Online Algorithms for Dilworth's Chain Partition

Selma İkiz & Vijay K. Garg
Parallel and Distributed Systems Laboratory, Dept. of Electrical and Computer Engg
The University of Texas at Austin
Austin TX 78712

Abstract

There are many interesting applications of partial order theory in distributed and parallel systems. These include testing and monitoring the concurrent behaviour of the system. Dilworth's chain partition and width of the partial order plays a key role when the trace is modelled as such. In this paper, we discuss the desicion problem of testing the width of a partially ordered set, and finding the Dilworth's chain partition in an online fashion. We present an online algoritm with worst case time complexity being $O(wn^2)$ for finding the Dilworth's chain partition. We implement two of the previously known offline algorithms. In particular, we compare the experimental performances of online Dilworth's chain partition algorithms with the offline ones.

1 Introduction

Partial order theory now plays an important role in many disciplines of computer science and engineering. For example, it has applications in distributed computing, concurrency theory, programming language semantics, and data mining. The theory is also useful in disciplines of mathematics such as combinatorics, number theory and group theory.

Partially ordered sets (posets) are particularly accurate for modeling dynamic behavior of complex systems that can be captured by causal relations. Capturing the partial order online appears in distributed system in the context of the analysis of distributed systems. A distributed program is consists of n sequential processes which communicate and synchronize only by means of message passing. A distributed computation describes the execution of a distributed program. However, a distributed computation do not yield a linear sequence of events. The relationship between events defines a partial ordering. For a particular computation, events produced by each process are totally ordered and communications create dependencies among events belonging to distinct processes. In a seminal paper Lamport [17] calls this partial order the *happened-before order*. Later, Fidge [7] and Mattern [18] independently introduced vector clocks to timestamp events such that, happened-before relationship between any two events can be determined by examining their timestamps. These timestamps allow online computation of this order induced by the distributed computation, and then the checking step can be done during the computation.

Bouchitte et al. [3] discuss *online* and *offline* algorithms for the interval dag recognition, and algorithms building the transitive reduction of two lattices usually associated with posets which are the ideal lattice and the maximal antichain lattice, respectively. Mostly motivated by the importance of partial order theory in distributed systems, we like to investigate the problem of finding the Dilworth's chain partition, and the width of the poset further. One of the fundamental parameters of a partial order is the width, which corresponds to the maximum number of mutually incomparable elements. In this paper, we are interested in efficient online algorithms for the Dilworth's chain partition problem of distributed program traces. We introduce a novel online algorithm that has $O(swn)$ time complexity, where w is the width, n is the size of the poset, and s is the maximum number of elements between two observed antichains of poset. All of the algorithms discussed in this paper are centralized and events are represented as vector clocks. Our result is also useful in other applications of partial theory. Recently, partial order theory also receives attentions from the application community.

The paper comprises of six sections. In section 2 we give basic definitions of partial order sets and vector clocks as background. In section 3 we compare two algorithms on finding an antichain of size w, and partitioning the poset into w, mainly finding the Dilwort's partition. First algorithm is given by Tomlinson and Garg [21], hereafter we refer as $QueueMerge$. Second algorithm by Bogart and Magagnosc [9] is referred as $ReduceChain$. In section 4, we compare these algorithms in an online fashion, and discuss their drawbacks compared to each other and their

offline versions. In section 5, we present a framework of online algorithm development, and novel online algorithms that improve on the previous ones.(Change this sentence). In section 6, we present the simulation results of the presented algorithms. We also give the details of implementations and testing environment. In section 7, some of the related applications for the need of such online algorithms are portrayed.

2 Background

While we a detailed treatment of partial-order theory is not given, we will review a few of the important definitions.

A pair (X, P) is called a partially ordered set or *poset* if X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. We simply write P as a poset when X is clear from the context. We call X the *ground set* while P is a *partial order* on X. The $\leq$ and *divides* relations on the set of natural numbers are some examples of partial orders.

We write $x \leq y$ and $y \geq x$ in P when $(x, y) \in P$. Also, $x < y$ and $y > x$ in P means $x \leq y \in P$ and $x \neq y$. Let, $x, y \in X$ with $x \neq y$. If either $x < y$ or $y < x$, we say x and y are *comparable*. On the other hand, if neither $x < y$ nor $y < x$, then we say x and y are *incomparable* and write $x \| y$.

A poset (X, P) is called a *total order* or a *linear order*, if every distinct pair of points from X is comparable in P. It is possible to extend any partial order to a linear order by adding order between unordered elements. Each such linear order is called a *linearization* of the partial order. A total order L is said to be a *linear extension* of P, if L is a total order on the same ground set X of P, such that each couple of elements $u, v \in X$ for which $u \leq_P v$ implies $u \leq_L v$.

Similarly, we call a poset an *antichain* if every distinct pair of points from X is incomparable in P. The *height* of poset is defined to be the largest chain in the poset and is denoted by $height(P)$. Similarly, the *width* of a poset is defined to be the largest antichain in the poset and denoted by $width(P)$, hereafter will be refered as w.

A famous theorem of R. P. Dilworth [4] states that if an ordered set P has a maximum sized antichain with k elements i.e., $w = k$ then, P can be partitioned into k chains. Moreover, the set of all antichains of size k forms a distributive lattice [9].

3 Offline Algorithms

The width of a poset P, w, is the size of the largest antichain. A partition of a poset P is called *a Dilworth's chain partition*, if P is partitioned into w chains. There are two principle approaches for finding this partition: Partitioning the poset greedily into N initial chains, and reduce the number of partitions by finding the alternating sequences, or reduce the problem to the bipartite matching problem [8].

3.1 Alternating sequence approach

For an execution trace of a distributed program, finding the minimum chain cover of a poset seems more appealing since the history of execution can easily be accumulated as a chain on each process, which is the trivial partition of an execution. At the termination of the execution, there are N such chains, one history for each process, and reducing the chains until we reach an antichain of size w sounds trivial. All events in an antichain are pairwise concurrent, essentially incomparable. However, concurrency is not a transitive relation which makes the problem delicate. We define the offline version of the problem as follows:

Definition 1 *Let P be a partially ordered set with n elements. Given a chain partition of P, $C = \{C_0, \ldots, C_{t-1}\}$ into t disjoint chains, the problem is to rearrange these chains into a chain partition with fewest chains, or to show that no such rearrangement is possible.*

Bogart and Magagnosc [9] solves this problem by defining the reducibility conditions, and alternating sequence. They begin with the trivial chain partition, the one in which each element of poset is a one element chain. In $O(n^2)$, a lookup table is constructed. This table essentially is the adjacency list representation of a poset. With this setup, the procedure *Reduce* is called. Let C be a chain partition, then it either finds a reducing sequence, forms the chains by calling $redoChains$, and returns the reduced C; or returns the original chain partition C. The procedure *Reduce* uses a breadth first search to find an alternating sequence, and then executes $redoChains$. The complexity of *Reduce* and $redoChains$ are $O(n+e_{\leq})$ and $O(n)$ [9]. For the completeness, the *Reduce* and $redoChains$ algorithms are given in Figure 1 and Figure 2, respectively.

Tomlinson and Garg [21] solve this problem using a notable data structure. Given N chains, $QueueMerge$ answers the question that is whether there is an antichain of size at least K. Their result follows from Dilworth's theorem that a poset can be partitioned into $K-1$ chains if and only if there does not exist an antichain of size at least K. Therefore, procedure $QueueMerge$ takes N chains and calls $Merge$ subroutine repeatedly, and returns either $K-1$ chains, or an antichain of size at least K. The algorithm uses queues to represent chains. Each

```
procedure reduce()
 N ← ∅; M ← ∅
 for x ∈ P do
     USED[x] ← false; PAIR[x] ← ∅
 endfor
 for c ∈ C do push(first(c), N) endfor
 E ← N
 while N ≠ ∅ do
     for a ∈ N do
         I ← I[a]
         for b ∈ I with USED[b] = false do
             T ← TAIL[b]
             if |T| > 1 then //b has a successor, a'
                 a' ← second(T); PAIR[a'] ← (a, b)
                 USED[b] ← true; push(a', E);
                 push(a', M)
             else
                 return(redoChains(a, b))
             endif
         endfor
     endfor
     N ← M; M ← ∅
 endwhile
 return(C)
endprocedure
```

Figure 1: Reduce Algorithm

queue is stored in increasing order so that the head of the queue is the least element in the queue.

The $Merge$ is performed by repeatedly selecting an element and removing it from one input queue and placing it in an output queue. This continues until either one of the input queues is empty or no such element can be found. In the second case, an antichain of size K is returned. The notable data structure is a spanning tree formed by input and output queues. Input queues are represented as nodes while edges are represented as output queues. While the spanning tree structure is preserved, the tree changes dynamically.

The $QueueMerge$ algorithm calls the $Merge$ function $N - K + 1$ times. The $Merge$ function takes K queues as input. Instead of reducing N chains to $N - 1$, then $N - 1$ chains to $N - 2$, Tomlinson and Garg [21] choose K chains and calls $Merge$ function on this selected subset. To minimize the number of comparisons, they use the idea from classic merge techniques used for sorting. They choose the chains that have been merged the fewest number of times. For the completeness, the $QueueMerge$ and $Merge$ algorithms are given in Figure 3 and Figure 4, respectively. The complexity of $Merge$ is given as Kl, where l is the total number of elements to be merged. The upper bound of the $QueueMerge$ is given as $KMN(K + log_\rho N)$ where M is the number of elements in the largest queue, and ρ is the reducing factor and equals to $K/(K-1)$. The lower bound is given as $\Omega(KMN)$ [21].

We can employ the rotating technique to the *Reduce* algorithm as well. Therefore, it is more realistic to compare the performances of only rounds of $Merge$ and *Reduce*, instead of $QueueMerge$ and *Reduce*. Moreover, we assume the Dilworth's chain partition instead of the decision question whether there is an antichain of size K. This is the other way of asking whether the width of the poset w is less than K.

The complexity of offline algorithms could be computed as follows: Let the initial partition be given in N disjoint partitions. Subsequently modified offline algorithms have to call $Merge$ and *Reduce* subroutines $N - w$ times to obtain the Dilworth's chain partition. Hence, the total complexity of *Reduce* is $O((N - w)(n + e_{\leq}))$, where the total complexity of $Merge$ is $O((N - w)Nn)$. The worst case complexity of both algorithms is the same, $O(n^3)$. *Reduce* algorithm reaches the worst case when the density of edges are high, the initial partion is the trivial partition, and the width of the poset is small. $Merge$ algorithm reaches the worst case when the initial partion is the trivial partition, and the width of the poset is small.

```
procedure redoChains(a, b)
 while PAIR[x] ≠ ∅ do
     rest(TAIL[b]) ← TAIL[a]
     PAIR[x] ← {first(PAIR[a]), second(PAIR[a])}
 endwhile
 rest(TAIL[b]) ← TAIL[a]
 remove(TAIL[a], C)
 return(C)
endprocedure
```

Figure 2: Redo Procedure

```
procedure
 QueueMerge(K:integer, Qlist:list of queues):antichain;
 assume: 1 ≤ K ≤ |Qlist|
 assume: q ∈ Qlist ⇒ q ≠ ∅
 for (n := |Qlist|; n ≥ K; n := n − 1)
     (p_1, ..., p_N) := Qlist);
     (p_1, ..., p_K) := Merge(p_1, ..., p_K);
     if (q_K = ∅) then
         Qlist := (p_{K+1}, ..., p_n, q_1, ..., q_{K−1});
     else
         return(head(q_i)|1 ≤ i ≤ K);
     endif
 endfor
 return(∅)
endprocedure
```

Figure 3: QueueMerge Algoritm

3.2 Bipartite approach

The second approach has been to reduce the problem of finding the width of a poset to a maximum matching problem in a bipartite graph and then using the results of maximum matching problem [8]. Given a bipartite graph $G = (U, V, E)$, with $n = |U| + |V|)$ and $m = |E|$, the bipartite matching problem is finding a set of edges $M \subseteq E$ of maximum cardinality such that no edge in the set shares a vertex with any other edge in the set. This equivalence follows from the proof of the Dilworth's theorem.

Theorem 1 *The maximum size of an antichain in P equals the minimum number of chains needed to cover the elements of P (Dilworth's theorem).*

The proof of the above theorem is given by Fulkerson. He uses the notion of split of a graph. We give the definition of split graph first.

Definition 2 *Given a poset $P = (X, \leq)$, split $S(P)$ of P is a bipartite graph having as vertices two copies of X',X'' of X and an edge (x', y'') whenever $(x \leq y)$ in P.*

```
procedure
 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 : 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;
           endif
           if head(P_j) < head(P_i) then
                move := move ∪j;
                bigger[j]:= i;
           endif
      endfor
      for i ∈ move do
           dest:= FindQ(G,i,bigger[i]);
           x := removehead(P_i);
           insert(Q_dest,x);
      endfor
      ac := all -move;
 endwhile
 if (∃i :: P_i = ∅) then
      FinishMerge(G, P_1, ..., P_K, Q_1, ..., Q_K−1);
      return (Q_1, ..., Q_K−1, ∅)
 else
      return(P_1, ..., P_K);
 endif
endprocedure
```

Figure 4: Merge Algorithm

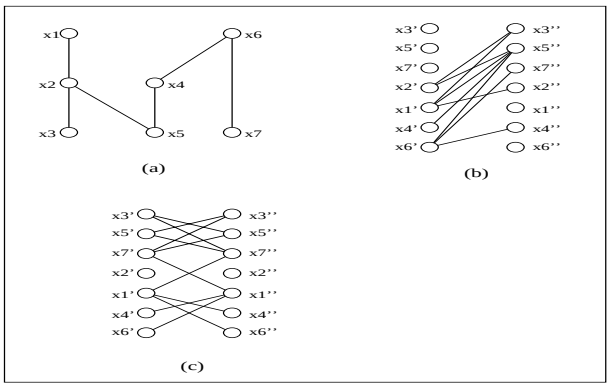

Figure 5: PWP ⇒ MMP. (a) A poset (b) Split graph of P (c) Dual of split graph

An example of a poset, and its corresponding split graph is given in Figure 5.
A matching M in $S(P)$ corresponds to a partition of P into $|X| - |M|$ chains. To see this, begin with a partition of P having 1 element chains. For each edge $(x', y'') \in M$ combine the chains ending with x and beginning with y hence reduce the number of chains by one. Let $A_U = \{x \in X : x', x'' \notin U\}$ where U is the minimum vertex cover of $S(P)$. Then, it is easy to see that A_U forms an antichain. $|A_U| = |X| - |U|$ follows from the fact that $S(P)$ has a transitive relation. Now, Dilwort's therom follows from the Konig-Egervary duality theorem; maximum matching problem in a bipartite graph (MMP) is equivalent to problem of finding the width of a poset (PWP).
An augmenting alternating path can be found with a depth first search in $O(n^2)$ and at most n augmentations along alternating path are possible. Thus using the alternating path algorithm for bipartite matching allows one to to compute the width of (P) in $O(n^3)$ time, and using the Hopcroft and Karp algorithm [13] improves this result to $O(n^{2.5})$ [10].
From a computational complexity point of view, in the case of sparse graphs, the best sequential algorithm for finding a maximum matching is by Hopcroft and Karp, which achives a worst-case running time of $O(\sqrt{n}m)$. In [20], Setubal showed that implemantation of the push-relabel algorithm developed by Goldberg [11] and generalized by Goldberg and Tarjan [12] was significantly faster than an implementation of Hopcroft-Karp algorithm. The push-relabel algorithm has a worst-case running time of $O(nm)$.
Partitioning the poset greedily into N initial chains, and reduce the number of partitions by finding the alternating sequences. To utilize the bipartite approach, we need to construct the bipartite graph in (n^2), and find the matching in (n^3). Complexity

wise, this approach is not different than *Reduce* algorithm. Hence, we do not consider this approach any further.

4 Online Algorithms

Online algorithms are algorithms working on dynamic structures with unpredictable behavior. An informal definition of an online approach for the problem at hand can be stated as follows: At each step, further information is added and we want to compute Dilworth's chain partition of the work already done. There are two frameworks for general online algorithms. The first one, greedy approach, assumes that a value computed at one step is never updated in a latter step. This kind of approach is particularly accurate for practical applications where a previous decision cannot be changed, e.g. scheduling problems or motion control. The second approach allows a value to be updated to reach the real solution. In this study, we are concerned with both of the approaches for two parts of the problem; finding the width of the known subposet $P' \subseteq P$, and partitioning the subposet into $w(P')$ chains. The first part of the problem is a decision question and we want to answer this accurately at each step without changing the answer of previous step. On the other than, each vertex can change the chain that it belongs at each step.

However, when dealing with posets, adding a new element with arbitrary predecessor and successor sets can lead to an inconsistency since the new vertex may induce new comparabilities between vertices of the already known poset. In order to preserve the structure of the poset, Kierstead [15, 16] makes an assumption called *subposet hypothesis*, and later Bouchitté et al. [2] defines online paradigm of posets as follows: The current knowledge is a dag $G = (X, E)$ whose transitive closure is a poset (X, P). The additional knowledge is a new vertex $x \notin X$ together with a list of predecessors, $p(x)$, and a list of successors, $s(x)$, where $p(x)$ and $s(x)$ are subsets of X. Then a new directed graph $G' = (X', E')$ is obtained such that $X' = X \cup x$, and $E' = E \cup \{(y, x), y \in p(x)\} \cup \{(x, y), y \in s(x)\}$. The transitive closure of G' is always a poset P' and P is a subposet of P'. Also some properties on the predecessor sets and successor sets associated at each step with the incoming vertex can be assumed. In a later work, Bouchitté et al. [3] define the *linear extension hypothesis* extending to these properties, i.e., the new vertex x is maximal in the new poset P', that is $s(x) = \emptyset$. In the online versions of our algorithms, we also assume *linear extension hypothesis*, that is the elements arrive in an increasing (or non-decreasing) order.

Given the $Merge$ and *Reduce* procedures, one simple way to form the online algorithm is calling these procedures whenever an element arrives. We refer to these algorithms as $onlineMerge$ and $onlineReduce$. The figures 6 and 7 show the online versions of the algorithms.

```
procedure OnlineMerge(v:vector clock):Q_1, ..., Q_K:queues;
 assume: 1 ≤ K ≤ |w(P')|
         //P' is the part of the execution seen
 Q = Merge(Q, v);
 return Q;
endprocedure
```

Figure 6: Online Merge Algoritm

Let the number of elements seen is i at step j, then running $Merge$ procedure once has time complexity of $O(wi)$. This result follows from the fact that at one iteration of $Merge$ algorithm, each element must be represented in the variable ac of the $Merge$ procedure 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 i at step j, then i elements must pass through ac on their way to the output queue for a total of Ki comparisons at step j. Since $i = j$, $K \leq w$, and total number of steps is n, the total complexity of the online $Merge$ algorithm is $O(wn^2)$. Running *Reduce* only once has time complexity of $O(i + e_{\leq})$. Since there are n steps, worst time complexity of *Reduce* is $O(n^3)$. Felsner et al. [10] argues that even for the decision problem $O(Kn^2)$ is optimal.

```
procedure OnlineReduce(v:vector clock):C_1, ..., C_K:chains;
 assume: 1 ≤ K ≤ |w(P')|
         //P' is the part of the execution seen
 C = Reduce(C, v);
 return C;
endprocedure
```

Figure 7: Online Reduce Algoritm

5 New Online Algorithms

In this section, we explore the ideas to increase the efficiency of the online algorithm. Since both online and offline versions of $Merge$ outperformed *Reduce*, hereon we only focus on the comparisions with $Merge$ algorithm.

The first idea is trying to place the element at the tail of one of the queues whenever an element arrives, instead of calling the $Merge$ subroutine immediately. As a first step to explore this idea, we implemented

```
procedure OnlineMerge1(v:vector clock):Q_1, ..., Q_K:queues;
 assume: 1 ≤ K ≤ w'
//w' is the width of the execution seen
 if ∃i : Q_i.tail ≤ v then
      Q_i.add(v);
 else
      Q = Merge(Q, v);
 endif
 return Q;
endprocedure
```

Figure 8: OnlineMerge1 Procedure

the procedure called $OnlineMerge1$ in Figure 8; Evidently, the efficiency of this algorithm depends on the $mergeCount$. We prove an upper bound and a lower bound. The upper bound is proven by defining an adversary which forces the $OnlineMerge1$ to call $Merge$ algorithm as much as possible. The lower bound is, in fact trivial, i.e., at least w calls should be made to reach w partitions.

5.1 An Upper Bound for Merge Calls

Proposition 1 *Let P be any partially ordered finite set of size n and width K. $OnlineMerge1$ calls $Merge$ subroutine at most $n.(K-1)/K$ times.*

Proof: We use an adversary argument. Let $OnlineMerge1$ algorithm at step j has K disjoint partition queues, i.e., the tails of the queues form a K antichain. As the next step, adversary cannot give a new element that forces $OnlineMerge1$ algorithm to call $Merge$, since such call increases the partition size to $K+1$ which is contradictory to the assumption that poset P has width K. However, for the next $K-1$ step adversary will give new elements that forces the $OnlineMerge1$ algorithm to call $Merge$. By the same reasoning before, at step $j+K$ adversary cannot give a new element that forces $OnlineMerge1$ algorithm to call $Merge$ as the new tails of the partitions also form a K antichain. Hence, $OnlineMerge1$ calls $Merge$ subroutine at most $n.(K-1)/K$ times. ■

The upper and lower bound of $OnlineMerge1$ algorithm are $O((K-1)n^2)$ and $\Omega(Kn)$, respectively. Motivated by this result, the second idea exploits the fact that an element of the poset arrives in a non-decreasing order, namely *linear extension hypothesis*. When we examined the algorithm for $Merge$, there are two different places where we can readily alter according to this hypothesis. First part is when there is an increase in the width of the poset, P', when a new element, v, arrives. $Merge$ algorithm finds the first antichain that contains v, and places all the elements that are properly below this antichain into the output queues, while the rest (the antichain and all the elements that are properly above this antichain) remains in the input queues. On the other hand, when $Merge$ subroutine reduces K queues into $K-1$ queues, this happens when there is no increase in the width of the poset P', $Merge$ places all the elements in the output queues. When there is an increase in the width of the poset P', *linear extension hypothesis* allows us to consider the elements only in the input queues. For an example

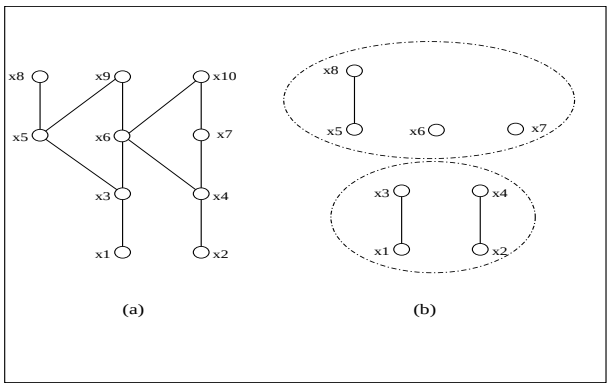

Figure 9: (a) A poset P (b) Partition of poset P' into work space and history

in Figure 9, $L = x_1, x_2, x_3, x_4, x_5, x_6, x_8, x_7, x_9, x_{10}$ is a linear extension of the given poset P. Let $X' = \{x_1, x_2, x_3, x_4, x_5, x_6, x_8\}$, and the new element v be x_7. $Merge$ places x_1 and x_3 into the first output queue, and places x_2, and x_4 into second output queue. $Merge$ also finds the antichain as $\{x_5, x_6, x_7\}$, and returns it. Now first input queue contains x_5, and x_8, while the second input queue contains x_6, and third input queue contains x_7. The following steps of partitioning should only consider the set $X' = \{x_5, x_6, x_7, x_8\}$, and corresponding poset P'. The complete procedure called $OnlineMerge2$ can be described as in Figure 10. P is set of input queues of $Merge$ subroutine as Q is set of output queues. $OnlineMerge2$ algorithm keeps the elements that are not used in the current work space in the history queues, Q'. The complexity of the above algorithm depends on the interval between two increments of the number of partitions. If we assume uniform distribution, then the number of elements in the work space is less than $n/(w-1)$. Hence, the complexity becomes $O(n^2)$.

The second improvement is applying the same idea when there is no increase in the width of the poset P'. However, after a reduction takes place, splitting the work space and history queues according to the

```
procedure OnlineMerge2(v:vector clock):Q_1,...,Q_K:queues,
                       Q'_1,...,Q'_K:queues;
 assume: 1 ≤ K ≤ w'
//w' is the width of the execution seen
 if ∃i : Q_i.tail ≤ v then
      Q_i.add(v);
 else
      if Merge(Q,v).reduced then
           Q = Merge.Q;
      else
           Q = Merge.P;
           Q' = Q' + Merge.Q;
      endif
 endif
 return Q, Q';
endprocedure
```

Figure 10: OnlineMerge2 Procedure

```
procedure OnlineMerge3(v:vector clock):Q_1,...,Q_K:queues,
                       Q'_1,...,Q'_K:queues;
 assume: 1 ≤ K ≤ w'
//w' is the width of the execution seen
 if ∃i : Q_i.tail ≤ v then
      Q_i.add(v);
 else
      if Merge(Q,v).reduced then
           Q = Merge.Q;
           Q' = Q' + Merge.Q;
      else
           Q = Merge.P;
           Q' = Q' + Merge.Q' + Merge.Q;
      endif
 endif
 return Q, Q';
endprocedure
```

Figure 11: OnlineMerge3 Procedure

last $K-1$ antichain is costly. This reduction technique should also blend well with the $Merge$ subroutine. We observe that the variable ac in the $Merge$ subrotine could also serve to this purpose. The variable ac keeps the maximum antichain according to the heads of the inQueues. When $|ac| = K$, the subrotine returns an antichain of size K and shows that no reduction is possible. We utilized ac to keep track of the $K-1$ antichains, i.e. when $|ac| = K-1$ $Merge$ subroutine appends the content of the output queues to the history queues, and empties the output queues. Whenever $Merge$ subroutine returns a reduced partition, online algorithm updates its work partition according to the output queues of $Merge$ as usual, and updates its history queues or partitions with the history queues of $Merge$. If we consider the previous example in Figure 9; assume that the execution seen is $X' = \{x_1, x_2, x_3, x_4, x_5\}$, and the new element v is x_6, then, in the $Merge$ subroutine the last seen $K-1=2$ antichain in the ac is $\{x_5, x_6\}$, and online algorithm updates history queues as $\{(x_1, x_3), (x_2, x_4)\}$ and work space as $\{(x_5), (x_6)\}$. This procedure called $OnlineMerge3$ can be depicted as in Figure 11; Details of new $Merge$ algorithm can be seen in Figure 12. New $Merge$ algorithm keeps three types of queues, namely input queues, otput queues and history queues. At the same time $OnlineMerge3$ algorithm keeps two types of queues work queues and history queues. At each step, either the work queues, or history queues grow. Once an element is placed into the history queues, it never appears into the work queues again. Besides, the partition of the history queues never changes. In Figure 11, P is the input queues of $Merge$ subroutine as Q is the output queues, and Q' is the history queues. $OnlineMerge3$ algorithm keeps the elements that are not used in the current work space in the

```
procedure
 Merge(P_1,...,P_K : queues) :
              Q_1,...,Q_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 : 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;
           endif
           if head(P_j) < head(P_i) then
                move := move ∪j;
                bigger[j]:= i;
           endif
      endfor
      for i ∈ move do
           dest:= FindQ(G,i,bigger[i]);
           x := removehead(P_i);
           insert(Q_dest,x);
      endfor
      ac := all -move;
      if (|ac| = K − 1) then
           for (1 ≤ k ≤ K) do
                Q'_k = Q'_k + Q_k;
                empty(Q_k);
           endfor
      endif
 endwhile
 if (∃i :: P_i = ∅) then
      FinishMerge(G, P_1,...,P_K, Q_1,...,Q_{K−1});
      return (Q_1,...,Q_{K−1}, ∅)
 else
      return(P_1,...,P_K);
 endif
endprocedure
```

Figure 12: Online Merge Algoritm

history queues, Q'. The complexity of the above algorithm depends on the frequency of antichains of in the poset. However, its worst case complexity is still $O(wn^2)$. If we assume that there are no more than s elements in the working partition then the complexity reduces to $O(wsn)$.

Proposition 2 *Let P be any partially ordered finite set of size n, width 2 and a linear extension L of P. $OnlineMerge3$ algorithm partitions P into 2 chains in $O(n)$ time.*

Proof: Let $L = x_1, x_2, \cdots, x_n$ be the linear extension of P which is assumed to be given as part of the input. Let $x_j || x_{j+1}$ be the first incomparable adjacent pair in L. The algorithm returns two queues that each contain either x_j or x_{j+1}. Moreover, because of the pruning, queues contain only x_j and x_{j+1}. When inserting a new element $x = x_t$, we first compare x to the tail elements of queues. If x is incomparable to both, we call merge subroutine.
Observe that, when width is 2, merge subroutine returns two queues containing only one element in each, and merge subroutine performs $2n$ comparisons to find the antichain of size 2.
Assume that we need to call merge subroutine in every r elements. Then, n/r calls are made with $2r$ comparisons each time. Other times, we make at most 2 comparisons.
In worst case we need to make 4 comparisons to place a new element, hence the complexity of the algorithm for width 2 is $O(n)$.

■

6 Experiments

In this section we demonstrate our simulation results of both online and offline algorithms. All the tests are done on a computer system equipped with 512Mb RAM and 1.13 Ghz Pentium3 processor. Each test case is performed 5 times, then the average is used in the analysis.

6.1 Offline Algorithms

At the implementation of the *Reduce* algorithm, chains are represented as linked-lists. Each event is represented as a *vectorClock* object, while adjacency list of an event is a vector containing pointers to *vectorClock* objects that are properly below this object. Moreover, adjacency lists are sorted to increase the efficacy of the algorithm.
At the implementation of the $QueueMerge$ and $Merge$ algorithm, queues are represented as *Queue* objects containing a vector of *vectorClock* objects. Moreover, instead of keeping a matrix for the spanning tree, output queues have two nodes that point to the input queues. The identical *vectorClock* objects are used for the two of the implementations.

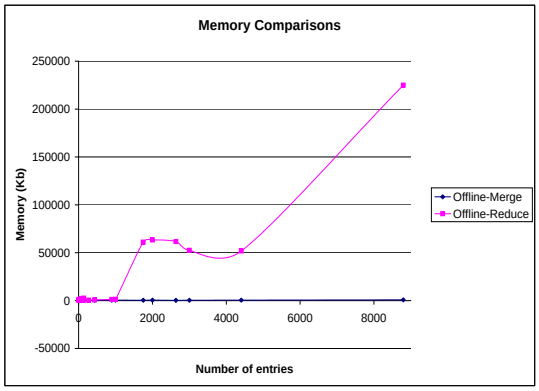

Figure 13: Memory usage of Offline versions of $Merge$ and *Reduce*

The algorithm $QueueMerge$ calls the $Merge$ function $N - K + 1$ times. The $Merge$ function takes K queues as input. Instead of reducing N chains to $N - 1$, then $N - 1$ chains to $N - 2$, K queues are choosen and $Merge$ function is called on this selected subset. To minimize the number of comparisons, the idea from classic merge techniques for sorting is used; use the queues that have been merged the fewest number of times.

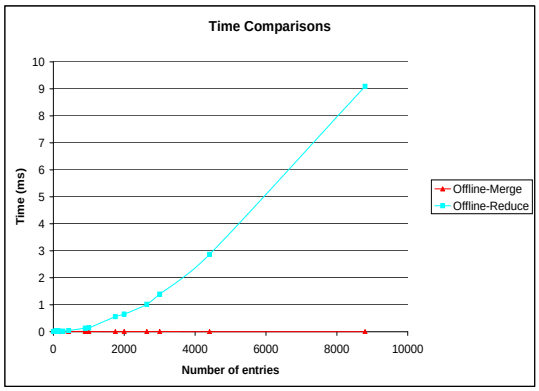

Figure 14: Running times of Offline versions of $Merge$ and *Reduce*

We used a test suite containing twelve tests of different sized partitions to test the implementations initially. This test is done using rotating technique for $QueueMerge$ algorithm, while only repeated calls is performed for *Reduce*. Also, test is performed by asking whether there exists a K antichain. $QueueMerge$ algorithm outperforms both in memory and time consumption. Also number of cycles differ due to the

decision question. We can employ the rotating technique to the second algorithm also. Therefore it is more realistic to compare the performances of only rounds of $Merge$ and $Reduce$ instead of $QueueMerge$ and $Reduce$.

We setup three different test suites such that each includes twelve simple test cases. The first test suite includes several small sized posets that has at most ninety elements. Second test suite includes posets of size 100 while width of posets change from 2 to 13. Third test suite includes fixed size initial partition of 6, while number of elements changes from 10 to 10000. $Merge$ outperforms the $Reduce$ both in memory consumption, see in Figure 13, and running times, see in Figure 14. Moreover, $Reduce$ gives *out of memory* error when the number of elements is bigger than 9000. This is, in fact, expected since constructing the lookup table is done in $O(n^2)$ and need space $O(n^2)$.

6.2 Online Algorithms

We implement the simple online versions of $Merge$ and $Reduce$. Same test cases that are used in the offline versions are used. Only this time we introduce one element at a time adhering to linear extension hypothesis. We observe that the online versions $Merge$ has at least twice better memory usages and running times than $Reduce$ Figure 15, and Figure 16, respectively.

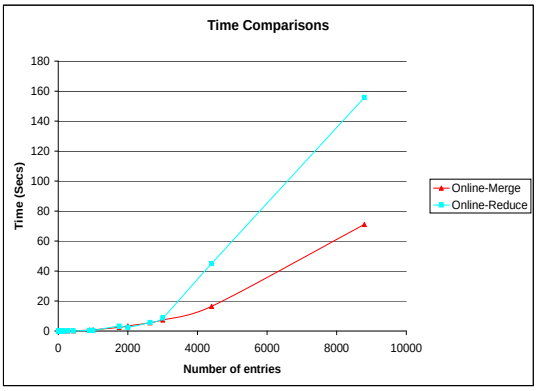

Figure 15: Running times of Online versions of $Merge$ and $Reduce$

Memory used for online versions illustrate the total amount used for the entire run. Although total memory usage of online algorithms are much higher than the offline versions, it is trivial to observe that memory usage of each step of online versions never exceeds the memory usage of the offline algorithms. Therefore, hereafter we do not compare memory usages of the online algorithms.

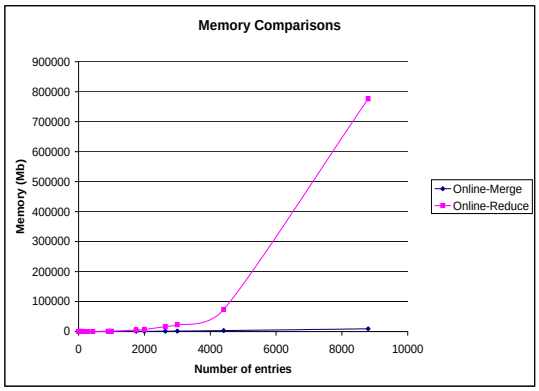

Figure 16: Memory usage of online versions of $Merge$ and $Reduce$

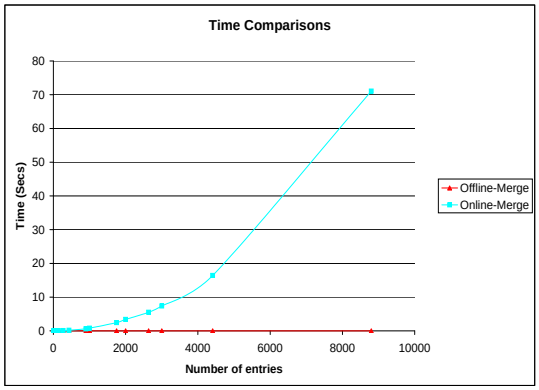

Figure 17: Running times of $Merge$ (offline vs. online)

Another comparison that is worthwhile is the running time differences of online and offline versions of these algorithms, see Figure[17- 18]. Although online version of $Merge$ algorithm performs better that online version of $Reduce$ algorithm, it is still far from the offline version of $Merge$.

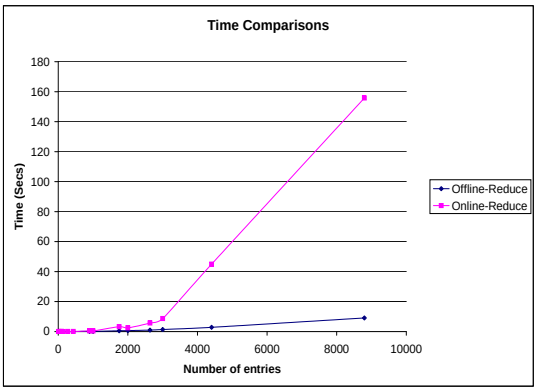

Figure 18: Running times of *Reduce* (offline vs. online)

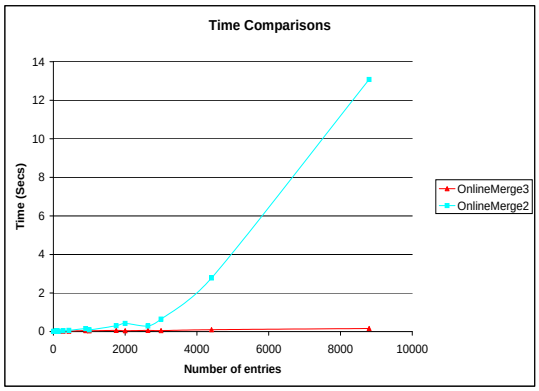

Figure 20: Running times of $OnlineMerge3$ vs. $OnlineMerge2$

6.3 New Online Algorithms

We implemeted the $OnlineMerge1$ algorithm to observe the frequency of $Merge$ callings. By using the same test suites, for each test case we observed that $Merge$ subroutine is called at most 22% of the times when a new element arrives.

We implemented $OnlineMerge2$ procedure, and run the test suites on this implementation. Since we know the bound on the memory usage, we will compare only running times of the algorithms hereafter. We compared it with the previous online and offline $Merge$ implementations, although we achieve quite an improvement on the previous $onlineMerge$ according to time usage, the improvement is poor when it is compared with the offline algorithm, see in Figure 19.

We implement $OnlineMerge3$ procedure, and run

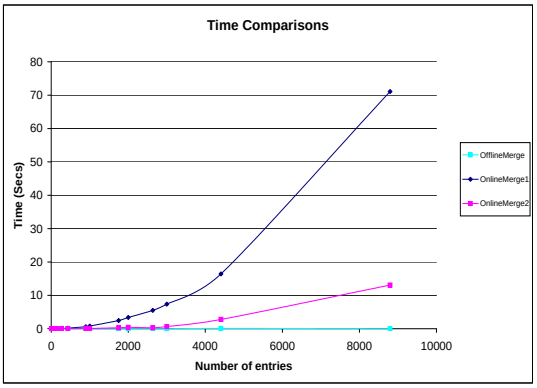

Figure 19: Running times of $OnlineMerge2$ vs. $OnlineMerge1$ & $Offline - Merge$

the test suites on this implementation. We compare it with the previous $onlineMerge2$ and offline $Merge$ implementations. Pruning the work space according to the last seen $K-1$ antichain has a big impact on the online implementation, see Figure 20, and gives promising results when it is compared to the offline $Merge$ algorithm, see Figure 21. To confirm these results, we increased our test cases. Six new suites are created. Each test suite contains 10 different test cases whose initial partition varies from 40 to 450, and size varies from 1,500 to 50,000. Test suites differ in the reducing factor, mainly width of the poset generated. The first test suite contains such test cases that width of the poset is 90% of the initial partition. The other test suites follow the same convention. We test the range of 10% to 60% reductions of the initial partitions. Each test case is executed five times, and the results are averaged. As expected, when the reducing factor is low, offline $Merge$ outperforms online $Merge$, on the other hand online $Merge$ outperforms offline $Merge$ when the reducing factor is high, see in Figure 22. However, there is no single cut off point.

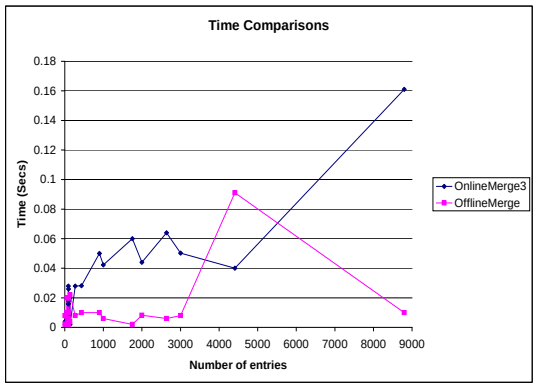

Figure 21: Running times of $OnlineMerge3$ vs. $Offline - Merge$

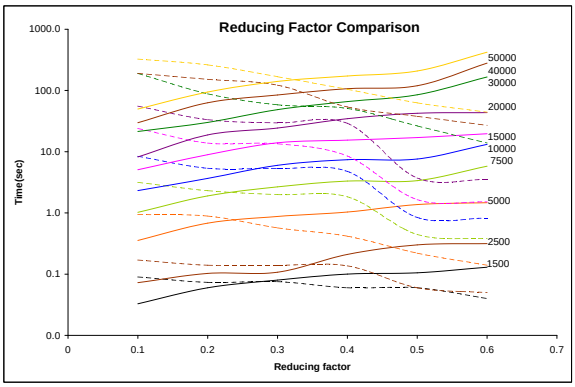

Figure 22: Running times of $OnlineMerge3$ vs. $Offline - Merge$

7 Other Relevant Applications

Partial order theory now plays an important role in many disciplines of computer science and engineering. For example, it has applications in distributed computing, concurrency theory, programming language semantics, and data mining. The theory is also useful in disciplines of mathematics such as combinatorics, number theory and group theory.

In their research paper, Ngom et al. [19] study the learning abilities of (n, k, s)-perceptrons. The (n, k, s)-perceptrons partition the input space $V \subset R^n$ into $s + 1$ regions using s parallel hyperplanes. When the output vector is not known during this learning process, the perceptron searches for a partial order relation defined over f's values, for a given function f. If such order relation exists, then a good linear extension of the corresponding partially ordered set is sought and used as the output vector of the (n, k, s)-perceptron. They propose that the selection should be done by computing the width of the currently constructed consistent poset and keeping track of the smallest width and the associated poset.

In his paper, Benczúr [5] introduces an algorithm for connectivity augmentation and poset covering for the posets that satisfy a special property by the minimal number of intervals of the poset. They argue that their algorithm can be modified for general posets as well. When modified, this is not significantly different from the one obtained by unfolding the standard reduction of Dilworth's problem to bipartite matching. Moreover, they do not give any analysis of the algorithm, other than mentioning that the steps of the algorithm could easily be checked to be polynomial in the number total possible different intervals and the length of the a longest chain in the poset [5]. They achive a polynomial bound due to the reduction in the number of edges from $O(n^2)$ to $O(n)$.

In his PhD thesis Crampton [6] indicates that there are three important access control paradigms: the Bell-LaPadula model, the protection matrix model and the role-based access control model. He points out that partial orders play a significant part these models. In the role-based access control model, width of the subposet is used to identify how many such roles can be accessed concurrently. That is the mutual exclusion problem of distributed and parallel systems.

In their drug discovery process study, Joslyn et al. [14] state that, they wish to understand the overall effect of some cell treatment or condition after some gene expression analysis experiment. In order to address this need, they view bio-ontologies more as combinatorially structured databases than facilities for logical inference. Thus, they use partially ordered sets to develop data representation for the Gene Ontology (GO). In their paper, pseudo-distances between comparable nodes are used to develop scoring functions that rank-order the GO nodes with respect to a query. One of the metrics used in these functions is defined as the width of the poset, and they state that since calculating the width of a poset is still a daunting task algorithmically, they only use it as a lower bound estimate [14].

8 Conclusion

Finding the width of a poset by an online algorithm is an important issue in distributed computing. To investigate this problem, first, we implemented and analyzed two offline algorithms. Secondly, we generalized the offline algorithms to online versions, and examined their performance. Thirdly, we present a novel online algorithm for Dilworth's chain partition under the *linear extension hypothesis* assumption. In the offline fashion, $Merge$ algorithm performs efficiently. Our first approach is to find an algorithm A that guarantees the number of calls to $Merge$ in a constant factor c of the width of poset P, then the complexity A will be $O(cwn)$. Although, such a constant factor could not found, we reduced the complexity to $O(swn)$, where s is the number of vertices between two antichains. All the algorithms for finding the Dilworth's chain partition discussed in this paper are centralized. A further research topic could be finding a decentralized online algorithm.

References

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, *"Computing a Maximum Cardinality Matching in a Bipartite Graph in time"*, Information Processing Let., 37:237-240, 1991.

[2] V. Bouchitté, R. Jégou and J.-X. Rampon, *"Online recognition of interval orders"*, In IRISA Research Report 751, 1993.

[3] V. Bouchitté, and J.-X. Rampon, *"On-line algorithms for orders"*,In Theor. Comput. Sci. 175(2):225-238, 1997.

[4] B.A. Davey, and H.A. Priestly, *"Introduction to Lattices and Order"*, Cambirage University press, Cambridge, UK, 1990.

[5] A. A. Benczúr, *"Pushdown-reduce: an algorithm for connectivity augmentation and poset covering problems"*, In Discrete Applied Mathematics, 129(2-3):233-262, Aug. 2003

[6] J. Crampton, *"Authorization and antichains"*, In PhD thesis, University of London, Birkbeck, UK, 2002.

[7] C. Fidge, *"Logical time in distributed computing systems"*, Computer, 24:28-33, Aug. 1991

[8] L.R. Ford, and D.R. Fulkerson, *"Flows in Networks "*, Princeton University Press, 1962.

[9] R. Freese, J. Jaroslav, and J. B. Nation, *"Free Lattice"*, American Mathematical Society,1996.

[10] S. Felsner, V. Raghavan, and J. Spinrad, *"Recognition algorithms for orders of small width and graphs of small Dilworth number"*, In Order, 20(4):351-364, 2004

[11] A.V. Goldberg, *"Efficient graph algorithms for sequential and parallel computers"*,In PhD thesis, Massachussetts Institute of Technology, Cambridge, Mass., Jan. 1987.

[12] A.V. Goldberg, and R.E. Tarjan, *"A new approach to the maximum-flow problem"*, In J. Assoc. Comput. Mach., 35(4):921-940, 1988.

[13] J.E. Hopcroft, and R.M. Karp, *"A $n^{5/2}$ algorithm fro maximum matchings in bipartite graphs"*, In SIAM J. Comput. 2:225-231, 1973.

[14] C. A. Joslyn, S. M. Mniszewski, A. Fulmer, and G. Heaton, *"The Gene Ontology Categorizer"*, In Bioinformatics, 20(Supplement 1):i166-i177, Aug. 2004.

[15] H.A. Kierstead, *"An effective version of Dilworth' theorem"*, In Trans. Amer. Math. Soc. 268, 1981.

[16] H.A. Kierstead, *"Recursive ordered sets"*, In Comtemporary Mathematics, 57, 1986.

[17] L. Lamport, *"Time, Clocks, and the Ordering of Events in a Distributed System"*, In Communications of the ACM(CACM),21(7):558-565, July 1978.

[18] F. Mattern, *"Virtual time and global states of distributed systems"*, In Proceedings of the International Workshop on Parallel and Distributed algorithms, pp:215-226, 1989.

[19] A. Ngom, C. Reischer, D. A. Simovici, and I. Stojmenović, *"Learning with permutably homogeneous multiple-valued multiple-threshold perceptrons"*, In Proc. 28th IEEE Int. Symp. Multiple-Valued Logic, pp:161166, May 1998.

[20] J. Setubal, *"Sequential and parallel experimental results with bipartite matching algorithms"*, Technical Report EC-96-09, Institute of Computing, University of Campinas, Brasil, 1996.

[21] A. I. Tomlinson and V. K. Garg, *"Monitoring Functions on Global States of Distributed Programs"*, Journal of Parallel and Distributed Computing, 41(2):173–189, Mar. 1997.