Detecting Conjunctions of Global Predicates

Vijay K. Garg* J. Roger Mitchell†

Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712
http://maple.ece.utexas.edu

Abstract

We present an efficient algorithm to detect if the conjunction of two nonlocal predicates is possibly true in a distributed computation. For offline detection of such global predicates, our algorithm is significantly more efficient than the previous algorithms by Cooper and Marzullo, and by Stoller and Schneider.

Keywords: Distributed systems, Predicate detection

1 Introduction

The detection of global conditions is a fundamental problem in an asynchronous distributed system. In an asynchronous distributed system a process cannot know the state of other processes at any given time due to communication delays. This creates difficulty in detecting conditions, or predicates, spread across the system. Deadlock and termination are two such global predicates. Detection of these predicates is useful for distributed debugging, monitoring distributed systems for faults, and other areas of distributed computing.

There are two interpretations of truthness of a global predicate in a distributed computation. A global predicate denoted by Φ is *possibly* true [2], (or true in the weak sense [3]) if there exists a consistent global state in the computation in which the global predicate is true. A global predicate is *definitely* true [2], (or true in the strong sense [3]) if all sequential observations of the computation go through a consistent global state in which the global predicate is true. In this paper, we will restrict ourselves to the first interpretation.

Cooper and Marzullo [2] discuss methods of detecting any global predicate. However their method involves searching the entire lattice of possible global states, a search which is $O(m^n)$, where m is the number of states in a process, and n is the number of processes. By restricting the class of predicates detected, algorithms have been proposed which are of polynomial time. For example, the class of conjunctive predicates can be detected in $O(n^2m)$ time using the algorithms proposed by Garg and Waldecker [3].

Stoller and Schneider [5] combine the approach of Garg and Waldecker with that of Cooper and Marzullo. Their algorithm, like the algorithm of Cooper and Marzullo, can detect any

*supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

†supported in part by a Virginia & Ernest Cockrell fellowship

general global predicate Φ; and like the algorithm of Garg and Waldecker, can exploit the structure of the predicate Φ to reduce the computational complexity of detection. They assume that the global predicate is a conjunction of predicates of the form

$$\Phi(x_1, ..x_j), \tag{1}$$

where $\Phi()$ is a predicate with variables, x_i, from different processes. An example of a predicate in this form is

$$(x_1 = x_2) \wedge (x_3 > x_4) \wedge (x_5 < 10), \tag{2}$$

where $x_i's$ are variables on different processes. When the global predicate has only one conjunct, their approach reduces to that of Cooper and Marzullo; and when each of the conjuncts is local to a single process, their approach reduces to that of Garg and Waldecker. They introduce the notion of a *fixed set* of a global predicate. A fixed set of a global predicate is a set of variables such that by fixing these variables in the predicate, the predicate reduces to a conjunction of local predicates. The sets $\{x_1, x_3\}$ and $\{x_2, x_3\}$ are some examples of fixed sets for the global predicate in equation 2. The algorithm proposed by Stoller and Schneider is exponential in the size of the fixed set.

In this paper we present a significantly faster algorithm for a restricted class of predicates. Specifically, Stoller and Schneider's algorithm requires $O(m^{j+k-1})$ time for a predicate of the form

$$\Phi_1(x_1, ..., x_j) \wedge \Phi_2(y_1, ..., y_k) \tag{3}$$

where m is the number of states at one process. In equation (3) we assume that all x_i's and y_i's are on different processes. We present an offline algorithm which requires $O(m^l)$ time, where $l = max(j, k)$.

We note here that the algorithm of Stoller and Scneider can be used either off-line or on-line with the same time complexity. However, the algorithm proposed in this paper can be run on-line only by increasing its time complexity. A trivial way in which the proposed algorithm can be used on-line is by running it from scratch every time a new local state is received. This increases the time complexity by a factor of m.

We use many novel ideas in the algorithm and therefore make the following contributions:

- We formalize a property of predicates called *monotonicity* [6] which is crucial for efficient evaluation of global predicates in real-life applications. This property allows detection algorithms to consider at most as many states as the number of messages sent and received by the process as opposed to the total number of states from a process.

- We introduce a technique which transforms the problem of predicate detection into a computational geometric problem.

- We give an efficient algorithm for the computational geometric problem. Our algorithm for this problem may also be of independent interest.

This paper is organized as follows. Section 2 presents our model of distributed computation and the notation that is used in the paper. Section 3 describes our algorithm and its correctness. Section 4 presents some extensions and optimizations of the basic algorithm.

2 Model

We assume a loosely-coupled message-passing system without shared memory or a global clock. A distributed program consists of a set of n processes denoted by $\{P_1, P_2, ..., P_n\}$ communicating solely via asynchronous messages. In this paper, we will be concerned with a single run r of a distributed program. Each process P_i in that run generates a single execution trace $r[i]$ which is a finite sequence of *states*. The state of a process is defined by the values of all its variables including its program counter. The state transition occurs in any process due to an internal action, the send of a message, or the receive of a message.

We define the usual *causally precedes* relation ($\rightarrow$) on states as follows. A state $s \in r[i]$ causally precedes ($\rightarrow$) a state $t \in r[j]$ if and only if one of the following conditions holds.

1. $i = j$ and s occurs before t in $r[i]$. We say that $s \prec t$ (s locally precedes t) when this is true.

2. The action following s is the send of a message and the action preceding t is the receive of that message.

3. There is a state u such that $s \rightarrow u$ and $u \rightarrow t$.

We use $s||t$ to denote that states s and t are concurrent. That is, $s||t \equiv s \not\rightarrow t \wedge t \not\rightarrow s$. We use g and h to represent global states. A global state g is a set of states $\cup_i g[i]$ for which $g[i] \in r[i]$ and $\forall i, j : i \neq j : g[i]||g[j]$. The global state g is also called a consistent cut. We also use the notion of sub-cuts – a sub-cut is a cut across a subset of processes in the system. We use $g(\Phi)$ to represent a consistent sub-cut across a minimal set of processes in which the predicate Φ is true. This notation uniquely specifies the local states used to satisfy Φ. The notation $g(\Phi_1)||g(\Phi_2)$ means that every pair of states in the two sub-cuts satisfying Φ_1 and Φ_2 are concurrent; we say, then, that $g(\Phi_1)$ and $g(\Phi_2)$ are concurrent. A set of cuts is written as G.

2.1 State Interval

A state interval is a sequence of states between two *external* events where an external event is the sending or receiving of a message, the beginning of the process or the termination of the process. Formally, the k^{th} interval in P_i (denoted by (i, k)) is the subsequence of $r[i]$ between the $(k-1)^{th}$ and k^{th} external events. For a given interval (i, k), if k is out of range then (i, k) refers to $\perp$ which represents a sentinel value (or a "null" interval). The notion of intervals is useful because the relation of two states belonging to the same interval is a congruence with respect to $\rightarrow$. That is, the relation of two states being in the same interval is an equivalence relation and for any two states s, s' in the same interval and any state u from a different interval: $(s \rightarrow u \Leftrightarrow s' \rightarrow u)$ and $(u \rightarrow s \Leftrightarrow u \rightarrow s')$. We exploit this congruence in our algorithms by assigning a single timestamp to all states belonging to the same interval.

2.2 Predecessor and Successor functions

The predecessor and successor functions are defined as follows for any state u and $1 \leq i \leq n$:

$$pred.u.i = \max\{v \in r[i] \mid v \rightarrow u\}$$

$$succ.u.i = \min\{v \in r[i] \mid u \rightarrow v\}$$

Consider a state $s \in r[i]$. The predecessor of s in $r[j]$, denoted $pred.s.j$, is the latest state in $r[j]$ which causally precedes s. Due to the congruence for states belonging to the same interval, the *pred* and *succ* functions and the $||$ relation are well defined on intervals. The following lemma establishes some useful properties of *pred* and *succ*.

Lemma 1 *Let u and v be states in $r[i]$ and $r[j]$ respectively.*

1. $(i, w) = pred.(j, x).i \quad \Leftrightarrow \quad (\forall y : y > w : (i, y) \not\rightarrow (j, x)) \wedge (i, w) \rightarrow (j, x)$
2. $(i, w) = succ.(j, x).i \quad \Leftrightarrow \quad (\forall y : y < w : (j, x) \not\rightarrow (i, y)) \wedge (j, x) \rightarrow (i, w)$
3. $v \preceq pred.u.j \Leftrightarrow succ.v.i \preceq u.$
4. $succ.(pred.u.j).i \preceq u \preceq pred.(succ.u.j).i$

In addition to *pred* and *succ*, we also use *prev* and *next*. For any state u in $r[i]$, $prev.u$ returns the state preceding u in $r[i]$ if there exists one, else it returns $\perp$. The definition of $next.u$ is similar.

3 The algorithm

For simplicity, we first give an algorithm for predicates of the form

$$\Phi_1(x_1, x_2) \wedge \Phi_2(y_1, y_2).$$

This algorithm is of complexity $O(m^2)$. This is faster than the previously known algorithm by Stoller and Schneider [5] which requires $O(m^3)$ time. Later we briefly explain how to extend the algorithm to a more general case.

There are three steps in the algorithm – computing the sub-cuts that satisfy Φ_1 and Φ_2, transformation of the problem to a computational geometric problem, and finally, solution of the transformed problem. We discuss each of these steps next.

3.1 Computing the sub-cuts

In the first step, we compute the set of consistent sub-cuts in processes P_1 and P_2 that satisfy $\Phi_1(x_1, x_2)$. For example, if $\Phi_1(x_1, x_2) = (x_1 < x_2)$, then we find pairs of states $s \in P_1$ and $t \in P_2$ such that s and t are concurrent and the value of x_1 in s is less than or equal to x_2 in t. If there are at most m distinct states in each of P_1 and P_2, and evaluation of the predicate Φ_1 requires $O(1)$ time, then this step can be done in $O(m^2)$ time. This step can be viewed as an application of Cooper and Marzullo's algorithm for detection of Φ_1.

It is important to observe that in most applications, m as defined earlier is too large for this step to be practical. However, we now show that for most predicates Φ_1, m can be taken as the total number of state intervals. The total number of state intervals is much smaller than the total number of states.

The property of any global predicate Φ that allows this reduction is considered next.

Definition 2 *Assume that x_1 takes its value from a set totally ordered with respect to a relation $<$. We say that Φ is monotone with respect to x_1 if it satisfies the following equation:*

$$\forall a, x_2 : \Phi(a, x_2) \Rightarrow (\forall b : b < a : \Phi(b, x_2))$$

or,

$$\forall a, x_2 : \Phi(a, x_2) \Rightarrow (\forall b : a < b : \Phi(b, x_2)).$$

Informally, a predicate is monotone with respect to a variable x_1 if replacing the variable by a larger value (or by a smaller value) while keeping all other variables the same does not violate the truthness of the predicate. For example, consider the predicate $\Phi = x_1 < x_2$ where x_1 and x_2 are integers. Then Φ is monotone with respect to x_1, because if $\Phi(x_1, x_2)$ holds for a certain value of x_1, then it would continue to do so for any smaller value of x_1. Φ is also monotone with respect to x_2 because if $\Phi(x_1, x_2)$ holds for a certain value of x_2, then it would continue to do so for any larger value of x_2. An example of a predicate that is not monotone with respect to x_1 or x_2 is $(x_1 = x_2)$.

Monotonicity of a predicate allows us to restrict our attention to state intervals rather than states. For example, for the predicate $(x_1 < x_2)$, it is sufficient to keep the state with the smallest value of x_1 for each state interval. This reduction also exploits the fact that if two local states, s and s', on the same process are separated only by internal events, then they are indistinguishable to other processes as far as consistency is concerned.

We denote by $G(\Phi_1)$ and $G(\Phi_2)$ the set of all consistent sub-cuts which satisfy Φ_1 and Φ_2. In the worst case, $G(\Phi_1)$ and $G(\Phi_2)$ are of size $O(m^2)$. However, as mentioned above, for monotonic predicates the value of m is at most the number of messages sent or received by any process. From now on we will use state and state interval interchangeably with the understanding that for monotonic predicates, at most one state will be used from each state interval.

3.2 Transformation of the Predicate Detection Problem

We now describe the transformation of the predicate detection problem. Each element in $G(\Phi_1)$ can be viewed as a point in a 2-dimensional space. For example, if the third interval on process P_1 (denoted by $(1,3)$) and the fifth interval on P_2 (denoted by $(2,5)$) together satisfy Φ_1, then we view this consistent cut as a point $(3,5)$ in the plane generated by intervals in processes P_1 and P_2. For our example of the predicate $(x_1 < x_2)$ the point $(3,5)$ signifies that the state interval $(1,3)$ is concurrent to the state interval $(2,5)$ and the minimum value of x_1 in the interval $(1,3)$ is less than the maximum value of x_2 in the interval $(2,5)$.

Next, we consider the set of sub-cuts $G(\Phi_2)$ on P_3 and P_4. For each sub-cut $\{(3,u),(4,v)\}$ in $G(\Phi_2)$, we construct a rectangle in the plane defined by the intervals of P_1 and P_2. Each rectangle represents the set of all sub-cuts on P_1 and P_2 which are concurrent with $\{(3,u),(4,v)\}$. This is done by finding intervals s_l and s_h on P_1 and t_l and t_h on P_2 as follows.

$$s_l = next(max(pred.u.1, pred.v.1))$$

$$s_h = prev(min(succ.u.1, succ.v.1))$$

$$t_l = next(max(pred.u.2, pred.v.2))$$

$$t_h = prev(min(succ.u.2, succ.v.2))$$

The points (s_l, t_l) and (s_h, t_h) define a rectangle on the plane such that any point within represents a sub-cut on P_1 and P_2 which is concurrent to $\{(3,u),(4,v)\}$. The following lemma helps demonstrate this.

Lemma 3 *Any state s such that $s_l \preceq s \preceq s_h$ is concurrent with u and v. Conversely, any state s in P_1 that is concurrent with both u and v satisfies $s_l \preceq s \preceq s_h$.*

Proof: From the definition of s_l, it follows that $pred.u.1 \prec s_l$. Since $s_l \preceq s$ we get that $pred.u.1 \prec s$. From Lemma 1, it follows that $s \not\to u$. Similarly, $s \preceq s_h$ and $s_h \prec succ.u.1$, implies that $u \not\to s$. Therefore, $s \| u$. By repeating the argument for v, we get $s \| v$.

Conversely, let s be concurrent with u and v. Since $s \not\to u$ and $s \not\to v$, it follows that $max(pred.u.1, pred.v.1) \prec s$. Thus, $s_l \preceq s$. The argument for the upper bound is similar. $\square$

Let $R(\Phi_2)$ be the set of rectangles constructed from $G(\Phi_2)$. The following result shows that the problem of checking whether any sub-cut in $G(\Phi_1)$ is concurrent with any sub-cut in $G(\Phi_2)$ is equivalent to checking whether any point corresponding to a sub-cut in $G(\Phi_1)$ lies in any of the rectangles in $R(\Phi_2)$.

Theorem 4 *For any global predicate* $\Phi_1(x_1, x_2) \wedge \Phi_2(y_1, y_2)$,
$\exists (s,t) \in G(\Phi_1), (u,v) \in G(\Phi_2) : (s,t) || (u,v)$ *iff* $\exists (s,t) \in G(\Phi_1)$ *which lies in one of the rectangles in* $R(\Phi_2)$.

Proof: Using Lemma 3. $\square$

We now turn our attention to the complexity of this transformation. Since there are $O(m^2)$ sub-cuts in $G(\Phi_2)$, the operation of defining these rectangle needs to be done in $O(1)$ time per rectangle. We achieve this bound by use of vector clocks for predecessors as well as successors. The traditional vector clock implements the predecessor function. The notion of successor vector clocks was previously used without giving a detailed implementation in [1]. Basten used these for finding a global state prior to a sub-cut, but not for predicate detection. We now show a construction for successor vector clocks.

We describe an off-line version of a successor vector clock denoted by sv. Let the interval of the final state on process P_i be (i, f_i). Then, the final state of any process P_i is timestamped with a vector sv defined as $sv[i] = f_i$ and $sv[j] = \infty$ for all $j \neq i$. The rules to construct the sv vector of other state intervals are as follows. The intervals are timestamped with sv in a backward manner. Assume that the state s on P_i immediately precedes t which has already been timestamped. If the transition from s to t was a receive, then the following action is taken.

$$s.sv[j] = \begin{cases} t.sv[j] & \text{if } j \neq i \\ t.sv[j] - 1 & \text{otherwise} \end{cases}$$

If the transition was a send of a message which was received at state u, then the following action is taken

$$s.sv[j] = \begin{cases} min(t.sv[j], u.sv[j]) & \text{if } j \neq i \\ t.sv[j] - 1 & \text{otherwise} \end{cases}$$

The above rules are the simple dual of predecessor vector clocks.

Given predecessor and successor clocks, it is easy to construct a rectangle for each sub-cut in $G(\Phi_2)$. Observe that end points of rectangles are constructed using functions *pred*, *max*, *next*, *succ*, *min* and *prev* each of which can be computed in constant time.

3.3 Solution of the transformed problem

From Theorem 4, checking whether a sub-cut in $G(\Phi_1)$ is consistent with a sub-cut in $G(\Phi_2)$ is equivalent to checking whether there exists a rectangle in $R(\Phi_2)$ which includes a point corresponding to a sub-cut in $G(\Phi_1)$. Since there are $O(m^2)$ points and $O(m^2)$ rectangles, a simple brute force algorithm of checking each point for each rectangle will lead to an inefficient $O(m^4)$ algorithm. We now describe an $O(m^2)$ algorithm for this problem. The algorithm shown in Figure 1 detects whether any of up to m^2 points on an $m \times m$ plane lie within any of up to m^2 rectangles. For the case when the number of points and the number of rectangles is much less than m^2, a more efficient algorithm can be used. This will be given in a following section.

```
boolean is_pt_in_rectangle() {
        istrue[$m, m$]: 0..1; /* $istrue[s,t] = 1$ iff $(s,t) \in G(\Phi_1)$ */
        R: list of record /* rectangles specified by two corners. */
                {int $s_l, t_l, s_h, t_h$};

        int A[$m+1, m+1$]; /* number of points that lie between (0,0) and (s,t) */

        for $i = 0$ to $m$ do
                A[0,i]=0; A[i,0]=0;

        /* Array assignment: coding A with the points */
        for $i = 1$ to $m$ do
                for $j = 1$ to $m$ do
                        A[$i, j$] = A[$i-1, j$] + A[$i, j-1$] - A[$i-1, j-1$] + istrue[$i, j$];

        /* Rectangle/point overlap check */
        for $(s_l, t_l, s_h, t_h) \in R$ do
                /* exclusion inclusion principle */
                if $A[s_h, t_h] - A[s_l - 1, t_h] - A[s_h, t_l - 1] + A[s_l - 1, t_l - 1] > 0$ then
                        return true;
        return false;
}
```

Figure 1: Detecting if any point in an $m \times m$ plane is contained in any rectangle.

The algorithm *is_pt_in_rectangle* operates as follows. The arrays *istrue* and R store the coordinate values of the points and rectangles created in steps 2 and 3. The values stored by R specify two opposite corners as (s_l, t_l) and (s_h, t_h). These arrays are assigned before calling the algorithm, an operation of complexity $O(m^2)$. The algorithm uses another array A of size $m \times m$. The entry $A[s,t]$ is used to store the total number of points that are in the rectangle defined by $(0,0)$ and (s,t). The first column and row in A is first initialized to zeroes. The "array assignment" loop then assigns the correct value to other entries of A. The "Rectangle/point overlap check" checks for overlaps as follows. For all rectangles, the values in A at the four corners of the rectangle are used to determine if a point is contained within. Using set-theoretic inclusion exclusion principle it follows that the number of points in the rectangle (s_l, t_l, s_h, t_h) is given by

$$A[s_h, t_h] - A[s_l - 1, t_h] - A[s_h, t_l - 1] + A[s_l - 1, t_l - 1]$$

The algorithm can check m^2 points against m^2 rectangles with time complexity $O(m^2 + m^2)$ or $O(m^2)$. The algorithm uses $O(m^2)$ space; however, we later show how the space overhead can also be reduced to $O(m)$. The correctness of the *is_pt_in_rectangle* algorithm is proven next.

First we show that $A[s,t]$ equals the number of points which are contained in the rectangle defined by $(0,0)$ and (s,t).

Lemma 5 *After the array assignment of A,*
$A[s,t] = |B(s,t)|$ *where* $B(s,t) = \{(i,j) | istrue(i,j) \wedge 0 \le i \le s \wedge 0 \le j \le t\}$

Proof: We use induction on (s,t). If either of them is 0, then by initial assignment $A[s,t]$ is 0. Since $B(s,t)$ is empty for this case, the assertion holds. Now consider the case when both s

and t are strictly greater than 0. The program calculates $A[s,t]$ using:

$$A[s,t] = A[s-1,t] + A[s,t-1] - A[s-1,t-1] + istrue[s,t]$$

But using induction hypothesis, and the fact that the points in $B(s-1,t-1)$ are common in $B(s-1,t)$ and $B(s,t-1)$ we get that $A[s,t]$ equals cardinality of $B(s,t)$. □

Theorem 6 *The function* is_pt_in_rectangle *returns true iff there exists a point and a rectangle such that the point is contained in the rectangle.*

Proof: Using Lemma 5, $A[s,t]$ contains the number of points in the rectangle $(0,0,s,t)$. It follows that $A[s_h,t_h] - A[s_l-1,t_h] - A[s_h,t_l-1] + A[s_l-1,t_l-1]$ is equal to the number of points in the rectangle (s_l,t_l,s_h,t_h). □

Finally, we consider the complexity of this algorithm. It is easy to verify that each step in *is_pt_in_rectangle* takes at most $O(m^2)$ time.

4 Discussion: Optimization and Extensions

4.1 Reducing the Space Complexity

One drawback of the above algorithm is that it requires $O(m^2)$ space. This may be unacceptable when m is large. We now show that the space requirements of the algorithm in addition to the input (points and rectangles) can be reduced to $O(m+r)$ where r is the number of rectangles. First, we assume that the set of points are provided as a list of coordinates rather than a $m \times m$ boolean array. This is similar to the way the rectangles are provided in the algorithm of Fig. 1. Next, we observe that the algorithm uses four values from the array A for each rectangle (s_l,t_l,s_h,t_h):

$$A[s_h,t_h], A[s_l-1,t_h], A[s_h,t_l-1], A[s_l-1,t_l-1].$$

We will store these values with each rectangle. The only problem is how to determine these values. We cannot use the entire array A since that itself is $O(m^2)$. The next important observation is that calculation of the i^{th} row of A does not need any entry from any row lower than $i-1$. A similar observation applies to columns. As a result, keeping the previous row of A is sufficient to calculate the next row of A With above observations, it is easy to verify that the algorithm takes additional $O(m+r)$ space.

4.2 Finding k points in l rectangles, when max$\{k,l\} \ll m$

Since there are m state intervals in a process, in the worst case there can be $O(m^2)$ consistent sub-cuts for processes P_1 and P_2. This implies that if almost all sub-cuts satisfied Φ_1, then the size of $G(\Phi_1)$ is also $O(m^2)$. Similar reasoning can be applied for the number of rectangles in $R(\Phi_2)$. However, in some cases it can be expected that the number of points in $G(\Phi_1)$ as well as the number of rectangles in $R(\Phi_2)$ is much smaller. In these cases, we show that the above algorithm can be modified to reduce its computational time. We first encode each rectangle using its bottom left corner and the upper right corner. With this convention, we have $k+2l$ points in all – k for the points themselves and $2l$ for the l rectangles. These $k+2l$ points can be sorted for each of the two dimensions, an operation which is not more than $O(m \log m)$ where m =max$\{k,l\}$. The coordinates of each point are now the rank of each dimension. A $(k+2l) \times (k+2l)$ space is created in which to place all points. The *is_pt_in_rectangle* algorithm is now used to detect any of the k points within any of the l rectangles. This approach requires $O((k+2l)^2)$ operations.

As an example, suppose that there are two points (3,15) and (6,12) and a rectangle specified by opposite corners (5,7) and (10,21). The coordinates can be ordered for the two dimensions as follows: 3,5,6,10 and 7,12,15,21. Mapping these sequences to the sequence 1,2,3,4 results in a mapping of the points to (1,3) and (3,2) and the rectangle corners to (2,1) and (4,4). Running the algorithm on this reduced space will give the same result as the original: a point is contained in the rectangle.

There are other approaches to finding k points in l rectangles. For example, Preparatos and Shamos [4] present an algorithm for finding k points in a rectangle. Applying their algorithm l times is slightly more efficient when k and $l \ll m^2$. Their algorithm would use only a k^2 space for the k points created in $O(k^2)$ time. The check for each rectangle is $O(\log k)$ because a binary search is performed on each of the four corners to determine where in the k^2 space it lies. With l rectangles their search is $O(k^2 + l \log k)$.

4.3 The algorithm for $j+k$ variables

We next demonstrate how to detect a predicate of the form

$$\Phi_1(x_1, ..., x_j) \wedge \Phi_2(y_1, ..., y_k)$$

where $x_1, ..., x_j$ are on $P_1, ..., P_j$ and $y_1, ..., y_k$ are on $P_{j+1}, ..., P_{j+k}$. The algorithm works as follows:

1. All $g(\Phi_1)$ and $g(\Phi_2)$ are found, and placed in the sets $G(\Phi_1)$ and $G(\Phi_2)$. These operations are $O(m^j)$ and $O(m^k)$ respectively.

2. The sub-cuts $g(\Phi_1) \in G(\Phi_1)$ found in step 1 are now used to create points for a j-dimensional space. Each sub-cut specifies one point. As for the case of 2 variables, we also determine a j-dimensional box for each of the sub-cut in $G(\Phi_2)$. By using predecessor and successor clocks, the box corresponding to a sub-cut in $G(\Phi_2)$ can be determined in $O(jk)$ operations. Since there are $O(m^k)$ sub-cuts, finding all the boxes requires $O(jkm^k)$ operations.

3. Following the procedure of the *is_pt_in_rectangle* algorithm, all points and boxes found in steps 2 and 3 are checked for overlap. There are $O(m^k)$ boxes, each of which is of dimension j. To check containment in each box is an $O(2^j)$ operation because of the 2^j corners of a box in a j-dimensional space. Therefore, this step requires $O(2^j m^k)$ operations.

Therefore, for the general case, the complexity is $O(m^j + m^k + jkm^k + 2^j m^k) = O(m^j + (jk + 2^j)m^k)$.

Acknowledgments

We are thankful to Scott Stoller and referees of the paper for their comments.

References

[1] T. Basten, "Breakpoints and Time in Distributed Computations," *LNCS 857*, pp340-354, 1993.

[2] 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.

[3] 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.

[4] F. Preparata and M. Shamos, *Computational Geometry: An Introduction,* Springer-Verlag, 1985.

[5] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th International Workshop on Distributed Algorithms*, pages 318–332, France, September 1995. Springer-Verlag.

[6] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. ACM/ONR.