Efficient Incremental Optimal Chain Partition of Distributed Program Traces

Selma Ikiz and Vijay K. Garg*
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, 78712
{ikiz, garg}@ece.utexas.edu

Abstract

An important problem in distributed systems is observation of global properties of distributed computations. What makes this problem difficult is that events in the computation can be concurrent, i.e. the relation between events forms a partial order, not a total order. One of the fundamental parameters of a partial order is the width, which corresponds to the maximum number of mutually incomparable elements. For example, a process-time diagram that shows this partial order decomposition in minimum number of chains can be very useful in monitoring or debugging such computations. In this paper, we present an incremental algorithm to compute the optimal chain partition. We compare our algorithm with existing chain reduction algorithms. From a practical point of view, performance evaluation shows that our approach achieves up to 90% run-time improvement over the previously known algorithms.

`Keywords`: Mutual exclusion, testing, predicate detection, partial order, chain partition

1. Introduction

Partial orders [3] play an important role in many disciplines of computer science and engineering such as distributed computing, concurrency theory, programming language semantics, and data mining. In this paper, we focus on applications in distributed computing. A distributed computation is generally modeled as a partially ordered set (poset) of events based on the *happened before* relation as defined by Lamport [14]. Fidge [6] and Mattern [15] independently introduced vector clocks to timestamp events such that the happened-before relationship between any two events can be determined by examining their timestamps.

A set of events with vector clocks implicitly defines a poset where the order of events is given by the vector clock order. To solve many problems in distributed systems, we need to determine properties of such a computation poset. For example, suppose that we are interested in determining whether a conjunctive predicate $(l_1 \wedge l_2 \wedge \ldots \wedge l_N)$ became true in a computation where l_i is a local predicate on process i [10]. Then, determining the least set containing N mutually incomparable elements is equivalent to determining the least global state in which the global predicate is true. As another example, let $token_i$ denote that process i has the token. Assume that we are interested in any violation of the assertion that "there are at most K tokens in the system." This assertion is violated if and only if the poset (of token related events) has width $K+1$ or more. In particular, if K equals to 1, then violation of the assertion is equivalent to mutual exclusion violation [16]. The violation of K-mutual exclusion problem can be detected by computing the width of a computation poset. Figure 1 shows an example for detection of 1-mutual exclusion violation.

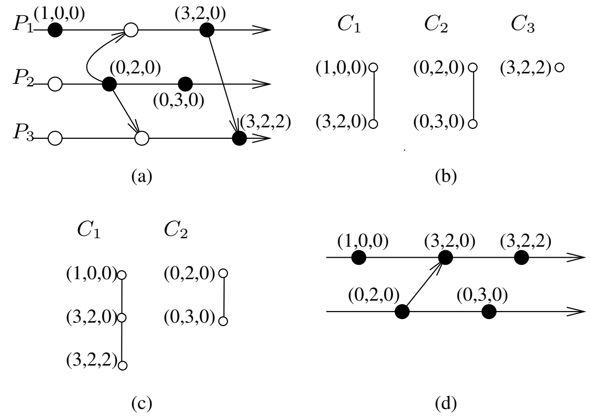

Figure 1. (a) A computation $<E, \rightarrow>$ (b) the trace as chain partition (c) optimal chain partition (d) the corresponding partial order trace.

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, CNS-0509024, Texas Education Board Grant ARP-320, and an Engineering Foundation Fellowship.

Determining properties of the underlying partial order of events is also useful in visualization of a distributed computation. Visualization tools for program traces significantly facilitate the efforts of debugging parallel and distributed programs. These tools provide a process-time diagram of the computation. There are two approaches to place the events in this diagram: the real-time of event occurrence [9] and the partial order induced by the vector clocks [13]. For example, POET is a tool for the visualization of partially ordered event traces [13]. Because process-time diagrams may be quite large and complicated, users may be interested in visualizing only a subset of events called interesting events. A visualization that partitions the set of interesting events into smallest number of chains provides user a much more simplified view than a random drawing of the poset. Thus, our algorithm has applications also in visualizing a set of events in a distributed computation.

In partial order theory, the optimal chain partition of a poset is known as the *Dilworth's chain partition (DCP)* [3]. Motivated by the importance of partial orders in distributed systems, we investigate the problem of computing the width of a poset and its Dilworth's chain partition in an incremental manner. We introduce a novel incremental 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 *relevant elements*. We define the relevant elements later in section 4. In the worst case s is n giving our algorithm complexity of $O(wn^2)$. Table 1 shows a comparison of our work with existing algorithms.

Table 1. Comparison with related work (N is the initial partition, n is the poset size, w is the poset width, $e_{\leq}$ is the number of edges in the poset and s is the maximum number of relevant elements)**.**

	Additional Space	Time Complexity
Bogart and Magagnosc [8]	$O(e_{\leq})$	$O((N-w)(n+e_{\leq}))$
Tomlinson and Garg [16]	$O(n)$	$O((N-w)Nn)$
This paper	$O(n)$	$O(wsn)$

In summary, this paper makes the following contributions.

1. We propose a new algorithm to compute optimal decomposition of a poset specified using vector clock representation. Our algorithm is incremental and requires less memory and time than previously known algorithms.
2. We show results of an experimental study that compares various algorithms for memory and time usage for optimal decomposition of posets.

The rest of the paper is organized as follows: In section 2, basic background definitions in partial order theory and its relation to distributed computing are given. In section 3, we compare two algorithms on finding the optimal chain partition of a poset. First algorithm is given by Bogart and Magagnosc [8], and hereafter we refer to it as BM. Second algorithm is given by Tomlinson and Garg [16], and hereafter is referred as TG. We also show how these algorithms can be used under incremental and offline assumptions, and discuss their drawbacks compared to each other. In section 4, a novel incremental algorithm CP (chain partitioner) that improves on the previous ones is introduced. The simulation results of the algorithms are presented in section 5. We also give the details of implementations and testing environment.

2. Background

2.1. Partially Ordered Sets and Lattices

A *partially ordered set* (or poset, or partial order) is a pair (X, P) where X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. A $subposet$ is a poset whose set is a subset of X, and whose relation is restriction of P to the subset. 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 an *antichain* (*chain* or a *linear order*), if every distinct pair of points from X is incomparable (comparable) in P. Moreover, we call P a k $antichain$ if the size of X is k. An antichain (chain) C of a poset (X, P) is a *maximum antichain (chain)* if no other antichain (chain) contains more elements than C. The $width$ of a poset is defined to be the largest antichain in the poset and is denoted by $width(P)$, and hereafter referred to as w. In the context of a distributed computation, the width must be less than or equal to the number of processes.

It is possible to extend any partial order to a linear order by adding order between incomparable elements. A total order L is said to be a *linear extension* or *linearization* of P, if L is a total order on the same ground set X of P, such that for all $u, v \in X$ for which $u \leq_P v$ implies $u \leq_L v$.

An element $x \in X$ is called a *maximal* (*minimal*) element if there is no element $y \in X$ with $x < y \in P$ $(x > y \in P)$.

A *chain partition* of a poset P is a partition of the ground set into (nonempty) chains. A famous theorem of R. P. Dil-

worth [3] 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. We call a partition of a poset P Dilworth's chain partition or DCP for short, if P is partitioned into w chains.

2.2. System Model

We presume the standard model of distributed systems initially defined by Lamport [14]. The system consists of multiple sequential processes (or threads) communicating via message passing. Each process executes a sequence of events. Each event is an internal, a send or a receive event. A *distributed computation* is the union of all of the events across all of the processes. In the happened before model it is defined as a tuple $(E, \rightarrow)$ where E is the set of events and $\rightarrow$ is a partial order on events in E. For an event $e \in E$, $e.p$ denotes the process on which e occurred.

The happened before relation $(\rightarrow)$ is the smallest transitive relation which satisfies:

- $e \rightarrow f$ if $e.p = f.p$ and e immediately precedes f in the sequence of events on process $e.p$.
- $e \rightarrow f$ if e is a send event and f is the corresponding receive event.

The happened before relation defines a partial order over the set of events in the distributed computation. The events for which the happened-before order needs to be determined are called *interesting events* and the set of such events are denoted by $I \subseteq E$.

3. Previous Algorithms

There are two principal approaches for finding a DCP of a poset P:

1. Partitioning the poset greedily into N initial chains and reducing the number of partitions by finding *reducing sequences* [8], or
2. Reducing the problem to a maximum matching problem in a bipartite graph and then using the results of the maximum matching problem [7].

For an execution trace of a distributed program, the first approach is more appealing since the history of execution can be recorded easily as a chain on each process. Hence, we do not consider the bipartite approach in this paper.
The algorithms by Bogart and Magagnosc [8] and Tomlinson and Garg [16] are based on the first approach. Given a chain partition of size k, they answer the question whether the partition could be reduced to $k - 1$ chains. The 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. We give a brief description of these algorithms in the next subsection. For a more detailed discussion of the algorithms, refer to [8] and [16].

3.1. Algorithm BM

Bogart and Magagnosc [8] solve the chain reduction 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 $O(n + e_{\leq})$, a lookup table is constructed. This table is essentially the adjacency list representation of a poset. With this setup, BM is called. Let C be a chain partition, then it either finds a reducing sequence, forms new chains, and returns the reduced C; or returns the original chain partition C.

Definition 1 [8] *A sequence of elements $a_0;b_0;a_1;b_1;\cdots;a_s;b_s$ is a reducing sequence if*

(a) a_0 is the least element of some chain,

(b) b_i is the immediate predecessor of a_{i+1} in some chain,

(c) all b_i's are distinct,

(d) for all i: $a_i > b_i$ in the partial order,

(e) b_s is the greatest element of its chain.

BM, as a first step, uses a breadth first search to find a reducing sequence. In the second step it modifies the chains in C according to the reducing sequence. An example is given in figure 2 to illustrate how this chain manipulation works according to a *reducing sequence*.

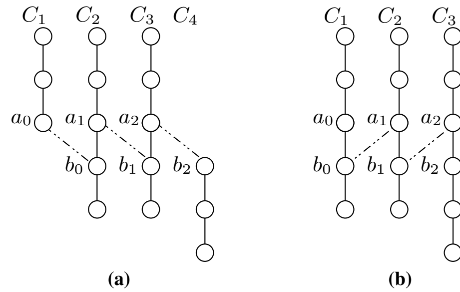

Figure 2. An example for BM: (a) a reducing sequence (b) chain reduction

The complexity of the first and second steps are $O(n + e_{\leq})$ and $O(n)$ [8], where n is the poset size and $e_{\leq}$ is the number of edges in the poset.

3.2. Algorithm TG

Tomlinson and Garg [16] solve the chain reduction problem using additional chains and a spanning tree data structure for the *reducibility condition*. They begin with a chain partition C that contains k chains called $input$ chains and $k-1$ empty chains called $output$ chains C'. With this setup, TG is invoked. It either merges k chains to $k - 1$ chains and returns C', or returns an antichain of size k. At each step, TG considers only the chains' heads. It repeatedly selects chain heads that satisfy the reducibility condition for

removal, and appends them to an output chain. This continues until either one of the input chains is empty or no such element can be found.

Definition 2 *Let a_i be the least element (head) of input chain i, and b_i be the maximum element (tail) of output chain i. Then, a chain head a_s satisfies the reducibility condition if*

(a) for all i: there exists j and k s.t. $b_i < a_j$ and $b_i < a_k$, where $j \neq k$,

(b) for all i: there exists j s.t. $b_j < a_i$,

(c) there exists i st. $a_s < a_i$, where $i \neq s$.

A spanning tree is formed by the input and the output chains. It is used in the decision process of where (which output chain) to append the elements. Input chains are represented as nodes, while edges are represented as output chains. The algorithm maintains the following invariant: If an edge between vertices C_i and C_j is labeled as C'_k, then the heads of C_i and C_j are bigger than the tail of C'_k. Hence, item b in the reducibilty condition is always true (each output chain tail is less than at least two of the input chain heads). Intuitively, the reducibility condition says that chain head a_s is selected for removal if it is less than at least one of the input chain heads.

An example is given in figure 3 to show how the state of the tree and of the chains are modified at each step. Figure 3a shows the initial setup; C shows the input chains, C' shows the empty output chains, and dashed lines shows the spanning tree. At step 1, chain head (1,0,0) is selected for removal since (1,0,0) is less then (2,0,0). Figure 3b shows the resultant chains and tree. Chain head (0,1,0) is selected for removal at the second step, while (2,0,0) and (2,2,0) are selected at the third step. At the fourth step chain heads (1,2,0) and (2,2,0) are selected for removal. Since the first chain is empty, TG places all the elements in the input chains to the output chains. Figure 3d shows the final output chains C'.

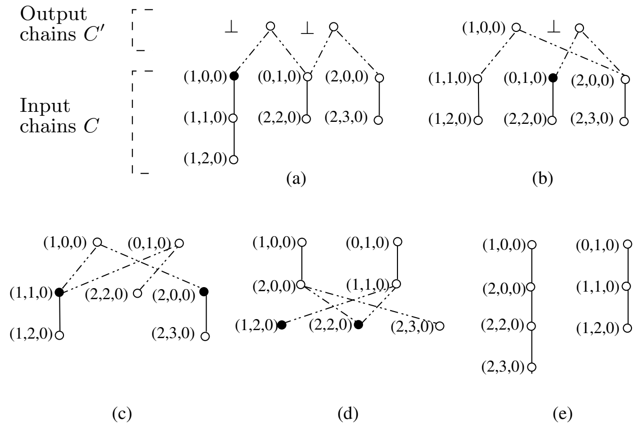

Figure 3. An example for TG

3.3. Computing DCP in Offline Manner

The execution trace of a distributed program can be recorded easily as a chain on each process. At the termination of the execution, N chains are obtained. Such a trace corresponds to the trivial partition of a computation poset. We can now define the offline version of the DCP problem as follows:

Definition 3 *Let P be a partially ordered set with n elements. Given a chain partition of P, $C = \{C_1, \ldots, C_N\}$ into N disjoint chains, rearrange these chains into a chain partition with the fewest number of chains.*

The complexity of an algorithm that computes DCP in an offline manner by using BM or TG could be computed as follows: Let the initial partition be given in N disjoint partitions. Subsequently, $N - w$ BM or TG calls need to be made to obtain the DCP. Hence, the total complexity of BM is $O((N - w)(n + e_{\leq}))$, where the total complexity of TG is $O((N - w)Nn)$. The worst case complexities of both algorithms are the same, $O(n^3)$.

3.4. Computing DCP in Online Manner

Online or incremental algorithms work on dynamic structures with unpredictable behavior. Informally, in an incremental approach, at each step some new event is added to the poset and we want to compute DCP of the new poset. There are two frameworks for general incremental algorithms. The first approach assumes that a value computed at a step is never updated in a latter step. This approach is useful for applications where a previous decision cannot be changed, e.g. scheduling or motion control. For this approach, the best known online algorithm for partitioning the poset is given by Kierstead [12]. His algorithm to partition any poset of width k requires $\frac{(5^k-1)}{4}$ chains. Felsner [4] and Agarwal et al. [1] have presented online algorithm for some special cases that require $\binom{k+1}{2}$ chains to partition a poset.

The second approach for incremental DCP allows the chains to be reorganized to reach the optimal solution. In this paper, we are concerned with the second approach for finding the width of the known subposet $P' \subseteq P$, and partitioning it into w' chains. At each step, we have the accurate width of the subposet and a corresponding chain partition. However, when a new element arrives the chain partition may get reorganized.

Here, we assume that a new element arrives adhering *linear extension hypothesis* -elements arrive consistent with to a linearization of the poset- defined by Bouchitté et al. [2]. Thus, when a new element arrives, all elements that happened-before it have already arrived and been processed. This is achieved by buffering the new element if it violates the assumption and processing it later. Whether to buffer an element can be determined efficiently by examining its vector clock. We now define the incremental version of the DCP problem as follows:

Definition 4 *Let P be a partially ordered set with n elements and C_{t-1} be the chain partition at time $t-1$. Given a linearization L of P, and v_t as the tth element of L, compute the optimum chain partition C_t of $C_{t-1} \cup v_t$.*

We now outline an algorithm that uses BM or TG for computing the DCP in an incremental manner. Given BM and TG, one simple way to use them in an incremental fashion is calling them whenever a new element arrives. Let C^j be the chain partition of size k at step j, and e_j be the new element. Then, we create a new chain C^j_{k+1} that contains only e_j and append it to the chain partition C^j. Later, we call BM or TG with the modified chain partition. Then, the chain partition returned by BM or TG becomes the current chain partition for step $j{+}1$. If there were no reduction, then the current width of the subposet is increased. We refer to these algorithms as BM-Incremental and TG-Incremental. Figure 4 shows the incremental algorithm.

```
procedure Incremental
        (v:vector clock, Alg:algorithm): C:chain partition
  assume: 1 ≤ k ≤ |w'|
     C =Alg(C, v);
     return C;
endprocedure
```

Figure 4. Incremental algorithm

We now analyze the time complexities of the algorithm. Let the width of the current subposet be k. Then running BM and TG once have time complexity of $O(j + e_{\leq})$ and $O(kj)$, respectively. Since there are n steps, and $k \leq w$, the total complexity of the incremental algorithm is $O(n^3)$ when BM is used, and $O(wn^2)$ when TG is used. Felsner et al. [4] [5] argue that $O(Kn^2)$ is optimal for any $K > 0$ even for the decision problem: Whether P can be partitioned into K chains.

4. Algorithm IG

In this section, we present a new incremental algorithm CP to compute DCP of a poset. Our algorithm can be used both in incremental and offline manner. We only give the incremental version, since offline version is trivial.

The algorithm uses chains to store the poset elements. Each chain is stored in an increasing order so that head is the smallest element and tail is the largest element in the chain. CP keeps two types of chains: work and history. We refer to the set of work (history) chains as work (history) space. W^j (H^j) represents the work (history) space, while W^j_i (H^j_i) represents the ith chain of the work space (history space) at time j. On arrival of a new element at time j, CP returns the width and the partition of the subposet P^j.

Whenever a new element arrives, CP tries to append it to one of the chain tails. However, this might not be always possible. In this case, CP calls the $Merge$ function to merge k chains into $k-1$ if possible. The $Merge$ function takes k chains as input. It returns three types of chain sets; input chains C=$C_1, \cdots, C_k$, output chains C'=$C'_1, \cdots, C'_{k-1}$, and history chains H'= $H'_1, \cdots, H'_{k-1}$.

$Merge$ uses the following operations on chains (q and p represent chainsS):

$append(p, e)$	: append element e to chain p
$removehead(p)$	: remove the head of the chain p
$head(p)$	: return the first item in p, or a maximal value if p is empty
$append(q, p)$	: append p content to the end of q, and empty p

The $Merge$ function borrows the basic merge idea, and the $FindQ$ function along with the spanning tree data structure from [16]. Merge is performed by repeatedly removing an element from one of the k input chains and inserting it in one of the $k-1$ output chains. The output chain to place the element is decided by $FindQ$ according to the spanning tree data structure. At each step, $Merge$ only compares the heads of chains which have not been compared earlier. It keeps track of this in the variable ac which is the set of indices indicating those input chains whose heads are known to form an antichain. It terminates when either ac has k elements or one of the input chains is empty. Whenever ac has k or $k-1$ elements, the $Merge$ function appends the content of output chains to history chains and clears the output chains. A detailed discussion of spanning tree data structure, and $FindQ$ can be found in [11].

CP updates its work and history chains according to the result of $Merge$. It uses the following operations on work and history chains (p and q represent chains, P and Q represent sets of chains):

$tail(p)$	: return the last item in p, or a minimal value if p is empty
$insert(P, e)$	: insert element e in P as a new chain
$join(Q, P)$	: append each p in P to a q in Q if possible, or add p to Q as a new chain

CP is shown in figure 5. It updates its work space according to the output chains of $Merge$, if $Merge$ reduces the chain partition (see line 7 in figure 5). Otherwise it uses the input chains of $Merge$ to update its work space (see line 9 in figure 5). The history space of CP is updated by the history chains of $Merge$ (see line 11 in figure 5).

CP splits the the current chain partition into work and history space, and continues the merge operation on new elements by only comparing them to the elements in the work space. Once an element is placed in the history space, it never appears in the work space in a later step. Besides, the partition of the history space never changes. This splitting reduces the number of comparisons in the $Merge$ function, since at step j we do not need to compare the new incoming

```
procedure CP (v:vector clock):W, H:chain sets
 assume: 1 ≤ K ≤ w^j (w^j is the width of the execution seen)
1:    if ∃i : tail(W_i) ≤ v then
2:         append(W_i, v);  //append at a chain tail
3:    else
4:         insert(W, v);
5:         (C,C',H')=Merge(W);
6:         if C = ∅ then  //no width increase
7:              W = C'
8:         else  //width increase
9:              W = C;
10:        endif
11:             join(H, H');
12:   endif
13:   return W, H;
endprocedure
```

Figure 5. Algorithm CP

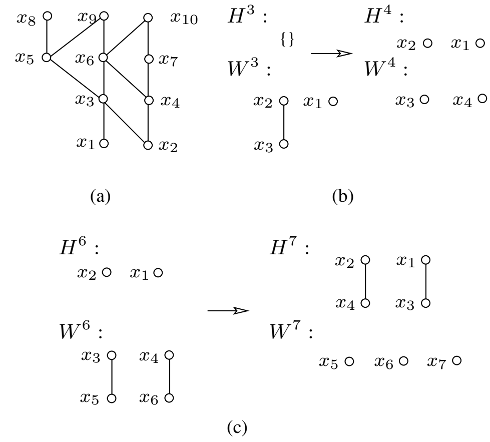

Figure 6. (a) A poset P (b) transition in steps 3 & 4 (c) transition in steps 6 & 7

element with the elements that are properly below the maximum w^j antichain, M^j. It is sufficient to compare it with M^j and the elements that are above it. The intuition behind this optimization is that when elements arrive according to linear extension hypothesis, the new element is either greater than an element in M^j, or forms a new antichain with M^j. We define the elements that are properly below this antichain as *irrelevant elements* for the later steps.

We now give the definition of a relevant element.

Definition 5 *Let M^j be the biggest maximum antichain of P^j. Then, an element $e \in P^j$ is relevant at time j if there exist an $f \in M^j$ s.t. $f < e$.*

Observe that once an element is labeled as irrelevant element, it is never relabeled. Moreover, after a $Merge$ call, work space includes all the relevant elements while, history space includes all the irrelevant elements.

When CP calls the $Merge$ function, the width of the chain partition either increases or stays the same. We know give an example how CP updates its work and history space according to two of the cases. The details of the $Merge$ could be found at [11].

4.1. Example

Consider the example in figure 6. Assume that the execution happened is x_2, x_1, x_3, x_4, x_5, x_6, x_7, x_8, x_9,x_{10}.

Case 1: There is no increase in width with new element

At time 4, the execution seen so far is $X^3 = \{x_2, x_1, x_3\}$ and the new element is x_4. The width of the subposet is two. The content of work space is as follows: C_1 contains x_2 and x_3, and C_2 contains x_1. There is no chain to append x_4 since both x_1 and x_3 are incomparable with x_4, hence CP creates a new chain C_3 and appends x_4 to C_3, and invokes $Merge$. Observe that k is three. $Merge$ places x_2 in C'_1 and x_1 in C'_2, and computes ac as $\{x_3, x_4\}$. Since ac has two elements, it appends the contents of C' to H'. It places x_3 in C'_1, x_4 in C'_2, and returns C (as empty), C', and H' to CP. Then, CP assigns the output chains C' as work chains, and appends the history chains H' of $Merge$ to its history space. Figure 6b shows the transition between time 3 and 4.

Case 2: The width increases with the new element

At time 6, the execution seen so far is $X^6 = \{x_2, x_1, x_3, x_4, x_5, x_6\}$ and the new element is x_7. The width of the subposet is two. The content of work space is as follows: C_1 contains x_3 and x_5, and C_2 contains x_4 and x_6. There is no chain to append x_7 since both x_5 and x_6 are incomparable with x_7, hence CP creates a new chain C_3 and appends x_7 to C_3, and makes a $Merge$ call. $Merge$ places x_3 in C'_1 and x_4 in C'_2, and computes ac as $\{x_5, x_6, x_7\}$. Since ac has three elements -an antichain of size three, hence no reduction is possible-, it appends the contents of C' to H', and returns C, C' (as empty), and H'. Then, CP assigns the input chains C as work chains, and appends the history chains H' of $Merge$ to its history space. Figure 6c shows the transition between time 6 and 7. The discussion of the algorithm correctness can be found in [11].

5. Experiments

In this section we present our simulation results for both incremental 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 five times and the average is used in the analysis.

5.1. Previous Algorithms

In the implementation of *BM* and TG, chains are represented as vectors, and 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 efficiency of *BM* .

We set up three different test suites such that each includes twelve simple test cases. First test suite includes several small width posets that have at most 90 elements. Second test suite includes posets of size 100 while width of the poset changes from 2 to 13. Third test suite includes a fixed size initial partition, while the number of elements changes from 10 to 10,000. The posets are randomly created according to a given poset size.

In offline manner, TG outperforms BM both in memory consumption (see figure 7a), and running times (see figure 7b). Moreover, BM gives *out of memory* error when the number of elements is bigger than 9,000. This is, in fact, expected since constructing the lookup table is done in $O(n^2)$ and needs space $O(n^2)$.

In online manner, we introduce one element at a time in agreement with linear extension hypothesis.

It is trivial to observe that memory usage of each step of incremental versions never exceeds memory usage of the offline algorithms. Therefore, hereafter we do not compare memory usage. We observe that the running time of incremental TG is at least twice as good compared to BM (see figure 7b).

Another worthwhile comparison is the running times of incremental and offline versions of these algorithms (see figure 7b). Although TG performs better than BM in incremental fashion, its performance is still far behind the offline versions.

5.2. Algorithm CP

We have implemented and tested CP using the identical objects and test suites mentioned in the previous section. When a new element arrives, we have observed that $Merge$ is called at most 22% of the time.

In this section, we compare our incremental algorithm CP with the offline TG since it has the best performance so far. Pruning the work space has a big effect on the performance of incremental algorithm, and gives promising results when compared to the offline TG, see figure 7c.

To confirm these results, we increase our test cases. Seven new test suites are created. Each test suite contains 18 different test cases whose initial partition vary from 10 to 450, and size vary from 100 to 70,000. Posets are generated randomly according to a given poset width and size. We fix the vectorclock size as 10. The random poset generator takes the vectorclok size, width and size of the poset, and its initial partition size as parameters. It creates a poset of the given width and size initially, then partitions it greedily into the given partition size. Let N and w be respectively the size of the initial partition and poset width. We define *reducing factor* as $(N - w)/N$. Test suites differ in reducing factor. The first test suite contains test cases that DCP of the poset is equal to the initial partition. The second test suite contains test cases that the width of the poset is 90%

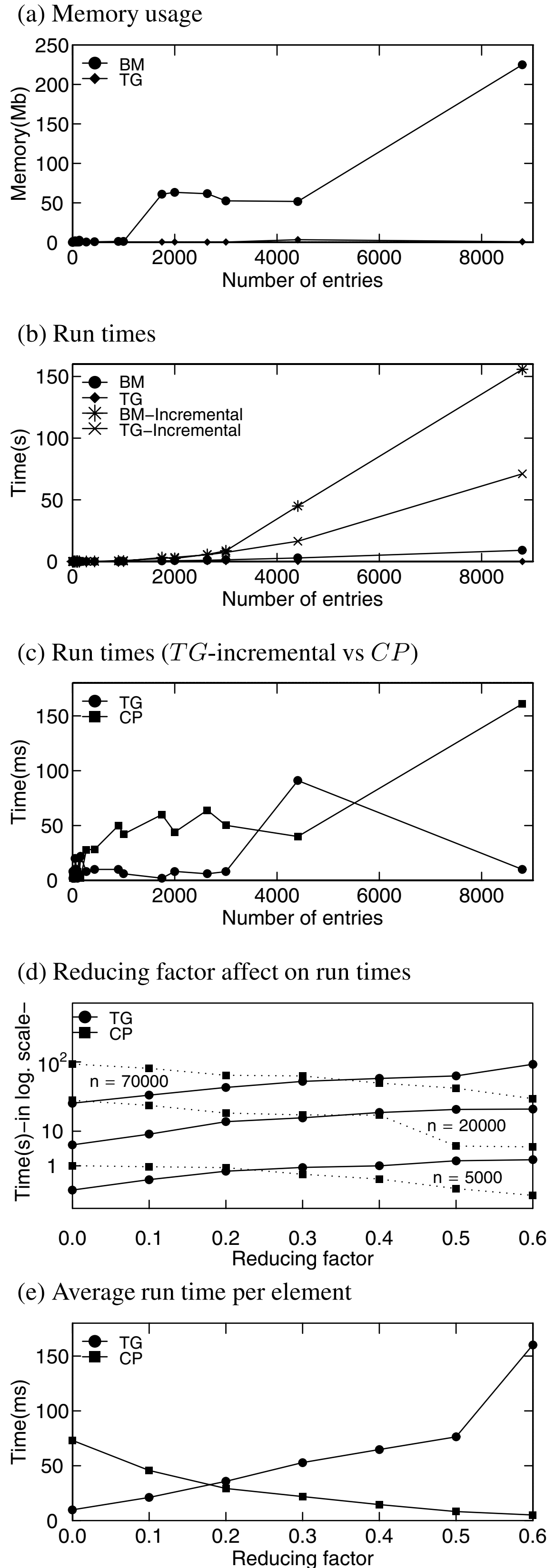

Figure 7. Simulation results

of the initial partition, hence reducing factor is 0.1. Other test suites follow the same convention. We test the range of 0% to 60% reductions of initial size. Figure 7d shows the running times of CP and TG for three test cases (note that the scale of time axis is logarithmic), detailed graphs for all test cases can be found in [11].

As expected, when the reducing factor is low, offline algorithm outperforms incremental algorithm. On the other hand, incremental outperforms offline when the reducing factor is high. It appears that there is no fixed cut-off point to choose between incremental and offline algorithm. When the reducing factor is 0.3, offline algorithm performs better for the second test, while incremental algorithm performs better for the first test case. However, when the reducing factor is more than 0.5, incremental algorithm computes the optimal chain partition efficiently.

To simplify the results, for each test case the average time spent for an element is calculated by dividing the running time by the number of elements in the test case. Then, we have averaged them according to their reducing factor. CP gives encouraging results when compared to TG, see figure 7e. TG computes at best 8 times faster than CP when the reducing factor is 0, while CP computes 30 times faster than TG when reducing factor is 0.6.

6. Conclusion

The ability to compute the optimal chain partition of a distributed computation is useful for debugging, testing and analyzing distributed programs. For example, we can determine if there is a potential violation of a limited resource by monitoring an execution of a distributed program.

In this paper, we have implemented and analyzed two offline algorithms by Bogart and Magagnosc [8], and Tomlinson and Garg [16] to investigate this problem. Then, we generalize the offline algorithms to incremental versions. However, our experiments have shown that these algorithms do not perform well under the incremental assumption.

Finally, we have developed a novel incremental algorithm for computing the optimal chain partition and the width of a poset under the linear extension hypothesis. The main idea of this algorithm is that we prune the work space and reduce the number of elements to those most essential that we call relevant elements. Thus, the number of relevant elements bound the number of comparisons at each step and reduce the complexity to $O(wsn)$, where s is the maximum number of relevant elements, w the width, and n the size of the poset.

A further research topic could be developing a decentralized incremental algorithm for computing the optimal chain partition of distributed program traces. Also, it would be interesting to see the application of the algorithm in a visualization tool.

References

[1] A. Agarwal and V. K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 19–28, New York, NY, USA, 2005. ACM Press.

[2] V. Bouchitté, , and J.-X. Rampon. On-line algorithms for orders. In *Ordal'94: Selected papers from the conference on Orders, algorithms and applications*, pages 225–238. Elsevier Science Publishers B. V., 1997.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[4] S. Felsner. On-line chain partitions of orders. *Theoretical Computer Science*, 175:283–292, 1997.

[5] S. Felsner, V. Raghavan, and J. Spinrad. Recognition algorithms for orders of small width and graphs of small Dilworth number. *Order*, 20(4).

[6] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

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

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

[9] M. Frumkin, R. Hood, and L. Lopez. Trace-driven debugging of message passing programs. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium*, page 753. IEEE Computer Society, 1998.

[10] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1323–1333, 1996.

[11] S. Ikiz and V. Garg. Efficient online optimal chain partition of distributed program traces. Technical Report TR-PDS-2005-001, University of Texas at Austin. Available as `"http://maple.ece.utexas.edu/TechReports/2005/TR-PDS-2005-001.ps"`.

[12] H. Kierstead. Recursive colorings of highly recursive graphs. *Canad. J. Math*, 33(6):1279–1290, 1981.

[13] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system-independent visualizations of complex distributed-application executions. *The Computer Journal*, 40.

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

[15] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al, editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[16] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *J. Parallel Distrib. Comput.*, 41(2):173–189, 1997.