An Online Algorithm for Dimension-Bound Analysis

Paul A.S. Ward

Shoshin Distributed Systems Group
University of Waterloo, Waterloo Ontario N2L 3G1, Canada

Abstract. The vector-clock size necessary to characterize causality in a distributed computation is bounded by the dimension of the partial order induced by that computation. In theory the dimension can be as large as the number of processes in the computation, but in practice it is much smaller. We present an online algorithm to compute the dimension of a distributed computation. This algorithm requires the computation of the critical pairs of the partial order followed by the creation of extensions of the partial order. This is our next step toward the goal of creating an online vector clock whose size is bounded by the dimension of a distributed computation, not by the number of processes.

1 Motivation

An important problem in distributed systems is monitoring and debugging distributed computations. What makes this problem hard is that events in the computation can be concurrent. The events form a partial order, not a total one. While a process-time diagram which shows this partial order of execution can be useful, it is not possible to display the whole partial order for any non-trivial computation. As a result, distributed debugging systems such as POET [8] need to provide much more than just a drawing. It is necessary to intelligently scroll around the display [14], search for interesting patterns [6], compute differences between subsequent executions of a computation [4], detect race conditions [18], determine appropriate abstractions to provide higher level views [7], and so on. To perform these operations it is frequently necessary to determine event precedence. That is, given two events, it is necessary to be able to efficiently determine if they are ordered or if they are concurrent.

There are several ways in which event precedence may be determined, depending on how the partial order is represented. If we store the partial order as a directed acyclic graph, then precedence determination is a constant time operation. This is because the partial order is transitively closed and so there is an edge between any two events that are ordered. However, the space consumption for this method is unacceptably high. Instead we store the transitive reduction of the partial order. This is space-efficient but requires a (potentially quite slow) search operation on the graph to determine precedence. To compensate for this deficiency a vector clock [2, 10] is associated with each event. If the processes in which the events occur are known, then it is possible to determine precedence with vector clocks in constant time.

The size of a vector clock necessary to capture causality is bounded by the dimension of the partial order induced by the computation. Charron-Bost [1] has shown that the dimension can be as large as the width, and all online vector clocks developed to

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 144–153, 1999.

date require a vector with size equal to the number of processes (which forms an upper bound on the width). Since we need to associate such a vector clock with every event in the computation, we are substantially constrained in the number of processes that we can observe. In POET we have found that due to this limitation we can handle at most a few-hundred processes. However, we have already shown [16] that in practice the dimension is much smaller than the number of processes. Our ultimate goal is therefore to develop a vector clock whose size is bounded by dimension, not width, and that vector clock must be implementable in an online algorithm.[1] This paper is a first step in that direction, by developing an online dimension-bound algorithm.

In the remainder of this paper we will specify first the formal model of distributed computation and the terminology necessary to deal with dynamic partial orders. In Sect.3 we will describe the supporting theorems and the online algorithm. We then provide some analysis of the algorithm. Finally we indicate what work remains to be completed to achieve online dimension-bounded vector clock.

2 Background

We use the standard model of distributed systems, initially defined by Lamport [9]: a distributed system is a system comprising multiple sequential processes communicating via message passing. Each sequential process consists of three types of events, send, receive and unary, totally ordered within the process. A *distributed computation* is the partial order formed by the "happened before" relation over the union of all of the events across all of the processes. We will refer to the set of all events with E, the set of events within a given process by E_i (where i uniquely identifies the process) and an individual event by e_i^j (where i identifies the process and j identifies the event's position within the process). Then the Lamport "happened before" relation ($\rightarrow \subseteq E \times E$) is defined as the smallest transitive relation satisfying

1. $e_i^j \rightarrow e_i^l$ if $j < l$
2. $e_i^j \rightarrow e_k^l$ if e_i^j is a send event and e_k^l is the corresponding receive event

Events are concurrent if they are not in the "happened before" relation:

$$e_i^j \parallel e_k^l \iff e_i^j \not\rightarrow e_k^l \wedge e_k^l \not\rightarrow e_i^j \tag{1}$$

These definitions presume that the partial order is a static entity. That is, implicit in the "happened before" and "concurrency" relations is a fixed computation to which they refer. In an online monitoring or debugging context, the computation, and hence the partial order, is not fixed. Therefore, we now describe the terminology we use in this paper for dealing with partial orders and changing partial orders.

2.1 Partial-Order Terminology

The basic partial-order terminology is due to Trotter [15], modified as necessary to take into account the dynamic nature of the partial orders that we are dealing with. A *strict*

[1] An offline algorithm, the Ore timestamp [11], already exists that is dimension-bounded, not process-bounded.

partial-order (or partially-ordered set, or poset) is a pair (X, P) where X is a finite set[2] and P is an irreflexive,[3] antisymmetric and transitive binary relation on X. An *extension*, (X, Q), of a partial order (X, P) is any partial order that satisfies

$$(x, y) \in P \Rightarrow (x, y) \in Q \tag{2}$$

If Q is a total order, then the extension is called a *linear extension*. A *realizer* of a partial order is any set of linear extensions whose intersection forms the partial order. The *dimension* of a partial order is the cardinality of the smallest possible realizer.

To deal with dynamic partial orders we will develop theorems regarding the relationship between the partial order before and after some element is added to the base set (with resulting changes to the binary relation of the partial order). In other words, the theorems we develop will relate two distinct partial orders. It is therefore necessary to make explicit which partial order the precedence ($\rightarrow$) and concurrency ($\|$) relations refer to. We will do so by subscripting those relations with the base set of the partial order on which the relations hold. We will likewise subscript any other relations or sets we define for partial orders to indicate explicitly the partial order to which the relation or set refers. Subscripts may be omitted if there is only a single partial order under discussion, or if the relation or set is identical for all partial orders in the theorem in question.

We can now continue the discussion of realizers and dimension by recalling the definition of a critical pair.

Definition 1 (critical pair). *(x, y) is a critical pair of partial order (X, P) if $x \parallel_X y$ and $(X, P \cup \{(x, y)\})$ is a partial order.*

Equivalently

$$(x, y) \in \mathrm{CP}_X \iff (x \parallel_X y) \wedge \forall_{z \in X} (z \rightarrow_X x \Rightarrow z \rightarrow_X y) \wedge (y \rightarrow_X z \Rightarrow x \rightarrow_X z) \tag{3}$$

where CP_X is the set of all critical pairs of the partial order (X, P). The significance of critical pairs, as regards dimension, is in the following theorem [15]

Theorem 1. *The dimension of a partial order is the cardinality of the smallest possible set of extensions that reverses all of the critical pairs of the partial order.*

A critical pair (x, y) is said to be reversed by a set of extensions if one of the extensions in the set contains $y \rightarrow x$. Note that the extensions need not be linear. Thus the approach we have taken for our algorithm is to incrementally create a set of extensions to the partial order that reverses the critical pairs of the partial order.

Finally, We say that "y is covered by z" or "z covers y" if there is no intermediate element in the partial order between y and z.

$$y <: z \iff y \rightarrow z \wedge (\nexists_w y \rightarrow w \wedge w \rightarrow z) \tag{4}$$

Before proceeding to describe the algorithm, we will briefly describe some of the work related to our own.

[2] Since we are modeling distributed computations, all of the sets will be finite.

[3] If P is reflexive, it is a *partial order* rather than a *strict partial order*. The difference is not an important point in our context. The "happened before" relation, as defined by Lamport [9], is irreflexive, and so we use strict partial orders.

2.2 Related Work

There are essentially two categories of related work. The first group are those who are developing systems for visualizing parallel and distributed systems in a process-time fashion. Such work includes GOLD [12], ParaGraph [5] and our own system, POET [8]. Other than our own, these systems tend to use vector-clocks or variants within the computation, rather than have the information sent to a central server which computes the vector clocks for visualization purposes. These systems tend to take the approach of just using what is presently available, and not being concerned with substantial scalability. The GOLD system uses dependency vectors, developed by Fowler and Zwaenepoel [3], which will be size $O(n)$ by the time they are attached to individual events. ParaGraph has the ability to provide space-time diagrams, but there is no attempt to determine causality. It is merely a visualization based on, possibly badly synchronized, local clocks. It is up to the user to trace the dependencies. Indeed, the authors acknowledge that the size would not scale well beyond 128 processes, as the display becomes too cluttered. POET uses standard Fidge/Mattern vector clocks [2, 10], and so it too requires vectors with size equal to the number of processes.

The second group are those who are trying to reduce the vector-clock size, but in the context of maintaining vector clocks by the processes involved in the computation. The primary work in this area is a technique by Singhal and Kshemkalyani [13]. The problem with the technique offered is that an $O(n^2)$ matrix is required at each process to recover the vector. In the context of a computation, this may be acceptable. In the context of monitoring or debugging, this implies moving from $O(n)$ vector associated with each event to $O(n^2)$ set of data associated with each event.

The closest work to this paper is our previous paper [16] which presented an offline algorithm for computing dimension. That paper presented results which indicated that a significant number of distributed computations have a low dimension. However, the work was offline, where this is an online algorithm.

3 An Online Algorithm

In this section we will describe the algorithms we have developed to bound the dimension of the partial order online. The basic idea is to generate incrementally the critical pairs of the partial order and then create extensions to the partial order that reverse those pairs. To achieve this in an online algorithm, we treat the set of events and their interaction at any given instant as the complete partial order. The arrival of a new event (that is, information pertaining to a previously unknown event) then modifies that partial order. Our algorithm must therefore do three things. It must determine which critical pairs are no longer valid when the new event arrives, which new critical pairs are created by the new event, and it must be able to incrementally reverse these critical pairs into a set of extensions of the partial order. We will now describe our solutions to these problems.

3.1 Computing Critical Pairs Online

To compute the set of critical pairs online we build on the definitions and theorem from our previous paper [16]. First we recall the definitions of $\mathrm{leastConcurrent}_E(e)$ and $\mathrm{greatestConcurrent}_E(e)$ and their relationship to the critical pairs of a partial order

$$e_l \in \text{leastConcurrent}_E(e) \iff e_l \parallel_E e \wedge \left(\nexists_{e'_l}\ e'_l \parallel_E e \wedge e'_l \rightarrow_E e_l \right) \tag{5}$$

$$e_g \in \text{greatestConcurrent}_E(e) \iff e_g \parallel_E e \wedge \left(\nexists_{e'_g}\ e'_g \parallel_E e \wedge e_g \rightarrow_E e'_g \right) \tag{6}$$

Theorem 2.

$$(x, y) \in \text{CP}_E \iff x \in \text{leastConcurrent}_E(y) \wedge y \in \text{greatestConcurrent}_E(x)$$

These definitions and theorem are sufficient for an offline algorithm where the partial order is fixed. In our online algorithm the set of events that are least or greatest concurrent to a given event may change as new events arrive. As a result, we required two additional theorems: one to indicate which critical pairs were no longer valid in the presence of a new event, and one to indicate which new critical pairs were present as a result of the new event.

In order to be able to efficiently determine the change in the critical-pair set, we must put a restriction on the order in which new events are processed. We require that the events are processed in some linear extension of the partial order. The effect of this condition is that we are now dealing with additions to the partial order of new maximal elements, and of no other kind. This enables the development of our required theorems. The first theorem indicates which critical pairs are no longer valid under the addition to set E of maximal element e.

Theorem 3.

$$(a, b) \in \text{CP}_E \Longrightarrow (b <: e \wedge a \parallel_{E \cup \{e\}} e) \Leftrightarrow (a, b) \notin \text{CP}_{E \cup \{e\}}$$

(a, b) remains in the set of critical pairs provided b is not covered by e or a is not concurrent with e.

Our second theorem defines the entire set of additional critical pairs that may result from the additional of e.

Theorem 4.

$$(a, b) \in \text{CP}_{E \cup \{e\}} - \text{CP}_E \iff (a = e \vee b = e) \wedge a \in \text{leastConcurrent}_{E \cup \{e\}}(b) \wedge b \in \text{greatestConcurrent}_{E \cup \{e\}}(a)$$

The only new critical pairs are those that are formed with the event e as one of the elements of the pair. Furthermore, the combination of these two theorems clearly indicates that if (a, b) is a critical pair but ceases to be after the addition of e, then (a, e) will be a critical pair. We can also observe that in such a case any extension of the partial order that reverses (a, e) must also reverse (a, b) since $b \rightarrow e$.

These theorems and observations lead to two possible approaches. We may generate the critical pairs and reverse them without heed to whether or not they are ultimately valid critical pairs, since the extensions we build will reverse any pair that was at one time believed to be a critical pair. This is consistent with the view that the entire set of events received thus far constitutes the whole of the partial order. A drawback with

```
 1: ∀_e { // For each new event 'e'
 2:   ∀_b b <: e {
 3:     if (a, b) ∈ tentative_CP {
 4:       if (a ∥ e) {
 5:         remove (a, b) from tentative_CP
 6:       }
 7:       else if (e is last covering event of b) {
 8:         remove (a, b) from tentative_CP
 9:         reverse (a, b)
10:       }
11:     }
12:   }
13:   l_C ← leastConcurrent(e)
14:   ∀_l l ∈ l_C {
15:     add (l, e) to tentative_CP
16:   }
17:   g_C ← greatestConcurrent(e)
18:   ∀_g g ∈ g_C {
19:     if (e ∈ leastConcurrent(g)) {
20:       add (e, g) to tentative_CP
21:     }
22:   }
23: }
```

Fig. 1. Critical Pair Calculation

this approach is the possible additional cost of such insertions. The alternate approach is to wait to reverse a critical pair until it is certain that the pair will always remain valid. From Theorem 3 we know that a critical pair, (a, b), will always remain valid if it is a critical pair after all events that cover b have been processed. In the context of a distributed computation, this condition is satisfied once the successor event to b in the process and any corresponding receive event (if b is a send event) have been processed. The algorithm shown in Fig.1 implements the second technique. It may be modified to implement the first technique by omitting lines 1–12 and altering lines 15 and 20 to instead reverse the pair, rather than add the pair to a tentative list.

There are three things that must be performed for each new event arrival. First, we must deal with all tentative critical pairs whose second element is covered by the new event. Such pairs cannot be critical pairs if their first element is concurrent with the new event. On the other hand, such pairs must be moved to the permanent set (*i.e.* reversed) if the new event is the last element to be processed that covers the second element of the pair and that new event is not concurrent to the first element of the pair. Second, we must consider all events that are leastConcurrent the the new event. Since the new event is, by requirement, a maximal element, it must be greatestConcurrent to all events that are leastConcurrent to it. As a result, any event leastConcurrent to it forms a tentative critical pair with it. Finally, we must consider events that are greatestConcurrent to the new event. If the new event is leastConcurrent to such events, they too would form a tentative critical pair.

Ignored in this algorithm is what to do at the end of a computation. For the first technique there is no problem. For the second technique, there may be valid critical pairs still in the tentative critical pair list. To deal with this we add a fake unary event to the end of each process. Then for those events, b, covered by the fake unary, which form tentative critical pairs (a, b) we move (a, b) to the permanent set. Note that if b is a send event, we must wait for the corresponding receive event to be processed before adding the fake unary.

Before we proceed to deal with critical pair reversal, we will briefly examine the computation costs. We implemented our algorithm as a client to the POET server. As such we have access to Fidge/Mattern timestamps for each event. Therefore it costs $O(w^2)$ to compute the least concurrent set [16], where w is the width of the partial order (that is, the number of processes in the computation). To compute the greatest concurrent set we observe that, since e is maximal, the only elements that could be greatest concurrent to e must also be maximal. We therefore start with the greatest event we have seen on each process so far. We then remove all events in this set that are not concurrent with e. Finally, we remove from the set any event that precedes some other event in the set. As with the leastConcurrent computation, this will be $O(w^2)$.

Due to the need to compute the least concurrent set of g for each g that is greatest concurrent to e (of which there may be as many as w), the algorithm in Fig.1 is $O(w^3)$. However, we can optimize away the need for this least concurrent of g computation by the following theorem.

Theorem 5.

$$x \parallel y \Longrightarrow (x \in \text{leastConcurrent}(y) \Leftrightarrow (\forall_z z <: x \Rightarrow z \rightarrow y))$$

Thus, rather than compute the complete leastConcurrent set, it is sufficient to check that the events covered by g are predecessors of the new event. Since POET only supports point-to-point communication, the set of events that are immediate successors or predecessors to an event has cardinality less than 3. This means that the cost to compute the critical pairs is $O(w^2)$ per event or $O(w^2 n)$, where n is the total number of events, for the whole distributed computation.

3.2 Incremental Critical Pair Reversal

Building the minimum number of extensions that reverses the critical pairs of a partial order is NP-hard for dimension greater than two [17]. Instead we propose a reasonably efficient algorithm that will not give an optimal solution, but will provide an upper bound on the minimum number of extensions necessary (that is, the dimension). Insofar as the dimension-bound we compute is small, it is a satisfactory tradeoff. To achieve this we developed a two-step algorithm. First we select the desired extension in which we will reverse the current critical pair and then we insert it in that extension.

The first step should not imply that the elements of the critical pair are only inserted into one extension, as per our offline algorithm [16]. In this algorithm all extensions contain all events of the partial order, including the relevant constraints of that partial order. The way we achieve this is to build a transitive reduction DAG of the partial order, with Fidge/Mattern vector clocks, as events arrive. An extension of the partial order will contain some additional constraints on some events, and alter various of the vector clocks. We do this by indexing into the extension for events, and having those events point back to the partial order DAG if the extension has not altered the event.

The second step is, in requirement, identical to that required in our offline algorithm [16]. We can insert a critical pair into an extension if we can add the events of the critical pair to the extension, such that they are reversed, and that the reversal does not cause a cycle. An extension *accepts* a critical pair if the critical pair may be inserted

into that extension. An extension *rejects* a critical pair if it does not accept it. Since insertion into an extension may fail, the first step must have a strategy for selecting an alternate extension in which to place the critical pair. The strategy we used is the same greedy one as in our offline algorithm [16].

Extension Insertion For the second step of the algorithm, we had to define a method for inserting critical pairs into an extension that would be acceptably efficient for incremental purposes. The algorithm developed in our previous paper required the examination of every event stored in the extension. While in that paper we merely stored the events that were a part of a critical pair, such extensions could get quite large, and thus unacceptably slow for an incremental algorithm.

The approach we take for this incremental algorithm is to update the vector clocks as a result of the critical pair reversal. We may then use the vector clocks to determine if the insertion of the reversal of the pair would cause a cycle in the extension, and thus must be rejected. Specifically, if we wish to add critical pair (a, b) we first determine if the extension implies $a \rightarrow b$. This may be done using the standard Fidge/Mattern precedence test in constant time. As such, before we change anything, we know whether or not the critical pair will be accepted.

In the event that the critical pair reversal is accepted, we must update the vector clocks of any events affected by this new edge in the DAG. The vector clock update is performed according to standard Fidge/Mattern algorithm. If (a, b) is the critical pair that is being reversed, then the set of events that need to have their vector clock updated is $\{x \mid a \rightarrow x \wedge b \not\rightarrow x\}$. The justification for this set should be clear, since these are precisely those events for which the existence of b was not known in their vector clocks.

Some observations should be made regarding the size of this set. For a given event, it can be as large as $n - 1$. It cannot be as large as $n - 1$ amortized (over critical pair reversals, not events). With some particularly pathological partial orders and linear orderings of same, it can be $O(n)$ amortized, though it appears that in such cases the number of critical pairs would be only a small fraction of n. In practice, it does not appear to be a substantial fraction of n. We therefore use the constant v to represent the average size of this set. The value of v is determined experimentally. Since acceptance is determined in constant time, critical pair reversal will be $O(vw)$ amortized, which reflects v vector clock updates, each with size w.

4 Algorithm Analysis and Experimental Results

We now turn to studying the quality of the algorithm. There are three aspects to this. We wish to study the efficiency of the algorithm, we wish to look at the experimental results, in terms of dimension-bounds produced, and we would like to know how tight those bounds are, given that the algorithm does not produce the optimal bound.

We have already seen that the computation cost to determine the critical pairs is $O(w^2)$ per event, where w is the number of processes. The cost of reversing a critical pair is $O(vw)$. There can be as many as $2w - 2$ tentative critical pairs generated by a new event. The total cost per event then depends on whether we reverse these tentative critical pairs immediately, or wait until it is confirmed that the pair will remain a critical

pair under the addition of new events. If we use the first technique, then the total cost per event is $O(w^2 + vw^2)$. If we use the second technique, we observe that at most two critical pairs in the tentative list can be made permanent (and thus must be reversed) for any new event. As such, the cost per event is $O(w^2 + vw)$.

Several comments should be made on this. Clearly the second technique has a lower cost, but how much lower (and whether or not that is significant) depends on two factors. First, it depends on how many tentative critical pairs are generated per permanent critical pair. While we put upper bounds on these numbers analytically, the actual numbers had to be determined experimentally. What we have found is that there are at least 3 and and upwards of 50 tentative critical pairs generated per permanent critical pair, being larger for a partial orders with larger width. Thus, $O(w)$ tentative critical pairs generated per new event does seem to be born out in experiment. The second factor in the cost difference between the techniques is the value v. If this were small, then the cost difference would still be small. Unfortunately this does not appear to be the case, and thus the second technique is clearly preferred from an efficiency perspective.

Ignored in our analysis is the cost of maintaining the list of tentative critical pairs. We ignore the cost of maintaining this list as it may be implemented using any reasonably efficient indexed data structure, which will have a cost much less than $O(w^2)$. The only problem would be if this list started to grow substantially. However, it turns out that this is not the case. This would only happen if the number of critical pairs in the computation exceeded twice the number of events in the computation. While this does occur in some short-lived computations, it is much more common that the number of critical pairs is less than the total number of events.

Our experimental results show little difference in dimension-bound produced from our offline algorithm [16]. In some instances the bound is higher by one, in others lower by one, in most it is the same. We will therefore not repeat the results here, other than to observe that over a range of environments and computations using between 3 and 300 processes, the dimension-bound was 10 or less.

The final question is what is the quality of the dimension-bound produced. This is something that we are currently investigating. We know that the bound we produce will never exceed w as to do so would require more than w mutually conflicting critical pairs. This would imply that the dimension could exceed the width, which is not possible [15]. Beyond this, we do not know how well the dimension-bound produced by the algorithm compares with the actual dimension. We plan to analyze this further.

5 Further Work

There are several areas in which we are actively working. We are attempting to discover the analytical quality of the bounds produced by this algorithm. In addition, we are developing an online variant of the Ore timestamp. Such a timestamp would have a size bounded by dimension, not number of processes. In this regard, we intend to apply our dimension algorithm recursively on the extensions that we develop in the course of this algorithm, as we believe it may give us insight into the problem.

Acknowledgment

The author would like to thank IBM for supporting this work and David Taylor for many useful discussions regarding this work.

References

[1] B. Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters*, 39:11–16, July 1991.

[2] Colin Fidge. Fundamentals of Distributed systems Observation. Technical Report 93-15, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1993.

[3] Jerry Fowler and Willy Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990.

[4] Jessica Zhi Han. Automatic Comparison of Execution Histories in the Debugging of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.

[5] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[6] Christian E. Jaekl. Event-Predicate Detection in the Debugging of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1997.

[7] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994.

[8] Thomas Kunz, James P. Black, David J. Taylor, and Twan Basten. POET: Target-System Independent Visualisations of Complex Distributed-Application Executions. *The Computer Journal*, 40(8):499–512, 1997.

[9] Leslie Lamport. Time, Clocks and the Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, 1978.

[10] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, December 1988. Elsevier Science Publishers B. V. (North Holland).

[11] Oystein Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[12] Joseph L. Sharnowski and Betty H. C. Cheng. A Visualization-based Environment for Top-down Debugging of Parallel Programs. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 640–645. IEEE Computer Society Press, 1995.

[13] M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, August 1992.

[14] David J. Taylor. Scrolling Displays of Partially Ordered Execution Histories. In preparation.

[15] William T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins University Press, Baltimore, MD, 1992.

[16] Paul Ward. An Offline Algorithm for Dimension-Bound Analysis. In *Proceedings of the 1999 International Conference on Parallel Processing*. IEEE Computer Society Press, September 1999.

[17] Mihalis Yannakakis. The Complexity of the Partial Order Dimension Problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, September 1982.

[18] Yuh Ming Yong. Replay and Distributed Breakpoints in an OSF DCE Environment. Master's thesis, University of Waterloo, Waterloo, Ontario, 1995.