1

Causality-Preserving Timestamps in Distributed Programs

Adam Beguelin Erik Seligman

June 1993
CMU-CS-93-167

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]This research was sponsored in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035. Support was also provided by a National Science Foundation graduate fellowship awarded to Erik Seligman.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Distributed,parallel,timestamps,logical timestamps,supercomputing,clock, monitoring,debugging,tachyon,causality

Abstract

A *tachyon* is an improperly ordered event in a distributed program. Tachyons are most often manifested as messages which are received before they are sent, violating the principle of causality. Although tachyons are not possible in "real life", they may appear to occur in distributed parallel program traces due to coarse clock granularity or poor clock synchronization. In this paper, we establish that tachyons do in fact occur commonly in distributed programs on our Ethernet at Carnegie Mellon University, and we discuss some ways of eliminating them from program traces while preserving at least some knowledge of the length of time intervals in our programs. Our methods are based on Lamport-style clock corrections; when a process receives a message stamped with a later sending time, it sets its own clock ahead to a time at least as great as the sending timestamp. We have implemented this both in real time and in a more comprehensive post-processor for Xab.

1 Introduction

When writing and debugging parallel programs, many programmers find it useful to be able to view an *event trace*, a sequential listing of each communication event and the time it has occurred. Many useful tools, such as Xab [4] and ParaGraph [7], have been created to better visualize parallel program traces.

One important property that we would like these traces to observe is the preservation of *causality*; if event A could have caused event B, then event A must have happened at an earlier time than B. A message that does not obey this property (it is received before it is sent) is called a *tachyon*. Clearly it is very disconcerting to try to debug a parallel program that contains tachyons.

Of course, in "real life", causality cannot be violated. In a trace file of a distributed parallel program, however, causality violations can appear, due to poor clock granularity or poor clock synchronization. In this paper, we establish that these causality violations do in fact occur, and discuss some ways to eliminate them from trace files.

2 The Problem

As discussed above, one important property that a trace file should have is that the timestamps reported in a program trace preserve causality. We say event A causally precedes event B (see [8]) if

- Events A and B occur at the same process, and A occurs before B,
- Event A sends a message received during event B, or
- There is an event C such that A precedes C and C precedes B.

If event A causally precedes event B, then the "real time" at which A occurred must be earlier than the time at which B occurred. If the timestamps in the program trace obey this commonsense property, then the program flow will conform to our notions of causality, and flow graphs created by utilities such as ParaGraph will not show messages travelling backwards in time.

In order to determine if causality violations occurred, we ran a small test program on approximately ten machines (including Sun4's and Pmax's, all running Mach) connected by Ethernet here at Carnegie Mellon. The

program was written using PVM version 3.1 (see [2, 3, 6, 10]), and consisted of a "master" process on one machine and a slave process on each of the others. Every five minutes, the master process would awaken and do the following for each slave:

1. Get the current time M1.
2. Send a message to the slave.
3. Wait for a reply message.
4. Get the time M2 immediately after receiving the reply message.
5. Read the values S1 and S2 from the reply, where S1 is the time at the slave just after receiving the original message and S2 is the time at the slave just before sending its reply.
6. Calculate the values (S1-M1) and (M2-S2); if either of these is negative, we have found a tachyon.

The slave programs were very simple:

1. Wait for a message from the master; upon receipt, get the time S1.
2. Get the time S2, and write S1 and S2 in a reply message.
3. Send the reply to the master, and go back to step 1.

The time differences for messages from the master to the slaves during one trial are shown in figure 1. The x-axis represented the iteration number; as mentioned above, the iterations were five minutes apart. The y-axis represents the time difference measured in microseconds. Each point on the graph represents the time difference (S1-M1) for one message from the master to a slave process. (The messages from the slaves to the master are not represented.) For several iterations, the time differences were very large (up to $1.9 * 10^6$ microseconds), and were omitted from the graph.

Tachyons turned out to be more common than we had expected. We observed two types of tachyons:

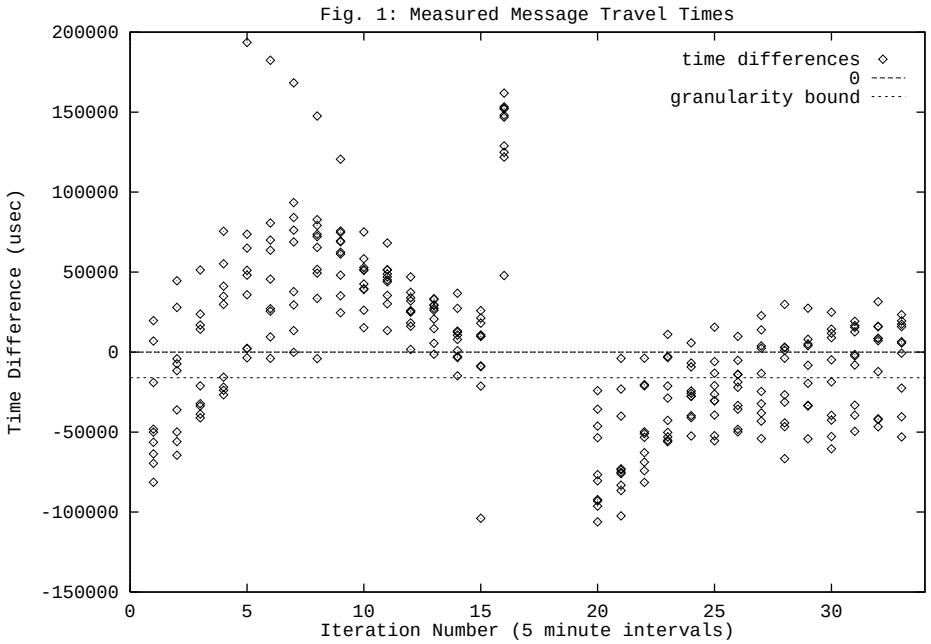

Figure 1: Measured Message Travel Times. Each point represents the difference between send and receive times for one message.

- Tachyons which appear due to poor clock granularity. If the system clock only changes every 15 milliseconds, for example, and a message travels more quickly than that, then the difference between the times observed at the sender and receiver will be essentially meaningless. (We measured the clock granularities on the machines involved; on the pmaxes it was between 15 and 16 milliseconds, and on the sun4's between 9 and 10 milliseconds.)

- Tachyons which are due to poor clock synchronization. If the sending machine's clock is sufficiently far ahead of the receiver's, then the measured time difference between sending and receipt will be negative.

In figure 1, the lower horizontal line marks a bound on the clock granularity (16 milliseconds); thus any points which appear below the line are due to poor clock synchronization, and those between the two horizontal lines (the upper one is y=0) may be due to clock granularity as well.

3 Preserving Causality

An obvious answer to the problem of tachyons is simply to synchronize the clocks of the processes at the beginning of a computation. This will not always be possible, however. Often the person running the parallel program will not be the exclusive owner or user of the machine, and may not be authorized to interfere with the system clocks. More importantly, though, this solution makes the assumption that all clocks will run at the same rate throughout the life of the computation, which is not necessarily true. From the drift observed during our experiment (see Fig.1), we can see that the clock rates vary over time.

Lamport [8] proposed "logical clocks", a simple solution to the problem of timestamping events to preserve causality. Each process keeps a counter, which it increments by 1 after each communication event. Whenever a process sends a message, it timestamps the message with its current clock value. When a process receives a message, it sets its clock to the maximum of its current clock and the message's timestamp. This insures that if A causally precedes B, then the timestamp of B will be at least as great as the timestamp of A.

Of course, this comes at the price of eliminating all information about the real time intervals between events. One way we have tried to solve this problem is the following: Have every process keep track of a real time (i.e., the system clock) and an "offset", initially 0. The parallel program will view its "total time" as being equal to the real time + the offset. Whenever a message is sent, it will be timestamped with the total time at the sender. When a message is received, the recipient checks to see if its total time is at least as great as the message's timestamp; if not, it will increase its offset so it is. Thus, just as with Lamport's logical time, causality will be preserved; in addition, the difference in timestamp between two events will be a reasonable approximation to the actual time difference, if not too many Lamport-style clock corrections intervene. Using this basic idea, we implemented two different methods for eliminating tachyons in Xab.

3.1 Approach 1: Real-time Logical Timestamps

The first method we used to insert (extended) logical timestamps into trace files was to modify Xab directly. The Xab instrumentation was changed so that before sending a message, a timestamp is inserted; when the message is received, the timestamp is immediately checked and the local clock offset changed if necessary.

Unfortunately, there are several pvm calls (such as pvm_barrier() and pvm_pstatus()) that communicate information at a lower level, so that tachyons can be introduced but there are no messages visible at the Xab level. Without seriously interfering with the program execution by inserting extra messages, the only way to enforce causality in these cases would be to modify pvm directly. The PVM developers are currently integrating Xab-style events into PVM version 3.

3.2 Approach 2: An Xab Postprocessor

Our second approach was to post-process the Xab trace. Of course, this will not help during realtime monitoring, but is a much more general method, in that it can be extended to handle pvm_barrier(), pvm_pstatus(), etc.

The basic algorithm is to use the event trace to construct the communication flow DAG (Directed Acyclic Graph) for the program, a graph where a node corresponds to each event and an edge corresponds to the flow of

information between two events. For example, there is an edge from any event to the next event at the same process, an edge between each send and its corresponding receive, an edge between a barrier and each corresponding barrier_done, etc. Once the DAG is constructed, it is traversed in a topological order, and each node's timestamp is corrected to be at least as great as the maximum timestamp of any of its predecessors. The complexity of this computation is O(E*P) in the worst case, where E is the number of events and P is the number of processes.

Note that this DAG exactly captures the notion of causality, and since each node has a timestamp guaranteed to be greater than any of its ancestors, the corrected timestamps provided by this algorithm will be the same as those provided by our first approach. Thus the processed tracefile can be replayed through Xab and a causality-preserving event ordering will be observed.

4 Observations

In order to measure the distortion produced by our corrected timestamps, we ran the experiment in section 2 calculating the value that our offsets would take on during the computation. Figure 2 shows the progression of the values of the clock offsets over time; note that these values can be viewed as the maximum distortion of the time difference that may be observed between two events. Over the course of a computation, these seem to get rather large, up to 2.5 seconds over the course of a 165 minute run in this case. This is to be expected, since a single erroneous clock will affect the offsets of all the processes to which it communicates.

5 Discussion and Future Work

We have observed that due to poor clock synchronization (combined with clock drift) and relatively large clock granularity, tachyons do indeed occur on a local ethernet, and have explored one general method for correcting the event times in a trace file to eliminate them. We have shown that it is possible to eliminate tachyons, though this comes with the cost of introducing some distortion when trying to figure out the time that has actually passed between two events.

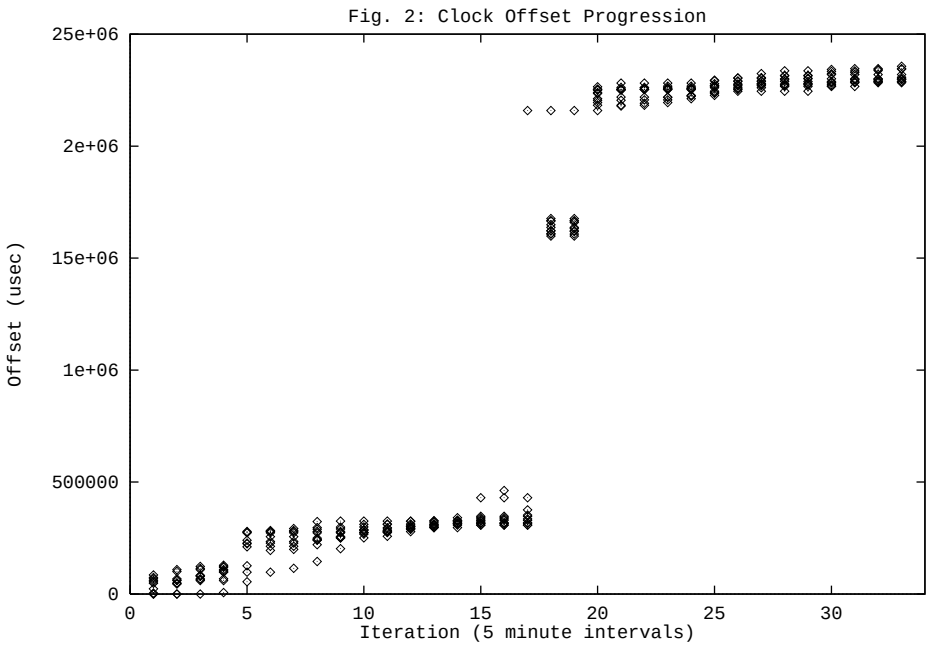

Figure 2: Offset Progression Over Time. Each point represents the clock offset at one machine during one iteration.

There are several further directions to pursue in this area. For example, it seems as if our post-processor could provide a range of possible times for each event, rather than just a (relatively arbitrary) causality-preserving time as it does now. In addition, it might be possible to provide additional useful information at runtime, such as detecting race conditions, perhaps using an algorithm like that in [9]. Finally, the user might be interested in seeing the DAG itself, which could be an additional useful debugging tool.

6 Availablility

Xab and the post-processor (xab-post) described in section 3.2 can be obtained from netlib. Simply send email to `netlib@ornl.gov` with the message "send index from pvm/xab". An index of the software and reports on Xab will be returned to you with instructions on how to obtain the various pieces.

References

[1] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[2] A. Beguelin, J. Dongarra, G. A. Geist, and V. S. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, June 1993. To appear.

[3] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[4] Adam Beguelin. Xab: A tool for monitoring pvm programs. In *Workshop on Heterogeneous Processing*, pages 92–97, Los Alamitos, California, April 1993. IEEE Computer Society Press.

[5] C.J. Fidge. Partial orders for parallel debugging. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.

[6] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[7] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[8] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] Robert H.B. Netzer and Barton P. Miller. Optimal tracing and replay for deugging message-passing parallel programs. In *Supercomputing '92*, pages 502–511. IEEE Computing Society Press, 1992.

[10] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.