_k^n[k,k]$ is incremented by one. Thus the diagonal will still be equal to row k. In the case of a message receive, the last statement sets row k to the diagonal.

■

Lemma 2 shows how to use the matrix clock M_1^n to determine lo_1, hi_1, lo_2, and hi_2 for any interval $(1,n)$. The procedure for any interval on P_2 is similar.

Lemma 2 *The following expressions hold for any interval* $(1,n)$ *with matrix clock* M_1^n*:*

$$\begin{array}{lclcl}
(1, hi_1) & = & pred.(1,n).1 & = & M_1^n[1,1] - 1 \\
(2, hi_2) & = & pred.(1,n).2 & = & M_1^n[1,2] \\
(1, lo_1) & = & pred.(2, hi_2).1 & = & M_1^n[2,1] \\
(2, lo_2) & = & pred.(1, hi_1).2 & = & M_1^{n-1}[1,2]
\end{array}$$

Proof: The equivalences on the left are the previously stated definitions for hi_i and lo_i.

hi_1: The predecessor for $(1, n)$ on P_1 is $(1, n-1)$. Thus, by M1, $hi_1 = M_1^n[1,1] - 1$.

hi_2: Also by M1, we know $pred.(1,n).2 = pred.(1, M_1^n[1,1]).2$, which by M2, equals $(2, M_1^n[1,2])$. Thus $hi_2 = M_1^n[1,2]$.

lo_1: We need to evaluate $pred.(2, M_1^n[1,2]).1$. Since the diagonal equals row 1, this is equal to $pred.(2, M_1^n[2,2]).1$. And by M2, this equals $(1, M_1^n[2,1])$. Thus $lo_1 = M_1^n[2,1]$.

lo_2: We need to evaluate $pred.(1, M_1^n[1,1]-1).2$, which equals $pred.(1, n-1).2$. Using the matrix clock M_k^{n-1} in the preceding interval, this equals $pred.(1, M_1^{n-1}[1,1]).2$, which by M2, equals $(2, M_1^{n-1}[1,2])$. Thus $lo_2 = M_1^{n-1}[1,2]$.

■

We have shown how to determine the values of lo_1, hi_1, lo_2, and hi_2 in any interval at P_1 or P_2. Thus in the remainder of the paper we refer directly to lo_1, hi_1, lo_2, and hi_2 instead of referring to the matrix clock or the *pred* function.

4.3 Preliminary Results

The following lemma is important in developing efficient algorithms. It indicates that we need not maintain the value of x_i in every state at process P_i, but instead we can maintain the maximum value of x_i in each interval. As before, we use the notation $maxx_i.(i, n_i)$ to represent the maximum value of x_i in interval (i, n_i). Formally, $maxx_i.(i, n_i) = (\max \sigma : \sigma \in (i, n_i) : \sigma.x)$.

Lemma 3 $(\exists \sigma_1, \sigma_2 : \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2 \ \wedge \ \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x > K)$
$\iff \ (\exists n_1, n_2 : (1, n_1) \parallel (2, n_2) : maxx.(1, n_1) + maxx.(2, n_2) > K)$

Proof: The proof is straightforward. It uses the following properties: 1) congruence between intervals and states, 2) addition distributes over *max*, and 3) *max* is commutative and associative.

■

The next lemma justifies our use of S_1^m and S_2^m in detecting if PS1 is satisfied. It states that if two intervals $(1, i)$ and $(2, j)$ are concurrent then there exists a message m which defines sequences S_1^m and S_2^m and that $(1, i) \in S_1^m$ and $(2, j) \in S_2^m$. It also states that from each message m we can find S_1^m and S_2^m and that every interval in S_1^m is concurrent with every interval in S_2^m. This lemma proves that our approach is sound and complete. See figure 4 for a graphical representation of the case (in the proof) where KEY is in P_1.

Lemma 4 *Given that every state in process P_1 happens before some state in process P_2, or vice versa: two intervals $(1, i)$ and $(2, j)$ are concurrent* if and only if *there exist a message received just before some interval, say KEY, at P_1 or P_2 such that $(1, i) \in S_1^m$ and $(2, j) \in S_2^m$ where:*

$$\begin{array}{rcl} S_1^m &=& (\ (1, lo_1+1), \ldots (1, hi_1)\ ) \\ S_2^m &=& (\ (2, lo_2+1), \ldots (2, hi_2)\ ) \\ (1, hi_1) &=& pred.KEY.1 \\ (2, hi_2) &=& pred.KEY.2 \\ (1, lo_1) &=& pred.(2, hi_2).1 \\ (2, lo_2) &=& pred.(1, hi_1).2 \end{array}$$

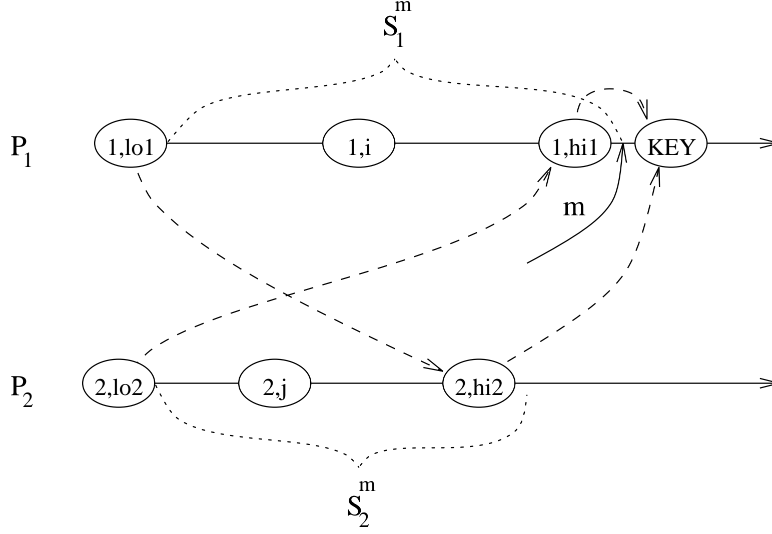

Figure 4: Relationship among intervals when KEY is in P_1. Dashed arrows represent the *pred* function, and the solid arrow represents some message being received at P_1.

Proof:
Proof of $\Rightarrow$: Assume $(1,i)\|(2,j)$.
Assume without loss of generality that every state in S_2 happens before some state in S_1. Then we know $succ.(2,j).1$ exists. If $succ.(1,i).2$ exists and $succ.(1,i).2 \rightarrow succ.(2,j).1$, then let $KEY = succ.(1,i).2$ (case 1), otherwise let $KEY = succ.(2,j).1$ (case 2). We prove case 2 (the proof for case 1 is nearly identical). Thus, at this point we know that: KEY is the first interval on P_1 such that $(2,j) \rightarrow KEY$, and $succ.(1,i).2$ (which may not even exist) doesn't precede KEY.

Proof of $i \leq hi_1$: We know $(1,i)$ must occur before KEY (otherwise $(1,i) \rightarrow (2,j)$). This is equivalent to saying $i \leq hi_1$ where $(1,hi_1) = pred.KEY.1$.

Proof of $lo_2 < j$: Since KEY is the first interval on P_1 such that $(2,j) \rightarrow KEY$, and since $(1,hi_1)$ precedes KEY, we know $(2,j) \not\rightarrow (1,hi_1)$. This is equivalent to saying $lo_2 < j$ where $(2,lo_2) = pred.(1,hi_1).2$.

Proof of $j \leq hi_2$: By definition of predecessors, any interval on P_2 which occurs after KEY's predecessor on P_2, say $(2,hi_2)$, does not happen before KEY. Then since $(2,j) \rightarrow KEY$ we know that $(2,j)$ does not happen after $(2,hi_2)$. This is equivalent to saying $j \leq hi_2$ where $(2,hi_2) = pred.KEY.2$.

Proof of $lo_1 < i$: Since $succ.(1,i).2$ does not precede KEY (as a result of assuming case 2; also note: it may be the case that $succ.(1,i).2$ doesn't exist) and since $(2,hi_2) \rightarrow KEY$, we know that $(1,i)$ cannot precede any interval which precedes $(2,hi_2)$. Therefore $lo_1 < i$ where $(1,lo_1) = pred.(2,hi_2).1$. (If we assumed case 1, this same argument would apply).
Proof of $\Leftarrow$: Assume $lo_1 < i \leq hi_1$ and $lo_2 < j \leq hi_2$.
Since $(1,lo_1) = pred.(2.hi_2).1$ we know that any interval on P_1 which follows lo_1 cannot happen before $(2,hi_2)$. Therefore since $(1,lo_1) \rightarrow (1,i)$ (by $lo_1 < i$) we know $(1,i) \not\rightarrow (2,hi_2)$. And since

$j \leq hi_2$ we know that $(1,i) \not\rightarrow (2,j)$.
Similarly, $(2,j) \not\rightarrow (1,i)$. ■

4.4 Overview of Algorithms

The centralized and the decentralized algorithms each gather the same information from the underlying program as it executes. They differ in what they do with the information. This section explains what the information is, and how it is gathered. Everything in this section applies to both algorithms.

Each process P_i, for $i \in \{1,2\}$, must be able to evaluate $maxx.(i,n)$ for each interval (i,n). Implementation of the $maxx$ function is straight forward. Additionally, for the algorithms to be complete, it is required that every state in P_2 occur before at least one state in P_1. This can be accomplished in one of two ways: by the use of a `<FINAL>` message, or by using periodic update messages. In the first approach P_2 sends a `<FINAL>` message to P_1 immediately before P_2 terminates and delivery of this message is delayed until immediately before P_1 terminates. The second approach can be used if P_1 and P_2 do not normally terminate. To use this approach, P_2 must periodically send a message to P_1.

The algorithms are based on lemma 4. Each message received at P_1 or P_2 defines sequences of intervals S_1^m and S_2^m such that every interval in S_1^m is concurrent with every interval in S_2^m. Since lemma 4 uses an *if and only if* relation, every pair of concurrent intervals at P_1 and P_2 will appear in the sequences that result from receiving some message. Thus if both P_1 and P_2 check the predicate for all pairs of states in these sequences each time a message is received, then the predicate will be detected.

4.5 Decentralized Algorithm

We describe the algorithm from P_1's point of view. The algorithm at P_2 is similar. Each time a message is received we evaluate lo_1, lo_2, hi_1, and hi_2. These values define a sequence of intervals at P_1 and at P_2. The sequence at P_i starts at $(i, lo_i{+}1)$ and ends at (i, hi_i). By lemma 4, every interval in the sequence at P_1 is concurrent with every interval at P_2. Thus we can find the maximum value of $maxx.(1,i)$ over all intervals $(1,i)$ in the sequence at P_1. We call this value $temp_x_1$ and define $temp_x_2$ similarly. If the sum of these two values is greater than K, then the predicate has occurred. Furthermore, since lemma 4 is stated with the *if and only if* relation, if the predicate occurs then this method will detect it.

S1: $temp_x_1 = (\ \max i\ :\ lo_1 < i \leq hi_1\ :\ maxx.(1,i)\ )$
S2: $temp_x_2 = (\ \max j\ :\ lo_2 < j \leq hi_2\ :\ maxx.(2,j)\ )$
S3: if $(temp_x_1 + temp_x_2 > K)$ then PREDICATE_DETECTED

To implement this, $temp_x_1$ can be computed locally. Then P_1 can send a message to P_2 containing $(temp_x_1, lo_2, hi_2)$, and P_2 can finish the calculation. This message is a debug message and is not considered an external event (i.e., does not initiate a new interval). Messages that are not debug messages are application messages. Theorem 1 states the correctness of this algorithm.

Theorem 1 *The condition in statement* S3, $(temp_x_1 + temp_x_2 > K)$, *is true if and only if* PS1 *holds.*

Proof: Statements S1, S2, and S3 are executed when ever a message m is received at either P_1 or P_2. Let i range over $\{1,2\}$. Then $temp_x_i$ equals $maxx(i, S_i^m)$. Thus by direct application of lemma 4, the theorem holds. ∎

4.5.1 Overhead and Complexity

Let A_i be the number of application messages sent and received by P_i, and let R_i be the number of application messages received by P_i. The message overhead consists of the number and size of the debug messages, and the size of message tags on application messages. P_1 sends one debug message to P_2 in each of the R_1 receive intervals at P_1. Similarly, P_2 generates R_2 debug messages. Thus the total number of debug messages generated by the decentralized algorithm is $R_1 + R_2$. The size of each debug message is 3 integers. Each application message carries a tag of 4 integers. The debug messages can be combined to reduce message overhead, however this will increase the delay between the occurrence of the predicate and its detection.

The memory overhead in P_1 arises from the need to maintain $maxx.(1,\cdot)$ for each of A_1 intervals. This can be reduced (for the average case) by a clever implementation since the elements of the array are accessed in order (i.e., the lower elements can be discarded as the computation proceeds). Likewise, the memory overhead for P_2 is A_2. Other processes incur only the overhead needed to maintain the matrix clock (i.e., 4 integers).

The computation overhead at P_1 consists of monitoring the local variable which appears in the predicate, and evaluation of the expression $(\max i : lo_1 < i \leq hi_1 : maxx.(1,i))$ for each debug message sent (R_1) and received (R_2). The aggregate complexity of this is at most $A_1(R_1 + R_2)$ since there are A_1 elements in $maxx.(1,\cdot)$. P_2 has similar overhead. Other processes have neither the overhead of monitoring local variables nor of computing the expression.

4.6 Centralized Algorithm

This version of the algorithm can be used as a checker process which runs concurrently with the underlying program, or which runs post-mortem. We describe the post-mortem version which reads data from trace files generated by P_1 and P_2. Since the trace files are accessed sequentially, the algorithm can be easily adapted to run concurrently with the underlying program by replacing file I/O with message I/O. First we explain what data is stored in the trace files, then we show how the predicate can be detected by one process which has access to both trace files.

Let R_1 be the number of receive intervals in P_1 and let $Q1[k]$, $1 \leq k \leq R_1$, be a record containing the values of lo_1,hi_1,lo_2, and hi_2 in the kth receive interval. Define R_2 and $Q2[1 \ldots R_2]$ similarly. The elements of both $Q1$ and $Q2$ must be checked to determine if one of the elements represents a key interval (i.e., satisfies the requirements of KEY in lemma 4.

By virtue of their construction, both $Q1$ and $Q2$ are already sorted in terms of all their fields. That is, for $Q = Q1$ or $Q = Q2$, and for every component $x \in \{lo_1, lo_2, hi_1, hi_2\}$, $Q[\cdot].x$ is a sorted array. This results from the fact that the elements are generated in order on a single process; thus the receive interval represented by $Q[k]$ happens before $Q[k+1]$.

P_1's trace file contains two arrays of data: $maxx.(1,i)$ for each interval $(1,i)$, and $Q1[1..R_1]$. Likewise P_2's trace file contains $maxx.(2,j)$ and $Q2[1..R_2]$. We have already demonstrated how to determine lo_1,hi_1,lo_2 and hi_2 for any interval in P_1 or P_2, and generating the values for the $maxx$ function is straight forward.

The trace files are analyzed in two independent passes. We describe a function $check(Q[1 \ldots R])$ such that the predicate has occurred if and only if $check(Q1[1 \ldots R_1])$ or $check(Q2[1 \ldots R_2])$ returns *true*. *Check* uses two heaps: $heap_1$ and $heap_2$. $Heap_p$ contains tuples of the form $\langle n, maxx.(p,n) \rangle$ where (p,n) is an interval in P_p. The first element of a tuple h is accessed via $h.interval$; the second element is accessed via $h.value$. The heap is sorted with the *value* field.

The algorithm maintains the following properties (HEAP holds at all times; I1 and I2 hold after S4 and before S5; k is a program variable):

$$\begin{aligned}
HEAP &\equiv (\forall h, p : h \in heap_p : heap_p.top().value \geq h.value) \\
I1 &\equiv (\forall i, p : Q[k].lo_p < i \leq Q[k].hi_p : \langle i, maxx.(p,i) \rangle \in heap_p) \\
I2 &\equiv (\forall p :: Q[k].lo_p < heap_p.top().interval \leq Q[k].hi_p)
\end{aligned}$$

HEAP is an inherent property of heaps: the top element, heap.top(), is the maximum element in the heap. Heaps are designed to efficiently maintain the maximum element of an ordered set. Statements S1 and S2 ensure I1, which states that in the kth iteration of the *for* loop, all intervals in the sequences defined by $Q[k]$ are represented in the heaps. Statements S3 and S4 ensure I2, which states that the top of $heap_p$ is in the sequence defined by $Q[k]$. The text of the *check* function is shown in figure 5 and theorem 2 proves its correctness.

Theorem 2 *There exists a value for program variable k such that at statement S5* $(heap_1.top().value + heap_2.top().value > K)$ if and only if $(\exists \sigma_1, \sigma_2 : \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2 \ \wedge \ \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x > K)$

Proof:
Proof of $\Rightarrow$: Let $\langle i, maxx.(1,i) \rangle = heap_1.top()$ and $\langle j, maxx.(2,j) \rangle = heap_2.top()$.
By lemma 3, it suffices to show that $(1,i) \| (2,j)$. By I2 we know that $Q[k].lo_1 < i \leq Q[k].hi_1$ and $Q[k].lo_1 < i \leq Q[k].hi_1$. Then by lemma 4 we know that $(1,i) \| (2,j)$.

Proof of $\Leftarrow$: Assume $(\exists \sigma_1, \sigma_2 : \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2 \ \wedge \ \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x > K)$.
By lemma 3 we know $(\exists i, j : (1,i) \parallel (2,j) : maxx.(1,i) + maxx.(2,j) > K)$. Referring to lemma 4, if the KEY interval is in P_1 then let $Q = Q1$ and let k equal the number of messages received at P_1 before KEY. Likewise for KEY in P_2. Then $Q[k]$ corresponds to KEY. Then by lemma 4 $Q[k].lo_1 < i \leq Q[k].hi_1$ and $Q[k].lo_2 < j \leq Q[k].hi_2$. By invariant I1, $\langle i, maxx.(1,i) \rangle$ is in $heap_1$, and $\langle j, maxx.(2,j) \rangle$ is in $heap_1$. And by invariant HEAP, $heap_1.top().value \geq maxx.(1,i)$, and $heap_2.top().value \geq maxx.(2,j)$. Thus in iteration k, $heap_1.top().value + heap_2.top().value > K$.

■

4.6.1 Overhead and Complexity

If we consider each record written to a trace file to be a debug message then the message complexity analysis is identical to the decentralized algorithm (except that the debug messages have a different destination).

P_1 and P_2 do not need to maintain $maxx.(\cdot,\cdot)$, thus the only memory overhead for each application processes is the 4 integers needed for the matrix clock.

The computation overhead consists of monitoring the local variables. The rest of the computation is offloaded to the checker process which uses the following data: $Q1[1 \ldots R_1]$, $Q2[1 \ldots R_2]$, $maxx.(1,i)$ for $1 \leq i \leq A_1$, and $maxx.(2,j)$ for $1 \leq j \leq A_2$. Recall from section 4.5.1 that R_i is the number of messages received by P_i and A_i equals R_i plus the number of messages sent by P_i .

Figure 5: *Check* function used in centralized algorithm for two integer variables.

function $check(Q[1\ldots R])$
    $n_1 := 1;\ n_2 := 1;$
    for $k := 1$ to R **do**
S1:   **while** $(n_1 \leq Q[k].hi_1)$ **do**
            $heap_1.insert(\ \langle n_1, maxx.(1, n_1)\rangle\ );$
            $n_1 := n_1 + 1;$
        end_while
S2:   **while** $(n_2 \leq Q[k].hi_2)$ **do**
            $heap_2.insert(\ \langle n_2, maxx.(2, n_2)\rangle\ );$
            $n_2 := n_2 + 1;$
        end_while
S3:   **while** $(heap_1.top().interval \leq Q[k].lo_1)$ **do**
            $heap_1.removetop();$
        end_while
S4:   **while** $(heap_2.top().interval \leq Q[k].lo_2)$ **do**
            $heap_2.removetop();$
        end_while
S5:   **if** $(heap_1.top().value + heap_2.top().value > K)$ **then**
            return $TRUE;$
        end_if
    end_for
    return $FALSE;$
end_function

Consider the call $check(Q1[1 \ldots R_1])$. On a heap of N elements, $insert()$ and $removetop()$ each cost $\Theta(lg\, N)$ and $top()$ costs $\Theta(1)$. Each element of $maxx.(1, \cdot)$ is inserted at most once and removed at most once from $heap_1$ for a total cost of $\Theta(A_1\, lg\, A_1)$. Similarly, the total cost of operations on $heap_2$ is $\Theta(A_2\, lg\, A_2)$. The outer loop executes R_1 times but is added to the cost of the heap operations since the heap operations are spread out through all R_1 iterations. Thus the total cost of $check(Q1[1 \ldots R_1])$ is $\Theta(R_1 + A_1\, lg\, A_1 + A_2\, lg\, A_2)$. Since $R_1 \leq A_1$, this simplifies to $\Theta(A\, lg\, A)$ where $A = max(A_1, A_2)$. Since $check$ is only called twice, the total complexity is $\Theta(A\, lg\, A)$.

4.7 Generalization

In this section, let σ_i represent a state in S_i. The algorithm given in this section for two integer variables detects the predicate $(\exists \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x > K)$. Our approach was to compute the maximum values of $\sigma_1.x$ and $\sigma_2.x$ and compare their sum to K. Since $+$ distributes over max (i.e., $a + max(b, c) = max(a + b, a + c)$), this is equivalent to computing

$$(max\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x)$$

and comparing it to K. The algorithm presented in this paper repeatedly calculates the above expression for different segments of a run; due to the idempotency of max it could be easily modified to determine the value of the expression for the entire run. The above expression uses max and $+$ over integers. This can be generalized to two operators, $\uparrow$ and $\oplus$, over a domain D which meet the following requirements:

Domain	D
Addition	$\uparrow : D \times D \mapsto D$
Multiplication	$\oplus : D \times D \mapsto D$
Commutativity	$a \uparrow b = b \uparrow a$
Associativity	$a \uparrow (b \uparrow c) = (a \uparrow b) \uparrow c$
Idempotency	$a \uparrow a = a$
Distributivity	$a \oplus (b \uparrow c) = (a \oplus b) \uparrow (a \oplus c)$
Unit Element	$(\exists a : a \in D : b \in D \Rightarrow a \uparrow b = a)$

Let $\sigma_i.x$ denote the value of a variable x with domain D in state $\sigma_i \in S_i$. Then our algorithm calculates

$$(\uparrow \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x \oplus \sigma_2.x)$$

This generalization is very useful as shown by the following examples.

Example 1 $(D, \uparrow, \oplus) := (Integers, \max, +)$.
The resulting calculation is $(max\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x + \sigma_2.x)$. This is the construction used in the presentation of the algorithm.

Example 2 $(D, \uparrow, \oplus) :=$ (Reals which are greater than or equal to $1.0, max, *$).
The resulting calculation is $(max\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x * \sigma_2.x)$ which is the maximum value of $\sigma_1.x * \sigma_2.x$ in any global state.

Example 3 $(D, \uparrow, \oplus) := (\{T, F\}, \vee, \wedge)$.
The resulting calculation is $(\vee\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x \;\wedge\; \sigma_2.x)$. This is equivalent to weak conjunction [GW94] and "possibly $\sigma_1.x \wedge \sigma_2.x$" [CM91].

Example 4 $(D, \uparrow, \oplus) := (\{T, F\}, \wedge, \vee)$.
The resulting calculation is $(\wedge\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x \;\vee\; \sigma_2.x)$. This is the dual of example 3 and could be called strong disjunction: in every cut either $\sigma_1.x$ or $\sigma_2.x$ is true.

Example 5 $(D, \uparrow, \oplus) := (\{T, F\}, \wedge, \wedge)$.
The resulting calculation is $(\wedge\, \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2 : \sigma_1.x \;\wedge\; \sigma_2.x)$ which states that both $\sigma_1.x$ and $\sigma_2.x$ are invariant.

5 N Boolean Variables

In this section we consider the case when there are N Boolean variables at N processes and we wish to determine if there exists a global state in which at least K of these N variables are true. Let x_i denote the boolean variable at process i. If we remove all states σ from S_i such that $\sigma.x_i$ is false, then we have the reduced set of states at process i:

$$R_i = \{\sigma | \sigma \in x_i \;\wedge\; \sigma.x_i\}$$

We define $(R, \rightarrow)$ to be the poset which results from taking the union of all R_i.

The problem of finding a global state in S such that at least K of the variables x_i are true is equivalent to finding a cut, or antichain, in $(R, \rightarrow)$ of size at least K. Thus the (revised) problem statement for this section is to determine if the following holds for a given execution:

$$(\exists c : c \in \mathrm{C} : |c| \geq K) \qquad \text{(PS2)}$$

where C refers to the set of all cuts in the reduced poset $(R, \rightarrow)$.

The techniques used to solve PS2 are based on Dilworth's theorem [Dil50] which shows that the maximum size antichain in a poset equals the minimum number of chains which cover the poset. It is known that if the poset is given in the form of an acyclic graph, then max-flow techniques [Law76] can be used to compute the width [Atk87]. However, to the best of our knowledge, the number of comparisons required to compute the width is not known if the poset is given in a chain-decomposed form. Two special cases of this problem ($k = 2$ and $k = n$) are solved in [Gar92].

This is one of the fundamental problems in deposets. It has many applications in distributed debugging. For example, we use the solution of the above problem to detect the unstable predicate of the form

$$x_1 + x_2 + ... + x_N \geq K$$

where $x_i \in \{0, 1\}$ is a variable in the process P_i for all i. As an example, consider the K-mutual exclusion problem. This problem requires a synchronization protocol such that no more than K processes are ever in the critical section. This is useful if the number of resources (for example, the number of copies for a copyrighted program) in a network is constrained to be at most K. If x_i represents the fact that process P_i is in the critical section, then detection of the global predicate $x_1 + x_2 + ... + x_N \geq K + 1$ is equivalent to detection of violation of K-mutual exclusion.

5.1 Algorithm

We are given R already partitioned into N chains, R_i, and we need to determine if there exists an antichain of size at least K. It follows from Dilworth's theorem that R can be partitioned into $K-1$ chains if and only if there does not exist an antichain of size at least K. Therefore, we can solve PS2 by trying to partition R into $K-1$ chains. If we succeed then PS2 is false. If we fail then PS2 is true.

So our problem is now reduced to taking the N chains R_i, $1 \leq i \leq N$, and trying to merge them into $K-1$ chains. The approach we take is to choose K chains and try to merge them into $K-1$ chains. After this step we have $N-1$ chains left. We do this step ("choose K chains, merge into $K-1$ chains") $N-K+1$ times. If we fail on any iteration, then R could not be partitioned into $K-1$ chains. Thus there exists an antichain c such that $|c| > K$. If we succeed then we have reduced R to $K-1$ chains and there does not exist an antichain c such that $|c| > K$.

The algorithm uses queues to represent chains. Each queue is stored in increasing order so the head of a queue is the smallest element in the queue. An element represents a state, and a smaller state "happens before" a larger state (this can be determined by comparing the vector clocks of the states). The algorithm uses the following operations on queues (q represents a queue):

insert(q,e)	insert element e in q
deletehead(q)	remove the head of q
empty(q)	true if q is empty
head(q)	return the first item in q, or a maximal value if q is empty

The algorithm $FindAntiChain$ (figure 6) calls the $Merge$ function $N-K+1$ times. The $Merge$ function takes K queues as input and returns K queues: $Q_1, \ldots, Q_K$. If Q_k is returned empty, then the merge was successful (the result is in $Q_1, \ldots, Q_{K-1}$) and $Merge$ continues to the next iteration. If Q_k is not empty then an antichain has been found and is given by the heads of the returned queues.

There are two important decisions in this algorithm. The first is how to choose the K chains for the merge operation. The answer follows from classical merge techniques used for sorting. We choose the chains which have been merged the fewest number of times. This is why the algorithm rotates $Qlist$ after each merge operation. This reduces the number of comparisons required by the algorithm.

The second and more important decision is how to implement *Merge*. The merge is performed by repeatedly removing an element from one of the K input chains and inserting it in one of the $K-1$ output chains. Output chain Q_K is only used if $Merge$ fails, thus when we say "output chain" we are only referring the the chains which contain successfully merged output: $Q_1, \ldots, Q_{K-1}$. We move the smallest elements first (that is, the ones which represent states that occurred earlier in the computation). An element will be moved from the input to the output if it is smaller than an element on the head of some other chain. The problem is deciding on which output chain to place the element.

Note that this is trivial for $K = 2$, when two input queues are merged into a single output queue as is done in the merge sort. For $K > 2$, the number of output queues is greater than one, and the algorithm needs to decide which output queue the element needs to be inserted in. The simple strategy of inserting the element in any output queue does not work as shown in figure 8 where there are three input queues (P_1, P_2, P_3) which need to be merged into two output queues

```
function FindAntiChain (K:integer, Qlist : list of queues of vector clocks) : antichain; begin
    assume: 1 ≤ K ≤|Qlist|
    assume: q ∈ Qlist ⇒ ¬empty(q)
    for (n :=|Qlist|; n ≥ K; n := n − 1) do
        (p_1, .., p_N) := Qlist;
        (q_1, . . . , q_K) := Merge(p_1, . . . , p_K);
        if (empty(q_k))
            then Qlist := (p_{K+1}, .., p_n, q_1, .., q_{k−1});                rotate list
            else return ( {head(q_i) | 1 ≤ i ≤ K} );
        end_if
    end_for
    return(∅);
end_function
```

Figure 6: Function that determines if an antichain of size K exists in the poset encoded by the queues of vectors listed in $Qlist$

(Q_1, Q_2). Suppose we use a simple strategy which results in the operations listed below. Initially Q_1 and Q_2 are empty. Each operation moves an element from the head of P_1, P_2 or P_3 to one of the two output queues. Note that we can only move the head of P_i if it is smaller than the the head of some other queue P_j. The operations we perform are:

1. $(1,0,0) < (2,0,0)$. So move $(1,0,0)$ to some output queue, say Q_1.
2. $(0,1,0) < (1,1,0)$. So move $(0,1,0)$ to some output queue, say Q_2.
3. $(1,1,0) < (2,2,0)$. So move $(1,1,0)$ to some output queue, say Q_1.
4. $(1,2,0) < (2,2,0)$. So move $(1,2,0)$ to some output queue, say Q_1.
5. $(2,0,0) < (2,2,0)$. So move $(1,2,0)$ to some output queue, but which one?

Notice that we have worked ourselves into a corner because when we decide to move $(2,0,0)$ there is no output queue in which we can insert it. The output queues must be sorted since they represent chains. This is done by inserting the elements in increasing order, but $(2,0,0)$ is not larger than the tails of any of the output queues. Thus we have nowhere to insert it.

This situation does not imply that the input queues cannot be merged. In fact in this case they can be merged into two queues as shown on the right side of figure 8. It merely implies that we did not intelligently insert the elements in the output queues. The function $FindQ$ chooses the output queue without running into this problem. Discussion of $FindQ$ is deferred until after we describe the details of the $Merge$ function.

The $Merge$ function compares the heads of each input queue with the heads of all other queues. Whenever it finds a queue whose head is less than the head of another queue, it marks the smaller of the two to be deleted from its input queue and inserted in one of the output queues. It repeats this process until no elements can be deleted from an input queue. This occurs when the heads of

all the queues are incomparable, that is, they form an antichain. Note that it may be the case that some input queues are empty. If none are empty, then we have found an antichain of size K. The heads of the input queues form the antichain. If one or more are empty, the merge operation (which is not complete yet) will be successful. All that is left to do is to take the non-empty input queues and append them to the appropriate output queues. This is done by the *FinishMerge* function whose implementation is not described because it is straight forward.

The *Merge* algorithm is shown in figure 7. Note that it only compares the heads of queues which have not been compared earlier. It keeps track of this in the variable *ac*, which is a set of indices indicating those input queues whose heads are known to form an antichain. Initially *ac* is empty. The *Merge* algorithm terminates when either *ac* has K elements or one of the input queues is empty.

The first *for* loop in *Merge* compares the heads of all queues which are not already known to form an antichain. That is, we compare each queue not in *ac* to every other queue. This avoids comparing two queues which are already in *ac*. Suppose $e_i = head(P_i)$ and $e_j = head(P_j)$ and inside the first *for* loop it is determined that $e_i < e_j$. This implies that e_i is less than all elements in P_j. Thus, e_i cannot be in an antichain with any element in P_j and therefore cannot be in any antichain of size K which is a subset of the union of the input queues. Thus, we can safely move e_i to an output queue, which eliminates it from further consideration. The set *moved* records which elements will be moved from an input queue to an output queue. The array *bigger* records the larger of the two elements which were compared. In this example, $bigger[i]$ equals j, implying that that the head of P_j is bigger than the head of P_i. This information is used by *FindQ* to choose the output queue where the head of P_i will be inserted.

The second *for* loop just moves all elements in *move* to an output queue. Consider the state of the program just before the second *for* loop begins. If the head, e, of an input queue is not marked to be moved, then e is not less than the head of any other input queue or else it would have been marked to be moved. This implies that any two elements which are not moved are concurrent, which in turn implies that set of heads which are are not moved form an antichain. This antichain is recorded in *ac* for the next iteration of the *while* loop.

We now return to describing how *FindQ* works. The formal description of *FindQ* is shown in figure 10. Given the queue which contains the element to be moved, and the queue with which this element was compared (it must be smaller than the head of another queue in order to move it), the procedure *FindQ* determines which output queue to use. The *FindQ* function takes three parameters:

G	:	an undirected graph called *queue insert graph*
i	:	the input queue from which the element x is to be deleted
j	:	the queue in which all elements are bigger than x

A "queue insert graph" is used to deduce the queue in which the next element is inserted. It has K vertices and exactly $K - 1$ edges. An edge corresponds to an output queue and a vertex corresponds to an input queue. Therefore, each edge (i, j) has a label, $label(i, j) \in \{1, \ldots, K - 1\}$, which identifies the output queue it corresponds with. No labels are duplicated in the graph, thus each output queue is represented exactly once. Similarly, each input queue is represented exactly once.

An edge (i, j) between vertex i and vertex j means that the heads of P_i and P_j are both bigger than the tail of $Q_{label(i,j)}$. The goal is to ensure that for any input queue (i.e. any vertex) there

```
function Merge(P_1, ..., P_K:queues) : Q_1, ..., Q_K: queues;
const all = {1,...K};
var ac,move: subsets of all;
    bigger: array[1..k] of 1..k;
    G: initially any acyclic graph on k-1 vertices;
begin
    ac := ∅;
    while (|ac|≠ K ∨ ¬(∃i : 1 ≤ i ≤ K : empty(P_i))) do
        move := {};
        for i ∈ all - ac and j ∈ all do
            if head(P_i) < head(P_j) then
                move := move ∪ {i};
                bigger[i] := j;
            end_if;
            if head(P_j) < head(P_i) then
                move := move ∪ {j};
                bigger[j] := i;
            end_if;
        end_for
        for i in move do
            dest := FindQ(G,i,bigger[i]);
            x := deletehead(P_i)
            insert(Q_dest, x);
        end_for
        ac := all - move;
    end_while
    if (∃i :: empty(P_i)) then
        FinishMerge(G, P_1, ... P_K, Q_1, ... Q_{K-1});
R1:     return (Q_1, ..., Q_{K-1}, ∅);
    else
R2:     return (P_1, ..., P_K);
    end_if
end_function
```

Figure 7: Generalized Merge Procedure for deposets

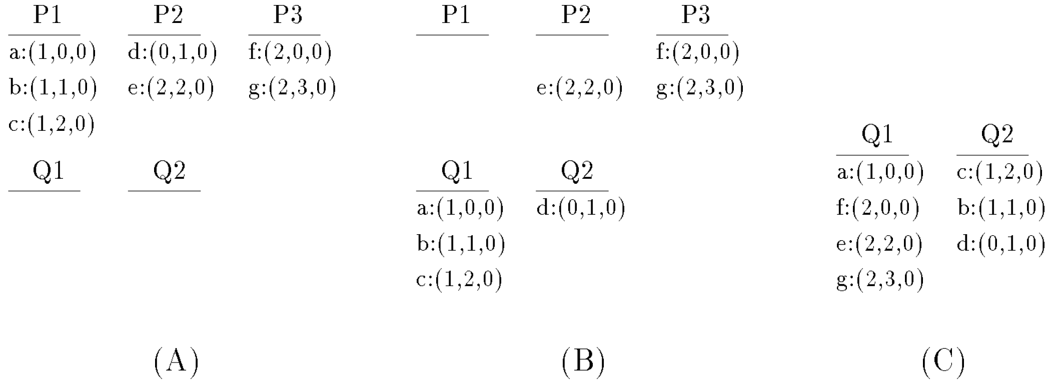

Figure 8: An example of a failed naive strategy. Diagram A shows the initial configuration. Diagram B shows the point at which the strategy fails: there is no where to insert $(2,0,0)$. Diagram C shows that this example can be merged into two chains.

exists an output queue (i.e. an edge) in which the head of input queue can be inserted. This constraint is equivalent to the requirement that every vertex has at least one edge adjacent to it. It is also equivalent to the requirement that the graph is a tree (i.e., acyclic) since there are K nodes and $K-1$ edges.

$FindQ$ uses the queue insert graph as follows. Consider the function call $FindQ(G,i,j)$. The element to be deleted is $e_i = head(P_i)$, and it is smaller than $e_j = head(P_j)$ (i.e., $bigger[i] = j$). $FindQ$ adds the edge (i,j) to G. Since G was a tree before adding edge (i,j), it now contains exactly one cycle which includes (i,j). Let (i,k) be the other edge incident on vertex i which is part of this cycle (it is possible that $k=j$). $FindQ$ deletes (i,k) and adds (i,j), and then sets the label of (i,j) equal to the label of (i,k). $FindQ$ then returns the label associated with the new edge. This label indicates which queue e_i will be inserted in.

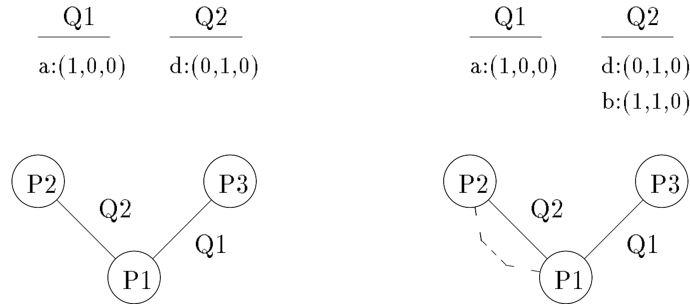

Figure 9: Using a queue insert graph to find the output queue

```
function FindQ(G: graph; i,j:1..K) : label;
    add edge (i, j) to G;
    (i, k) := the edge such that (i, j) and (i, k) are part of the same cycle in G;
    remove edge (i, k) from G;
    label(i, j) := label(i, k);
    return label(i, j);
end_function
```

Figure 10: Function that finds which output queue to insert an element which is being deleted from the head of input queue P_j. The chosen output queue is $Q_{label(i,j)}$.

Consider our previous example with the naive algorithm. It made a bad decision when it placed element $(1, 1, 0)$ on Q_1. Figure 9 shows the state of the queues and of the graph before and after the correct decision made by $FindQ$. Element $(1, 1, 0)$ is in P_1 and is less than the head of P_2. Thus the edge $(1, 2)$ is added to the graph (dashed line in figure 9). It forms a cycle with the other edge $(1, 2)$ which is labeled with Q_2. We copy this label to the new edge, delete the old edge and return Q_2, indicating that $(1, 1, 0)$ should be inserted in Q_2.

An important invariant of the queue insert graph is that given any edge (i, k) and the output queue associated with it, the queue is empty or the tail of the queue is less than all elements in input queues P_i and P_k. This is stated and proven in lemma 5 and is used later to show that the output queues are always sorted.

Lemma 5 $(i, k) \in G \Rightarrow (\quad empty(Q_{label(i,k)}) \ \vee \ (\forall e : e \in P_i \cup P_k : tail(Q_{label(i,k)}) \leq e) \quad)$

Proof: Initially, the lemma holds since all output queues are empty. Now assume that the lemma holds and we need to insert $e_i = head(P_i)$ into output queue Q_l where $l = FindQ(G, i, j)$ and $j = bigger[i]$. Since $j = bigger[i]$, we know $e_i \leq head(P_j)$. Since P_i and P_j are sorted, we know $e_i \leq e$ for any element $e \in P_i \cup P_k$. After moving e_i to Q_l, the tail of Q_l will be e_i and the lemma will still hold. ■

The merge operation must produce sorted output queues. Lemma 6 proves that our algorithm meets this requirement.

Lemma 6 *If the elements are inserted in the output queues using* FindQ*, then all output queues are always sorted.*

Proof: Initially all output queues are empty. Now assume each output queue is sorted and we need to move $e_i = head(P_i)$ to an output queue. The $FindQ$ procedure will return $Q_{label(i,j)}$, where (i, j) is some edge in G. By lemma 5, the tail of this queue is less than or equal to e_i. Thus after inserting e_i in Q_l, Q_l is still sorted. No other output queues are modified, thus the lemma still holds. ■

Lemma 7 *Suppose* $(Q_1, \ldots Q_K) = Merge(P_1, \ldots, P_K)$ *and each queue* P_i *is sorted. Then*

(A) $empty(Q_K) \Rightarrow$ $P_1, \ldots P_K$ *have been merged into* $K - 1$ *sorted queues* $Q_1, \ldots Q_{K-1}$.

(B) $\neg empty(Q_K) \Rightarrow \{head(P_i) \mid 1 \leq i \leq K\}$ *is an antichain of size* K.

(C) $\neg empty(Q_K) \iff$ *there exists an antichain of size* K.

Proof: We use the following invariant of the *while* loop:

$$i \in ac \ \wedge \ j \in ac \ \wedge \ \neg empty(P_i) \ \wedge \ \neg empty(P_j) \Rightarrow head(P_i) \| head(P_j).$$

(A): Suppose $empty(Q_K)$. Then *Merge* returned via statement $R1$ and just before returning there existed i such that $empty(P_i)$. Let P_K be the empty one. Then all elements reside in one of $P_1, \ldots P_{K-1}$ or $Q_1, \ldots Q_{K-1}$. Thus after *FinishMerge* all elements are in $Q_1, \ldots Q_{K-1}$ which by lemma 6 are sorted. Thus the original input queues have been merged into $K-1$ output queues.

(B): Suppose $\neg empty(Q_K)$. Then *Merge* returned via $R2$ and just before returning there was no i such that $empty(P_i)$. Then by the invariant and since $ac = all$ we know $head(P_i)$ is concurrent with the head of every other P_j. Thus $\{head(P_i) \mid 1 \leq i \leq K\}$ is an antichain of size K.

(C): Follows from (A), (B), and Dilworth's theorem. ∎

Theorem 3 *There exists an antichain of size K in $\cup_{1 \leq i \leq K} P_i$ if and only if* FindAntiChain$((P_1, \ldots P_K))$ *returns an antichain of size K.*

Proof: Assume there exists an antichain which satisfies the left hand side. Then by Dilworth's result, $\cup_{1 \leq i \leq K} P_i$ cannot be merged into $K-1$ chains. Thus by lemma 7, *Merge* will return an antichain of size K.

Now assume there does not exist such an antichain. Then $\cup_{1 \leq i \leq K} P_i$ can be merged in $K-1$ chains and *Merge* will return with P_K empty on iteration $N-K+1$ of the *while* loop, which will cause *FindAntiChain* to return the empty set. ∎

5.2 Complexity

In this section, we analyze the complexity based on the number of comparisons required by the algorithm (i.e., the number of times the heads of two queues are compared in the *Merge* function. We prove an upper bound and a lower bound. The lower bound is proven by defining an adversary which produces a set of input queues that forces the merge algorithm to use at least KMN comparisons, where M is the number of elements in the largest input queue.

5.2.1 An Upper Bound

Theorem 4 *The maximum number of comparisons required by the above algorithm is $KMN(K + log_\rho N)$.*

Proof: We first calculate the complexity of merging K queues into $K-1$ queues. From the *Merge* algorithm it is clear that each element must be in *ac* before it passes to an output queue and it requires K comparisons to be admitted to *ac*. Thus, if the total number of elements to be merged is l, then l elements must pass through *ac* on their way to the output queue for a total of Kl comparisons.

Initially $l \leq KM$ but the queues grow for each successive call to *Merge*. At this point, our technique of rotating the list of queues to be merged is useful. Let $level.i$ denote the maximum number of merge operations that any element in P_i has participated in. The algorithm rotates *Qlist* to ensure that in each iteration the K queues with the smallest level numbers will be merged. Initially, there are N queues at level 0. Each of the first N/K merge operations reduces K queues with level 0 to $K-1$ queues with level 1. This pattern continues until there are $2K$ queues left, at

which time the maximum level will be $log_\rho N$ where ρ is the reducing factor and equals $K/(K-1)$. Merging the remaining $2K$ queues into $K-1$ adds K more levels. Thus the maximum level of any final output queue is $K + log_\rho N$. Thus, there are at most MN elements, each of which participates in at most $K + log_\rho N$ merge operations at a cost of K comparisons per element per merge. Therefore the maximum number of comparisons required by the above algorithm is $KMN(K + log_\rho N)$. ■

Note that for the special case when $K = 2$, the complexity is $O(mn \log N)$. This is shown to be optimal in [Gar92]. Further, if $M = 1$ and $K = 2$, this reduces to the well-known merge sort algorithm with the complexity of $O(N \log N)$ comparisons. Another noteworthy special case is when $K = N$. In this case, the complexity becomes $O(MN^2)$ which is also known to be optimal [Gar92].

5.2.2 A Lower Bound

In this section, we provide a lower bound on the number of comparisons required by any algorithm to solve the above problem.

Proposition 1 *Let $(P, <)$ be any partially ordered finite set of size MN. We are given a decomposition of P into N sets $P_1, ...P_N$ such that P_i is a chain of size M. Any algorithm which determines if there exists an anti-chain of size K must make at least $\Omega(KMN)$ comparisons.*

Proof: We use an adversary argument. Let $P_i[s]$ denote the sth element in the queue P_i. The adversary will give the algorithm P_i's with the following characteristic:

$$(\forall i, j, s :: P_i[s] < P_j[s+1])$$

Formally, on being asked to compare $P_i[s]$ and $P_j[t]$, $(s \neq t)$ the adversary uses:

if $(s < t)$ **then** return $P_i[s] < P_j[t]$
if $(t < s)$ **then** return $P_j[t] < P_i[s]$

Thus, the above problem reduces to M independent instances of the problem which checks if a poset of N elements has a subset of size K containing pairwise incomparable elements. If the algorithm does not completely solve one instance then the adversary chooses that instance to show a poset consistent with all its answers but different in the final outcome.

We now show that the number of comparisons to determine whether any poset of size N has an anti-chain of size K is at least $N(K-1)/2$. The adversary will give that poset to the algorithm which has either $K-1$ or K chains such that any pair of elements in different chains are incomparable. In other words, the poset is a simple union of either $K-1$ or K chains. The adversary keeps a table $numq$ such that $numq[x]$ denotes the number of questions asked about the element x. The algorithm for the adversary is shown in Fig. 11.

If the algorithm does not ask $K-1$ questions about any element x, the adversary can produce a poset inconsistent with the answer of the algorithm. If the algorithm answered that no anti-chain of size K exists then the adversary can produce an anti-chain which includes one element from each of the $K-1$ chains and the element x. On the other hand, if the algorithm answered that an anti-chain exists, then the adversary could put x and all other elements for which $K-1$ questions have not been asked in $K-1$ chains.

```
var numq[N]:integer initially 0;          number of questions asked about element x
function  compare (x,y:elements)
    numq[x]++; numq[y]++;
    if (numq[x] = K − 1) then
        chain[x] := chain in which no element has been compared with x so far;
    end_if
    if (numq[y] = K − 1) then
        chain[y] := chain in which no element has been compared with y so far;
    end_if
    if (numq[x] < K − 1) or (numq[y] < K − 1) then
        return x||y;
    end_if
    if (chain[x] ≠ chain[y]) then
        return x||y;
    else
        if x inserted earlier than y then return (x < y);
        else return (y < x);
        end_if
    end_if
end_function
```

Figure 11: Algorithm for the Adversary

Since each comparison involves two elements, we get that the algorithm must ask at least $N(K-1)/2$ questions for each level. Thus, overall any algorithm must make at least $MN(K-1)/2$ comparisons. ■

It is easy to see that the lower bound is not tight. If we choose $M = 1$ and $K = 2$, we get the lower bound of $N/2$. However, the lower bound of $N \, log \, N$ is well known for this case.

6 Conclusion

In this paper we developed algorithms for monitoring the value of a global function of the form $x_1 + x_2 + \ldots x_N$ where x_i is a variable local to process i. We monitor an execution of a distributed program to determine if the value of the function exceeds some constant K. More precisely, we determine if there exists a global state (i.e., a set of local states which are mutually concurrent) such that $\sum_{1 \leq i \leq N} x_i > K$. The ability to monitor functions like this is useful for debugging, testing and analyzing distributed programs.

We consider two special cases of this problem. The first case imposes the following restrictions: each x_i is an integer variable, and $N = 2$. Notice that there can be more than two processes, but only two can contribute local variables to the global function. The second case restricts x_i to take on values from the set $\{0, 1\}$, and does not restrict N.

For the case of two integer variables, we presented a centralized and a decentralized algorithm, proved its correctness and analyzed its complexity. The algorithm requires four integers to be piggybacked onto messages generated by the underlying program (i.e., application messages). In addition, the monitoring algorithm generates one debug message per application message received at one of the two processes contributing variables to the global function. The computational complexity for the centralized algorithm is $\Theta(A \, lg \, A)$ where A is the number of messages sent plus the number received at the two contributing processes.

For the case of N Boolean variables, we present and prove an algorithm which has an upper bound complexity of $KMN \lceil log((N-K)/(K-1)) \rceil$ where M is the number of times x_i changed value in the computation. We also presented a lower bound of $\Omega(KMN)$.

There are several open problems relating to this research. A general problem is: "What classes of global functions can be efficiently monitored on distributed programs?" In this paper we addressed special cases of one class of global functions. Another more specific problem is: "Can we efficiently detect more general cases than those considered in this paper?" For example, can these results be extended to consider N integer variables at N processes. Another area of future research is determining how the properties of global functions affect the ability to monitor it. For example, for in this paper we identified the property "addition distributes over *max*" as being important to enable efficient monitoring of the global function. In this paper we have only touched on this line of reasoning, but we believe it merits further study.

7 Acknowledgments

We are grateful to J. Roger Mitchell, Ken Marzullo, Michel Raynal and Jong-Deok Choi for helpful comments on earlier versions of this paper.

References

[Atk87] M. D. Atkinson. The complexity of orders. In I. Rival, editor, *Algorithms and Orders*, volume 255 of *NATO ASI Series, Mathematical and Physical Sciences*, pages 195–230. 1987.

[BM93] Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems: fundamental concepts and mechanisms, in Distributed Systems*, chapter 4. ACM Press, Frontier Series. (S.J. Mullender Ed.), 1993.

[Bou87] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[BR94] Ö. Babaoğlu and M. Raynal. Specification and detection of behavioral patterns in distributed computations. In *Proc. of 4th IFIP WG 10.4 Int. Conference on Dependable Computing for Critical Applications*, San Diego, CA, January 1994. Springer Verlag Series in Dependable Computing.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–166, 1950.

[DJR93] C. Diehl, C. Jard, and J. X. Rampon. Reachability analysis on distributed executions. In *Theory and Practice of Software Development*, pages 629–643. TAPSOFT, Springer Verlag, LNCS 668 (Gaudel and Jouannaud editors), April 1993.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[FRGT94] E. Fromentin, M. Raynal, V.K. Garg, and A.I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc of the* 23rd *International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[Gar92] V.K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[GCKM94] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting conjunctive channel predicates in a distribute programming environment. Technical Report TR-PDS-94-02, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994.

[GTFR94] V.K. Garg, A.I. Tomlinson, E. Fromentin, and M. Raynal. An efficient decentralized algorithm for detecting properties of distributed computations. Technical Report TR-PDS-94-04, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994.

[GW92] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[GW94] V.K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).

[JJJR94] C. Jard, T. Jeron, G.V. Jourdan, and J.X. Rampon. A general approach to trace-checking in distributed computing systems. In *Proc. of the International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[MC88] B.P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, July 1988. IEEE.

[Ray92] M. Raynal. About logical clocks for distributed systems. *ACM Operating Systems Review*, 26(1):41–48, 1992.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[TG93] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 1993. ACM/ONR.