On the Scalability of Monitoring and Debugging Distributed Computations: Vector-Clock Size

Paul A.S. Ward
pasward@ccnga.uwaterloo.ca

Shoshin Distributed Systems Group
Department of Computer Science
University of Waterloo

Abstract

The vector-clock size necessary to characterize causality in an arbitrary distributed computation is equal to the number of processes in that computation. However, in practice the vector-clock size necessary may be much smaller. The vector-clock size is not strictly bounded by the number of processes, but rather by the dimension of the partial order induced by the computation. While the dimension theoretically can be as large as the number of processes, in practice we have found it to be much smaller. In this paper we quantify exactly how small the dimension, and hence the vector-clock size required, is over a range of typical distributed computations. We have found that typical distributed computations, with as many as 300 processes, have dimension less than 10. In order to achieve this quantification we developed several novel theorems and algorithms which we also describe.

1 Introduction

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. A process-time diagram which shows this partial order of execution can therefore be very useful in monitoring or debugging such computations. An example of a tool

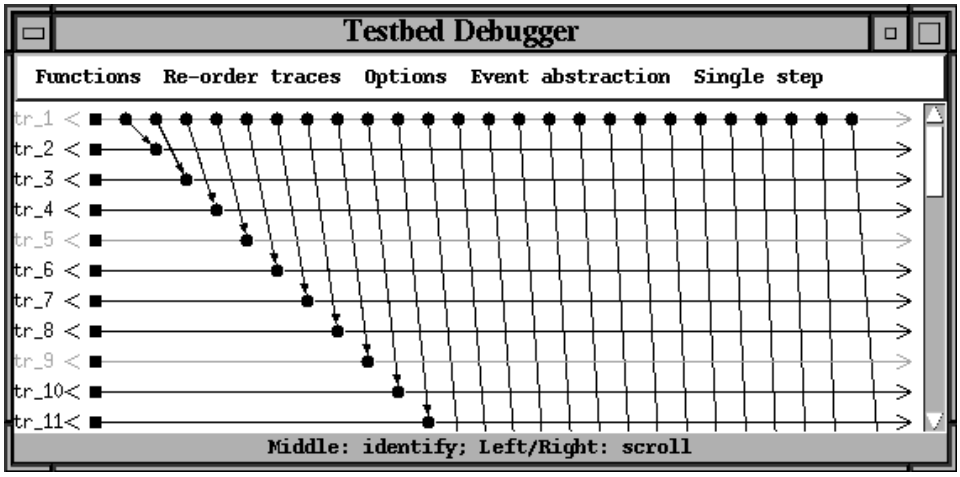

Figure 1: Sample POET Process-Time Diagram

that provides such a display is POET [29, 18]. An example of a POET display, presenting a portion of a point-to-point broadcast, is shown in Figure 1.

There is a significant limitation in POET and in any similar system. Such systems are limited to displaying only a small portion of the execution of at most a few-hundred communicating sequential processes.[1] For our purposes we require a tool that can work with a far greater numbers of processes. If the reader has difficulty imagining such a need, consider visualizing the execution of a parallel database engine over a few-hundred-processor machine. Each processor will execute a few-hundred database processes, communication processes, database-access agents, and so forth. It rapidly enters the realms of the tens or hundreds of thousands of concurrently executing processes.

In this paper we will specify first what the problems are with respect to the scalability of distributed debugging and monitoring, including detailing the necessary partial-order theory. In Section 3 we will describe the theorems and algorithms we

[1]While we will use the term "process" or "sequential process," out of deference to convention, the reader should consider it to be simply any sequential entity. We use the term "sequential entity" to represent any sequential portion of the computation that we wish to monitor or debug. The computation as a whole is parallel, distributed or concurrent, whatever the case may be. However, the entities that make up that parallel, distributed or concurrent computation are themselves sequential. These sequential entities may be processes, threads, semaphores, monitors, shared-memory accesses, or something altogether different. The key property of any such sequential entity is that its behaviour is a total order. Concurrency is then between such sequential entities. We may, from time to time, also use the term "trace" for a sequential entity, as that is the term POET uses. The POET system was developed from the ground up as a target-system-independent tool. This is why we in no way specify in further detail what the sequential entity represents other than that the entity behaves as a total order. POET deals with the partial order of the computation as a partial order. It neither knows, nor cares, what the sequential entities are.

developed to determine the dimension bound of distributed computations and, by inference, the vector-clock size. We then discuss the results we achieved from this after executing the algorithms over several computations in various parallel, concurrent and distributed environments. Finally we indicate what work needs to be completed to achieve scalability in distributed debugging and monitoring.

2 The Scalability Problems

We presume the, by now, standard model of distributed systems, initially defined by Lamport [19]: a distributed system is a system comprising multiple sequential processes communicating via message passing. The sequential processes consist of three types of events: send, receive and unary, totally ordered within the process. A *distributed computation* is the union of all of the events across all of the processes. The set of all events is usually referred to by E. The set of events within a given process is referred to by E_i where i uniquely identifies the process. An individual event is referred to by e_i^j where i identifies the process and j identifies the event's position within the process.

The Lamport "happened before" relation is defined as follows:

Definition 1 (happened before: $\rightarrow \subseteq E \times E$) *"happened before" is 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.*

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

$$e_i^j \parallel e_k^l \iff e_i^j \not\rightarrow e_k^l \wedge e_k^l \not\rightarrow e_i^j$$

The "happened before" relation defines a partial order over the set of events in the distributed computation. It is this partial order that the POET tool displays and it is this display that we wish to be able to scale. There are essentially two aspects that must be dealt with to achieve scaling of these partial-order displays. We must be able to handle more traces (that is, sequential processes), and we must be able to handle more events within traces. It is a subset of the first of these issues that we address in this paper.

Scaling with respect to the number of traces is a hard problem. There are several fundamental challenges that must be dealt with to achieve such scaling. First, since we wish to display the partial order of execution, we find ourselves limited by the typical 20-inch computer monitor and by human cognition. Such a screen is limited to displaying approximately 50 traces. While we could achieve more traces with some perhaps more-compact trace arrangement or a larger screen, the number is ultimately constrained by screen size. Human cognition presents a more

fundamental limitation in this regard. Any user of such a monitoring/debugging tool cannot be expected to extract meaningful information out of a few thousand traces even if they could all be displayed in one screen. Some forms of abstraction are needed to overcome this problem. While some work has been done in this area [5, 28, 13, 14, 15, 16, 17], it is still quite limited. This is, unfortunately, beyond the scope of this paper.

There is a second problem, equally hard, in scaling with respect to the number of processes that is the focus of this paper. In order to build partial-order displays efficiently it is necessary to be able to compare any two events to determine their precedence, if any (that is, whether or not they are an element in the "happened before" relation). This precedence information is also of substantial value to any user of such a display. Indeed, it is one of the primary reasons for using such a tool. This also requires efficient comparison of events to determine their status in the "happened before" relation. In order to perform such an efficient comparison visualization tools typically use vector clocks, usually of the Fidge/Mattern variety [3, 5, 6, 7, 20, 23]. These vector clocks do not scale well with respect to the number of processes. In order to understand the problem of scaling vector clocks it is necessary to first cover a little partial-order theory and then describe the problem in formal terms.

2.1 Some Partial-Order Terminology

The following partial-order theory is due to Trotter [30, 31]. While we will not give detailed partial-order theory here, we will review a few of the important definitions. 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 the restriction of P to the subset. An *antichain* is any completely unordered poset. The *width* of a poset is the longest antichain contained in that poset. In the context of a distributed computation, the width must be less than or equal to the number of processes.

An *extension*, (X, E), of a partial order (X, P) is any partial order that satisfies

$$(x, y) \in P \Rightarrow (x, y) \in E$$

If E 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. We are now in a position to define the dimension of a partial order

Definition 2 *The dimension of a partial order is the cardinality of the smallest possible realizer.*

We are now in a position to formally describe the problem with the scaling of the size of the vector clock.

2.2 The Vector-Clock Size Problem

One of the features of Fidge/Mattern timestamps is that they are a vector of size equal to the number of processes in the distributed computation. This vector size is not going to scale with an increase in the number of processes. With a few hundred processes it requires on the order of 1,000 bytes per event. To increase to a few thousand would require 10,000 bytes per event which is getting beyond the realm of the reasonable. To go to tens of thousands of processes would require on the order of 100,000 bytes per event. This is not feasible.

In 1991, Charron-Bost [3] showed the mathematical justification for vectors of size equal to the number of processes. To compare any two events to determine causality it is necessary to have a vector (or equivalent) whose size is the size of the dimension of the partial order that the distributed computation induces. This dimension can be as large as the number of processes in the distributed computation. Specifically, crown $\mathbf{S}_n^0$ is a partial order with dimension equal to its width. Charron-Bost turned crown $\mathbf{S}_n^0$ into a point-to-point distributed computation, though in essence it remained the same.

The limitation of the Charron-Bost proof is in the nature of what crown $\mathbf{S}_n^0$ represents. It represents a distributed computation in which each process sends a message to all other processes *except* the left neighbour of the sending process. In practical terms, this is not a realistic distributed computation. In addition, the more likely computation, in which each process broadcasts a message to *all* other processes, has dimension 2. It is therefore the objective of this paper to determine a bound on the dimension of actual distributed computations, rather than theoretical ones that are doubtful to ever occur in practice. This bound would represent a more accurate requirement for the size of timestamps necessary to capture causality.

Before we describe how we compute the dimension bound, and what our results are for typical distributed computations, we wish to justify that there are in practice alternate timestamps that are more compact than Fidge/Mattern vector clocks. We will also discuss work related to our own.

2.2.1 Ore Timestamps

There is an alternate timestamp to the Fidge/Mattern vector clock, whose size is bounded by the dimension, not the width, of the partial order. This is the Ore timestamp [21, 27]. Given a realizer for partial order (X, P) the timestamp associated with each $x \in X$ is simply the vector of its position in the various linear extensions that form the realizer. It is then straightforward to see that causality between two events, x and y can be determined by comparing the various elements in the vectors of the two events. If they are all less, then precedence is established. If some are less and some are greater then concurrency is established. While this timestamp has an essentially offline nature, and it is beyond the scope of this paper to correct this deficiency, it does indicate that timestamps whose size is bounded

by the dimension, not the width, of the partial order are not only possible in theory but exist in practice.

2.2.2 Related Work

Before proceeding to describe how we computed dimension bounds, we will briefly describe some of the work related to our own. 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 network visualization tools [22], GOLD [24], ParaGraph [11] and our own system, POET [29, 18]. 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-timestamps 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 first of these systems uses causal message ordering to ensure an appropriate visualization. This is known to be $O(n^2)$ in the number of processes [4], and so clearly is not going to scale. The GOLD system uses dependency vectors, developed by Fowler and Zwaenepoel [9], which will again 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, and 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. Our own system uses Fidge/Mattern timestamps.

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 [25]. The problem with the technique offered is that an $O(n^2)$ matrix is required at each process to recover the vector, which, in the context of event precedence in a debugging environment, would imply a moving from $O(n)$ vector associated with each event to $O(n^2)$ set of data associated with each event.

Finally, there are no performance results that we are aware of concerning the actual behaviour of the various systems in the context of large numbers of processes. In this respect, this work is unique.

3 Bounding the Dimension

Computing the exact dimension of a partial order is known to be NP-hard for any partial order of dimension greater than two [12]. We therefore approached the problem by attempting to simply bound the dimension. For our purposes, an order-of-magnitude difference between the dimension bound and the number of traces would be sufficient to justify proceeding. It was then necessary to develop an al-

gorithm to achieve a reasonable bound in a reasonable amount of time. Rather than take the direct approach of generating linear extensions and then determining if they formed a realizer we chose an indirect route based on the concept of *non-forced pairs*.[2]

Definition 3 (Non-forced pair) (x, y) *is a non-forced pair of partial order* (X, P) *if* $x \parallel y$ *and* $(X, P \cup \{(x, y)\})$ *is a partial order.*

An equivalent definition is

$$(x, y) \in \mathrm{NFP} \iff x \parallel y \wedge \forall_{z \in X} z \rightarrow x \Rightarrow z \rightarrow y \wedge y \rightarrow z \Rightarrow x \rightarrow z$$

where NFP is the set of all non-forced pairs of the partial order (X, P). The significance of non-forced pairs, as regards dimension, is in the following theorem, due to Rabinovitch and Rival [31].

Theorem 1 *Any set of linear extensions of a partial order that reverses all non-forced pairs is a realizer of that partial order.*

A non-forced pair (x, y) is said to be reversed by a set of linear extensions if one of the linear extensions in the set contains $y \rightarrow x$. There is a stronger form of Theorem 1 that we make use of for our algorithm.

Theorem 2 *If* (Y, Q) *is a subposet of* (X, P) *that contains all of the non-forced pairs of* (X, P) *then the dimension of* (Y, Q) *is identical to that of* (X, P).

In simpler terms, it is sufficient to simply reverse the non-forced-pair events. All other events of the computation may be ignored. Note also, that it is not strictly necessary for the extensions to be linear. They merely have to reverse the non-forced pairs.

The approach we have then taken to the problem of bounding the dimension is to first compute all of the non-forced pairs of the partial order (a polynomial-time problem) and then create extensions that reverse these non-forced pairs (an NP-hard problem to find a minimal-cardinality set).

3.1 Computing Non-Forced Pairs

While it is possible to use the given definition of non-forced pairs to compute the set of all non-forced pairs, any such algorithm would be very inefficient. To achieve reasonable performance in the computation it is necessary to develop an association of non-forced pairs with the relations that hold for the partial order. To this end, we define the sets $\mathrm{leastConcurrent}(e)$ and $\mathrm{greatestConcurrent}(e)$.

Definition 4 (leastConcurrent) *The set of events that are* leastConcurrent *to an event* e *are those events that are concurrent with* e *and which have no predecessor which is also concurrent with* e.

[2] Also known as *critical pairs*

In formal terms:

$$e_l \in \text{leastConcurrent}(e) \iff e_l || e \wedge \left(\nexists_{e'_l}\ e'_l || e \wedge e'_l \rightarrow e_l \right)$$

We likewise define the set of events that are greatestConcurrent to event *e*

$$e_g \in \text{greatestConcurrent}(e) \iff e_g || e \wedge \left(\nexists_{e'_g}\ e'_g || e \wedge e_g \rightarrow e'_g \right)$$

This then leads to the following theorem:

Theorem 3

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

Proof: Necessary: Assume $(x, y) \in \text{NFP}$. If $x \notin \text{leastConcurrent}(y)$ then, because $x \parallel y$, $\exists_z z \rightarrow x \wedge z \parallel y$. This implies that $\exists_z z \rightarrow x \wedge z \not\rightarrow y$ which contradicts $(x, y) \in \text{NFP}$. Likewise, if $y \notin \text{greatestConcurrent}(x)$ then $\exists_z y \rightarrow z \wedge z \parallel x$. This then implies that $\exists_z y \rightarrow z \wedge x \not\rightarrow z$ which again contradicts $(x, y) \in \text{NFP}$.

Sufficient: Assume $x \in \text{leastConcurrent}(y) \wedge y \in \text{greatestConcurrent}(x)$. Show $\forall_z z \rightarrow x \Rightarrow z \rightarrow y$. If $z \rightarrow x$ then $z \not\parallel y$ because $x \in \text{leastConcurrent}(y)$. Also, if $z \rightarrow x$ then $y \not\rightarrow z$ since if it did, by transitivity $y \rightarrow x$ which contradicts $x \in \text{leastConcurrent}(y)$. Therefore $\forall_z z \rightarrow x \Rightarrow z \rightarrow y$. Likewise we may show $y \rightarrow z \Rightarrow x \rightarrow z$. If $y \rightarrow z$ then $x \not\parallel z$ because $y \in \text{greatestConcurrent}(x)$. Also, if $y \rightarrow z$ then $z \not\rightarrow x$ since if it did, by transitivity $y \rightarrow x$ which contradicts $y \in \text{greatestConcurrent}(x)$. Therefore $\forall_z y \rightarrow z \Rightarrow x \rightarrow z$. □

This theorem enabled the development of the following algorithm:

```
For each event y in computation {
   lC <- leastConcurrent(y)
   for each event x in lC {
     gC <- greatestConcurrent(x)
     if (y in gC) {
        output (x,y) is NFP
     }
   }
}
```

Some comments should be made about this algorithm. First, it is not obvious from the above why we perform the computation in what appears to be the reverse order. That is, we take each event as a possible second element of a non-forced pair, rather than a first element. The reason has to do with the relative cheapness with which we can compute the leastConcurrent set versus the comparative expense of computing the greatestConcurrent set. As we are still working with the POET tool, we have access to Fidge/Mattern timestamps for each event. An important

property of these timestamps is that they give the set of events that are the greatest predecessors to the event for which they form the timestamp. What this means is that we can quickly compute the leastConcurrent set of an event as follows

```
leastConcurrent(e) {
   lC <- timestamp(e) + 1;
   for all x in lC {
      if (!(x||e))
         lC <- lC - x;
   }
   for all x in lC
      for all y in lC {
         if (x -> y)
            lC <- lC - y;
      }
   return lC;
}
```

In words, to compute leastConcurrent(e) we start with the timestamp of e, that is, those events that are the greatest predecessors to e on their respective traces. We advance this timestamp by one; that is, we increment each element of the timestamp. This now represents a set of events that are either concurrent or successors to e. We refer to this set as the set of potentially least concurrent events of e. We remove from this set any event that is not concurrent with the event e and any event that is preceded by some other event within the set. This leaves those events that are leastConcurrent(e).

The greatestConcurrent(e) set is more expensive to calculate. To compute it we start with the greatest predecessors (*i.e.* the Fidge/Mattern timestamp) of e and then iterate along each trace until we reach a successor event to e, or we come to the last event in that trace. This yields the potentially greatest concurrent set. We then remove all events in this set that precede any other events in the set. This leaves the greatestConcurrent set.

In the greatestConcurrent(e) calculation there are two optimizations that may be performed. The first is to use a binary search on the trace, rather than iterating through each event. The second is to attempt to avoid calculating it at all where possible. Specifically, when we have a first event of a potential non-forced pair, we already know what the matching second event must be (as we used it to compute the first event). Rather than compute the greatestConcurrent(x) set immediately, we first check to see if x is concurrent with the successor of y on the same trace as y. This is a very fast check to perform, since we can index into this successor event. If x is concurrent with this successor of y then we know that y cannot be in the greatestConcurrent(x) set, so there is no need to compute the set. This optimization reduces the need to compute the greatestConcurrent set by an order of magnitude or more.

3.2 Reversing Non-Forced Pairs

Reversing the non-forced pairs to achieve the minimum number of extensions is NP-hard. We prefer instead a reasonably efficient algorithm that gives a satisfactory upper bound on the minimum number of extensions necessary. To achieve this we developed a two-phase algorithm. First we select the desired extension into which to place the current non-forced pair and then we insert it in that extension.

The first phase may seem strange, since, even under the stronger non-forced pair theorem, the extensions of the subposet of the computation that contains all non-forced pairs must contain all of the events of that subposet. Under our algorithm, the "extensions" that we develop only contain a subset of the events of the subposet. In other words, the extensions that we develop are skeletal extensions that reflect the reversal of some of the non-forced pairs. However, although we do not insert all of the events of the subposet, we know that it is possible to insert all of the events within any given extension that we are building. The reason it is possible is that the non-forced pairs are concurrent in the original partial order, and so although they add constraints to the extension, they in no way conflict with the original partial order. Those events of the subposet that are not inserted into the extension could be so inserted, but they would not be reversed, and it is this reversal that we care about. In essence, we are building skeleton extensions that contain as many non-forced pair reversals as possible. Other events from other non-forced pairs are capable of fitting, but will not be reversed, and so we ignore their addition.

To understand the second phase we must define what it means to insert a non-forced pair into an extension. It means that we can add the events of the non-forced pair, such that they are reversed, and that it violates neither the partial order, nor the additional constraints that the reversal requires. It may also be the case, if the insertion algorithm is not optimal, that it rejects the non-forced pair even though it did not violate these conditions, but rather violated some aspect of the structure in which the extension was kept.

It is, perhaps, helpful to consider a simple example. Suppose we have a partial order consisting solely of two concurrent events, a and b. It has non-forced pairs (a, b) and (b, a). Once (a, b) has been inserted into an extension, that extension must reflect the constraint $b \rightarrow a$. As such (b, a) cannot be inserted into that extension, since it would require the extension to reflect $a \rightarrow b$.

We say that an extension *accepts* a non-forced pair if the non-forced pair may be inserted into that extension. An extension *rejects* a non-forced pair if it does not accept it. Finally, since insertion into an extension may fail, the first phase must have a strategy for selecting an alternate extension in which to place the non-forced pair. We can now describe the specific algorithms used for the two phases.

The algorithm we used for the extension-selection phase is a simple greedy one. We insert the current non-forced-pair events into the first extension that will accept it. In the event that all current extensions reject the non-forced pair, we create a new extension, containing no events, that must, by definition, accept the

non-forced pair. The non-forced pair is inserted into the new extension. Thus the first-phase algorithm is

```
insert(x,y) {
   // Insert in the first extension that will accept
   for (i = 0; i < numberExtensions; ++i)
      if (insert(extension[i], x, y))
         return;
   // None accepted; create a new extension
   create(extension[numberExtensions]);
   insert(extension[i], x, y);
   ++numberExtensions;
   return;
}
```

For the second phase of the algorithm, we had to define a method for inserting non-forced pairs into an extension. The initial algorithm that we developed was a greedy one that worked on the principle "place the event before the first event it *must* precede." While this approach produces some promising results, it also produces some spectacularly bad ones. We therefore decided to develop an optimal solution to this second phase. We maintain a directed acyclic graph for each extension. To add a non-forced pair we add the two events in turn and then determine if the graph is still acyclic. If it is acyclic we have accepted and inserted the non-forced pair. If it is not, we reject the non-forced pair, and remove the evidence of the addition. This method proved to be acceptable, as we can see in the next section.

4 Results and Observations

We have executed our dimension-bound algorithm over several dozen distributed computations covering over half-a-dozen different parallel, concurrent and distributed environments and a range of 3 to 300 traces. The environment types are self-debug (POET is capable of monitoring itself, as it is a distributed system), the Open Software Foundation Distributed Computing Environment [8], the μC++ shared memory concurrent programming language [1, 2], the Hermes distributed programming language [26] and the Java programming language [10]. Various of the raw results are shown in Tables 1, 2 and 3.

The quick summary is that the dimension bound that we discovered over this range of computations and environments was always 10 or less. For computations of trace count greater than 20 there is a minimum of an order of magnitude difference between the dimension and the number of traces. When the number of traces is greater than 100, it is usually a factor of 15 or greater. To help visualize what these results imply, we created a graph, shown in Figure 2, which plots dimension as a function of the number of traces. The two graphs shown are the same, but with

Number of Events	Number of Traces	Number of Non-Forced Pairs	Dimension Bound
45	5	12	3
90	19	27	2
121	20	61	4
249	40	124	3
291	42	164	3
467	42	183	4
297	44	237	4
499	70	443	5
501	72	496	6
833	110	1490	9
817	112	1378	8
928	114	1738	7
902	115	1402	8
1560	159	3579	10

Table 1: Dimension Results for OSF DCE Computations

differing scales. The x-axis is the number of traces while the y-axis is the dimension bound. We also plot two additional lines. First we show the "dimension = 10" line, as all results were less than or equal to that value. Second, we show the "dimension = width" line, which illustrates the increase in Fidge/Mattern vector-clock size as the number of traces increases.

In addition to testing with distributed computations, we created, using the testbed environment (which enables us to create arbitrary point-to-point partial orders), a series of broadcast patterns of varying sizes and crown patterns. The results from our program for these patterns are shown in Table 4. This helps to illustrate the quality of our algorithm. The dimension of point-to-point broadcast is 3, for any

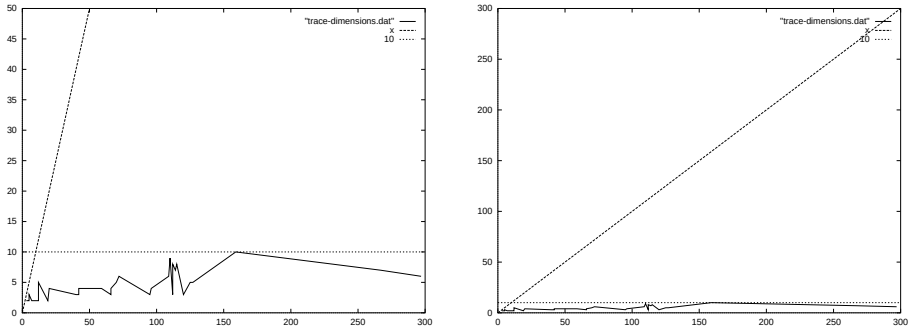

Figure 2: Dimension as a Function of Number of Traces

Number of Events	Number of Traces	Number of Non-Forced Pairs	Dimension Bound
15	7	38	2
2687	59	2360	4
3252	66	3044	4
2612	66	2032	3
49791	95	6622	3
3272	96	5906	4
7426	109	10019	6
7928	112	6969	3
35266	112	6675	4
30048	120	7999	3

Table 2: Dimension Results for Java Computations

size broadcast. Our algorithm achieved this optimal bound.[3]

5 Further Work

There are two broad themes that we are actively pursuing with regard to this work. The first is the development of better vector clocks. In particular, Ore timestamps are, in their current definition, offline in nature. That is, they work on the partial order as a whole entity, which typically presumes a completed computation. We require an online equivalent that has the size properties of the Ore timestamp without the offline nature.

The second broad theme we are pursuing is scaling the monitoring system in

[3] As a side note we would comment that our greedy algorithm for non-forced pair insertion yielded a dimension bound of $N - 2$ for these point-to-point broadcasts. We said it could produce spectacularly bad results!

Environment	Number of Events	Number of Traces	Number of Non-Forced Pairs	Dimension Bound
self-debug	405	3	10	2
self-debug	9693	5	23	2
μC++	360	12	156	2
μC++	1750	12	853	5
Hermes	1888	125	1323	5
Hermes	1944	127	1429	5
Hermes	4164	267	4403	7
Hermes	4086	297	21401	6

Table 3: Dimension Results for Various Environments

Pattern	Number of Events	Number of Traces	Number of Non-Forced Pairs	Dimension Bound
Crown $\mathbf{S}_3^0$	6	3	3	3
Crown $\mathbf{S}_5^0$	10	5	5	5
Broadcast-5	13	5	20	3
Broadcast-10	28	10	90	3
Broadcast-20	58	20	380	3
Broadcast-40	118	40	1560	3

Table 4: Testbed Results for Various Partial Orders

the context of visualization. As we discussed in Section 2, it is not sufficient to simply show the user several thousand traces and expect such a display to be of any value. Nor is it reasonable to expect the user to manually perform some type of clustering. The whole basis of displaying the partial order needs to be rethought when we consider a substantial increase in the trace count.

Acknowledgments

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

About the Author

Paul Ward is a Ph.D. candidate in the Department of Computer Science at the University of Waterloo. His Internet address is pasward@ccnga.uwaterloo.ca.

References

[1] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, February 1992.

[2] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.6. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1996. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.

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

[4] B. Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, Asynchronous and Causally Ordered Communication. Submitted to Distributed Computing.

[5] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1989.

[6] Colin Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, 1991.

[7] 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.

[8] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[9] 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.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/.

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

[12] Thomas Kunz. Personal communication.

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

[14] Thomas Kunz. Visualizing Abstract Events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, October 1994.

[15] Thomas Kunz. Automatic Support for Understanding Complex Behaviour. In *Proceedings of the International Workshop on Network and Systems Management*, pages 125–132, August 1995.

[16] Thomas Kunz. High-Level Views of Distributed Executions. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, pages 505–512, May 1995.

[17] Thomas Kunz. Evaluating Process Clusters to Support Automatic Program Understanding. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 198–207, March 1996.

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

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

[20] 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).

[21] O. Ore. *Theory of Graphs*. Number 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[22] Guru Parulkar, Douglas Schmidt, Eileen Kraemer, Jonathan Turner, and Anshul Kantawala. An Architecture for Monitoring, Visualization, and Control of Gigabit Networks. *IEEE Network*, pages 34–43, September/October 1997.

[23] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[24] 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.

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

[26] R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[27] James Alexander Summers. Precedence-Preserving Abstraction for Distributed Debugging. Master's thesis, University of Waterloo, Waterloo, Ontario, 1992.

[28] David J. Taylor. The Use of Process Clustering in Distributed - System Event Displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512, January 1993.

[29] David J. Taylor. Event Displays for Debugging and Managing Distributed Systems. In *Proceedings of the International Workshop on Network and Systems Management*, pages 112–124, August 1995.

[30] William T. Trotter. Graphs and Partially-Ordered Sets. In R. Wilson and L. Beineke, editors, *Selected Topics in Graph Theory II*, pages 237–268. Academic Press, 1983.

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