Some Optimal Algorithms for Decomposed Partially Ordered Sets

Vijay K. Garg

Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712-1084
email: vijay@pine.ece.utexas.edu

Abstract

We describe two problems and their optimal solutions for partially ordered sets. We first describe an optimal algorithm for computing the largest anti-chain of a partially ordered set given its decomposition into its chains. Our algorithm requires $O(n^2m)$ comparisons where n is the number of chains and m is the maximum number of elements in any chain. We also give an adversary argument to prove that this is a lower bound. Our second problem requires us to find if the given poset is a total order. Our optimal algorithm requires $O(mn \log n)$ comparisons. These algorithms have applications in distributed debugging and recovery in distributed systems.

Keywords: Partially-ordered sets, Distributed debugging

1. Introduction

Let $(S, <)$ be any partially ordered finite set. We are given a decomposition of S into n sets $P_0, ...P_{n-1}$ such that for any i, P_i is a chain of size at most m. We call such a structure a ***decomposed*** poset. These structures arise in distributed systems, because any execution of a distributed program can be viewed as a decomposed poset with local states of the processes as elements of the poset, and ordering between these states as that imposed by ***happened-before*** relation defined by Lamport[1].

Our first problem is to find an element s_i in each P_i such that $< \forall i, j : i \neq j : s_i \not< s_j >$. In other words, we have to find an anti-chain of size n if one exists. Clearly, this anti-chain is the largest as there cannot be any anti-chain of size greater than n. Our second problem is to find any two elements s_i and s_j such that $(s_i \not< s_j) \wedge (s_j \not< s_i)$. If no two such elements exist, then the given poset is actually a totally ordered set.

These problems have many applications in distributed processing. In distributed debugging[2,5], one may need to detect a global snapshot so that the local condition C_i is true in process i. Each process may then detect C_i locally and send the states in which it became true to a coordinator process. The task of the coordinator is to find one state in each process i in which C_i is true and states are causally unrelated to each other. Intuitively speaking, these states can be considered to have occurred simultaneously. We call such a set of states a consistent cut. For example, consider an implementation of the dining philosophers problem. Let $holding_i$ denote the condition that $philosopher_i$ is holding a fork. We may want to detect the condition $< \forall i : holding_i >$ if it ever becomes true for any consistent cut. Assume that each $philosopher_i$ sends the state in which $holding_i$ became

true to a coordinator process. The task of the coordinator is to find causally unrelated sets of states in which $holding_i$ is true. If it succeeds then we can claim that for some execution speeds of the philosophers the algorithm may lead to a deadlock.

As another example, consider the problem of recovery in a distributed system[3]. While a distributed computation proceeds, it is desirable to keep a global consistent cut so that in case of any fault the computation may start from the consistent cut instead of the beginning. Assuming that local checkpointing and messages-logging is done asynchronously, each process outputs states from which it can restart its local computation. The task again is to find a consistent cut.

The second problem may be useful in debugging a mutual-exclusion algorithm. Each process may record its state when it accessed the critical region. If there is any pair of concurrent states in which critical region is accessed, i.e., $(s_i \not< s_j) \wedge (s_j \not< s_i)$, then mutual exclusion can be violated with appropriate processor speeds.

In this paper, we describe an algorithm that requires $O(n^2m)$ comparisons for finding the largest anti-chain and an algorithm that requires $O(mn \log n)$ comparisons to find any anti-chain of size two.

2. Consistent Cut Problem

We use $s||t$ to denote that s and t are concurrent, i.e., $(s_i \not< s_j) \wedge (s_j \not< s_i)$. The consistent cut problem is: Given $(S, <)$ and its decomposition $(P_0, P_1, ..., P_{n-1})$, does there exists $\{s_i \in P_i\}$ such that $(\forall i, j : i \neq j : s_i||s_j)$? We first present an off-line algorithm which assumes that the entire poset is available at the beginning. Later, we adapt this algorithm for on-line detection of consistent cut in a distributed environment.

2.1 Algorithm

We will assume that any two elements of the set can be compared in O(1) time. This is true in distributed processing if states are time-stamped with vector clocks[4]. Our algorithm uses one queue q_i per chain to store elements of p_i. We assume that elements within a queue are sorted in increasing order. In the following discussion, we use the following operations on queues:

insert(q,elem); insert elem in the queue q
deletehead(q); remove the head of the queue
empty(q); true if q is empty
head(q); first element if ¬ empty(q), returns ***max*** (the biggest element) otherwise

The algorithm is given below:

function antichain($q_0, ..., q_{n-1}$:queues):boolean;
(* returns true if there is an anti-chain of size n, false otherwise. If true, the anti-chain is given by the head of all the queues *)
const all = {0,1,...,n-1};
var low,newlow: subsets of all;
i,j: 0..n-1;
begin
low := all;

```
    while low ≠ φ do
    begin
        newlow := {};
        for i in low do
            for j in all do
                if head(q_i) < head(q_j) then newlow:=newlow ∪ {i};
                if head(q_j) < head(q_i) then newlow:=newlow ∪ {j};
        low := newlow;
        for i in low do
            deletehead(q_i)
    end
    return(∀i : ¬ empty(q_i))
end
```

The algorithm works as follows. It compares only the heads of queues. Moreover, it compares only those heads of the queues which have not been compared earlier. For this purpose, it uses the variable *low* which is the set of indices for which the head of the queues have been updated. The invariant maintained by the ***while*** is

$(I) \qquad (\forall i, j \notin low : \neg empty(q_i) \wedge \neg empty(q_j) \Rightarrow head(q_i) || head(q_j))$

I is true initially because *low* contains the entire set. The while loop maintains the invariant by finding all those elements which are lower than some other elements and including them in *low*. This means that there can not be two comparable elements in $all - low$. The loop terminates when *low* is empty. At that point, if all queues are non-empty, then by the invariant I, we can deduce that all the heads are concurrent. The procedure clearly terminates because the number of elements in the queues decreases on every execution of the while loop unless *low* is ϕ in which case the procedure terminates.

The above procedure can be made to terminate earlier when any of the queue becomes empty. For that purpose, it is enough to introduce a variable *noempty* which is made false whenever the last element in any queue is deleted. This flag is also a part of the while condition. We have not done so for clarity of the discussion.

We now discuss the complexity of above algorithm. Our complexity analysis will be done based on the number of comparisons used by the algorithm. The following theorem shows that the number of comparisons are quadratic in n and linear in m.

Proposition 1 *The above algorithm requires at most $O(n^2m)$ comparisons.*

Proof: Let $comp(k)$ denote the number of comparisons required in the k^{th} iteration of the while loop. Let t denote the total number of iterations of the while loop. Then, total number of comparisons $= \sum_{k=1}^{k=t} comp(k)$. Let $low(k)$ represent the value of *low* at the k^{th} iteration. It is *all* in the first iteration. We note that $|low(k)|$ for $k \geq 2$ represents the number of elements deleted in the $k-1$ iteration of the while loop. Therefore, $\sum_{k=2}^{t} |low(k)|$ = total elements deleted < total elements in system $\leq mn$.

From the structure of the for-loops we get that $comp(k) = O(n * |low(k)|)$. Therefore, the total number of comparisons required are
$\sum_{k=1}^{t} comp(k) = n * \sum_{k=1}^{t} |low(k)|$
$= n * n + n * \sum_{k=2}^{t} |low(k)|$
$\leq n * n + n * mn = O(n^2 m)$

■

The above algorithm assumes that the entire queues were available as the input at the beginning. For many applications, the queues may be available only one element at a time. The problem of on-line computation of consistent-cut is to detect the consistent cut as soon as all elements forming the cut are available. This cut corresponds to the infimum element of the lattice of all consistent cuts in S[4]. The following algorithm assumes that a centralized process receives all the elements in the set S one at a time. This algorithm is a minor variant of the previous algorithm. The main difference is that an empty queue signifies that no consistent cut has been found so far. The centralized process may receive more elements in future and succeed in finding one. It computes only on receiving a message from some process. It maintains the assertion that heads of all non-empty queues are incomparable. The computation is shown below:

```
Upon recv(elem) from P_i do
begin
    if ¬ empty(q_i) then insert(q_i, elem)
    else begin
        insert(q_i, elem)
        low := { i }
        while low ≠ φ do
        begin
            newlow := {};
            for k in low do
                for j in all do
                    if head(q_k) < head(q_j) then newlow:=newlow ∪ {k};
                    if head(q_j) < head(q_k) then newlow:=newlow ∪ {j};
            low := newlow;
            for k in low do
                deletehead(q_k)
        end;(* while *)
    if ∀k : ¬empty(q_k) then found:=true
end
```

2.2 Adversary Arguments

In this section we show that the complexity of the above problem is at least $\Omega(n^2 m)$, thus showing that our algorithm is optimal. We first show an intermediate lemma which handles the case when the size of each queue is exactly one, i.e. $m = 1$.

Lemma 2 *Let there be n elements in a set S. Any algorithm which determines if all elements are incomparable must make at least $n(n-1)/2$ comparisons.*

Proof: The adversary will give to the algorithm a set in which either zero or exactly one pair of elements are incomparable. The adversary also chooses to answer "incomparable" to first $n(n-1)/2-1$ questions. Thus, the algorithm cannot determine if the set has a comparable pair unless it asks about all the pairs.

■

We use the above Lemma to show the desired result.

Proposition 3 *Let $(S,<)$ be any partially ordered finite set of size mn. We are given a decomposition of S into n sets $P_0,...P_{n-1}$ such that P_i is a chain of size m. Any algorithm which determines if there exists an anti-chain of size n must make at least $mn(n-1)/2$ comparisons.*

Proof: Let $P_i[k]$ denote the k^{th} element in P_i^{th} chain. The adversary will give the algorithm S and P_i's with the following characteristic:
$\forall i,j,k : P_i[k] < P_j[k+1]$
Thus, the above problem reduces to m instances of the problem which checks if any of the n elements is incomparable. The algorithm for the adversary can be stated as follows:
var num[k]:integer initially 0;
{ *number of questions asked about level k*}
On being asked to compare $P_i[k]$ and $P_j[l]$
if $(k<l)$ **then** return $P_i[k] < P_j[l]$
if $(l<k)$ **then** return $P_j[l] < P_i[k]$
if $(l=k)$ **then begin**
num[k]++;
if (num[k] = n*(n-1)/2) **then** return $P_i[k] < P_j[l]$
else return $P_j[l]||P_i[k]$
end
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.

■

3. Total Order Problem

Let $(S,<)$ be any decomposed partially ordered set. Our problem is to find if there exists elements s_i and s_j such that $s_i||s_j$. In other words, we have to find if the given decomposed poset is a total order.

3.1. An Algorithm

We first provide a sequential algorithm for the above problem. It can be seen that if there is a total order then the above poset can be sorted. Thus any sorting algorithm will

answer the query in $O(mn \log mn)$ time. However, we have not exploited the fact that all events within a chain are comparable. To do so, we employ the following algorithm:

$L = \{P_0, P_1, P_2, ..., P_{n-1}\}$
while $|L| \neq 1$ **do**
begin
    $X_i := removemin(L)$;
    $X_j := removemin(L)$;
    $Y := merge(X_i, X_j)$;
    $insert(L, Y)$
end

L in the above algorithm is a set of all chains known so far. It may be a heap in an actual implementation. $removemin$ removes the chain from L with the smallest size. $merge$ inputs two chains and outputs a merged chain if all elements are comparable. The complexity of merging two lists X_i and X_j is $O(|X_i| + |X_j|)$. Each chain is involved in at most $O(\log n)$ merges. In each merge it makes a contribution of m. Since there are n chains, the total number of comparisons is $O(mn \log n)$.

The above algorithm assume that all elements of the poset are available as input. An on-line computation can be done as follows. All the elements seen so far are kept in a sorted order. On receiving any new element, its position in the sorted list can be determined in at most $O(\log mn)$ comparisons. Thus, $O(mn \log mn)$ comparisons would be required to detect the first anti-chain. This algorithm is similar to insertion sort used for sorting an array of integers. Note that if $n < m$, then $O(mn \log mn) = O(mn(\log m + \log n)) = O(mn \log m)$.

3.2. An Adversary Argument

The lower bound for the problem is $\Omega(mn \log n)$. We use our previous technique of dividing the poset in m levels. Given two elements s and t at levels i and j with $i < j$, the adversary always returns $s < t$. Thus, the task of the algorithm is reduced to finding if each level is a total order. For each level, we show that at least $\Omega(n \log n)$ comparisons are required. We just need to show that for any poset of size n, at least $\Omega(n \log n)$ comparisons are required to determine whether it is a total order. To prove this we claim that the algorithm must ask enough questions so that it can determine the precise order just to answer if the set is totally ordered or not. In other words, the algorithm must ask enough questions to be able to "sort" the poset. If not, there is a pair of elements e and f for which all the answers are consistent for both $e < f$ and $f < e$. In this case, the adversary can always produce a poset that is inconsistent with the answer given by the algorithm. As the number of comparisons required to sort is $\Omega(n \log n)$, so is the complexity for determining if any level is a total order.

4. Conclusions

We have described the notion of a decomposed poset and its relevance to distributed systems. We have also discussed two important problems in connection with decomposed posets and provided optimal solutions to them.

Acknowledgements

We would like to thank anonymous reviewers of this paper for their suggestions. This research was supported in part by a NSF Grant CCR-9110605, a Navy Grant N00039-91-C-0082, and a TRW faculty assistantship award.

5. References

[**1**] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM* 21(7):, July 1978, 95-114.

[**2**] V.K. Garg, B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," Electrical and Computer Engineering Department, University of Texas at Austin, 1992, submitted for publication

[**3**] D.B.Johnson, W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms Vol. 11, No. 3*, Sept 1990, 462-491.

[**4**] Friedemann Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, M. Cosnard et al.(eds.), Elsevier, North-Holland, (1989), 215-226.

[**5**] B. Waldecker, and V.K. Garg, "Detection of Strong Predicates in Distributed Programs," *Proc. IEEE Symposium on Parallel and Distributed Processing*, December 1991, 692-699.