An Efficient Logical Clock for Replaying Message-Passing Programs

WEI WANG BINXING FANG

Department of Computer Science
Harbin Institute of Technology
92 Dazhi Street, Nangang District, Harbin 150001, Harbin
P. R. China

Abstract: - Cyclic debugging is one of the most important and most commonly used activities in programs development. During cyclic debugging, the program is repeatedly re-executed to track down errors when a failure has been observed. The cyclic debugging approach often fails for parallel programs because parallel programs reveal nondeterministic characteristics due to message race conditions. Execution replay is a technique developed to facilitate the debugging of nondeterministic programs. The trace file can be used to force the replay of the parallel program with the same input. The size of trace file is very important to evaluate the scalability of record&replay scheme. This paper proposes an improved clock system, called 1-n clock. By combining local logic clock and vector clock, 1-n clock can compress the size of trace file. This method especially supports record&replay of long-running parallel programs.

Key-Words: - Cyclic debugging, record, replay, trace, vector clock, 1-n clock

1 Introduction

Debugging is a process to find out and correct errors by analyzing the program, and debugging is the one of the most important stages in software development. Cyclic debugging [1] is one of the most important and most commonly used activities in programs development. During cyclic debugging, the program is repeatedly re-executed to track down errors when a failure has been observed.

The executions of parallel programs can be nondeterminism. An execution is called nondeterministic if two subsequent runs with identical user input cannot be guaranteed to have the same behavior. During software development this nondeterministic behavior can be very troublesome, as it prevents a reconstruction of what happened during an earlier execution. In fact, nondeterminisim is just one of the difficulties to debug parallel and distributed programs. Lack of global time, multiple threads of control and complex patterns of interaction might make it complex to track down the fault using the information gathered.

Nondeterministic execution of erroneous parallel programs may result in transient errors, which appear very infrequently or vanish when debugging tools are used. The most classical technique used to catch transient errors appearing during executions of parallel programs is to **record** an initial execution and to force subsequent **replayed** executions to be deterministic with respect to the initial execution, using the recorded information. Debugging an erroneous program then amounts to record an erroneous execution and to apply cyclic debugging techniques during subsequent replayed executions. This mechanism can be called **record/replay**. The primary goal of record/replay is to replay a recorded execution as accurately as possible. A second goal is to record with as little intrusion as possible. The third goal is to operate swiftly. When replay is considered, record in implied. So for convenience, we only use the terminology relay in the following contents.

Replay algorithms are either based on the data or the synchronization of instructions. The former is called content-driven replay, and the latter is called ordering-driven replay. Replay was first introduced in [2], which is a content-driven strategy, and the similar approach is described in [3]. The biggest drawback of these approaches is the requirement of significant monitor and storage overhead. Probably the first ordering-driven method is Instant Replay [4]. Several methods are the extension of Instant Replay, including [5], [6], etc. Better introduction about replay systems is given in [7]. One of the first implementations of a replay mechanism for MPI was proposed in [8]. In order to apply an execution replay system in practice, a number of properties should be satisfied, including accuracy, non-intrusiveness, space and time efficiency, etc.

As mentioned above, the size of trace file is an important item to evaluate record/replay algorithm, especially for those long-running parallel programs. To replay the execution of a parallel programs accurately, a timestamp should be assigned to every events occurred during the execution of a parallel program. Vector clock is kind of ideal logical clock, which is used widely in replay stage. But one defect of vector clock is its scalability, for its size equals to the number of processes participating in the execution of a parallel program. This condition will enlarge the size of trace file, especially be a serious disadvantage when long-running parallel programs are concerned. In this paper, we proposed an efficient logical clock system, called 1-n clock, which have a high compression rate compared with traditional vector clock.

In section 2 the background about parallel program execution is given. Section 3 introduces the 1-n clock system. The experiment results are shown in section 4, and the conclusion remarks will be given in section 5.

2 Parallel Program Execution Model

2.1 General Model

In traditional asynchronous distributed system, the sites are connected together with a communication network. Processes are different sites cooperate by exchanging messages and do not share any clock or memory. The message transmission time is arbitrary but finite. A parallel program execution consists of n sequential processes denoted by $P_0, P_1, \ldots, P_{n-1}$, which communicate via exchanging asynchronous messages. A parallel Program Execution (PE) in a distributed system is represented by a pair $PE = (E, \rightarrow)$, where E is a finite set of events and $\rightarrow$ represents Lamport's Happened-Before relation[9], a casual precedence relation. The execution of a process in a computation can viewed as a sequence of events with events across processes ordered by $\rightarrow$. There are two relations between events in $\rightarrow$, one is precedence, and the other is concurrency. Two events e and f are concurrent if $\neg(e \rightarrow f) \wedge \neg(f \rightarrow e)$, otherwise they have precedent relation between each other.

It is common to depict parallel programs execution using an equivalent graphical representation called a space-time diagram, illustrated in Figure 1. P_0, P_1 and P_2 are processes, and a, b, c, d, e, f are events.

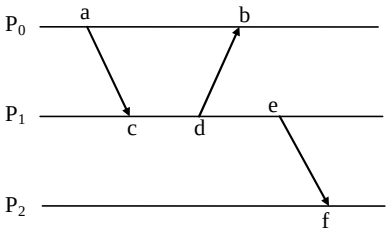

Figure1: The example of space-time diagram

2.2 Equivalent Execution

Replay must provide any number of equivalent executions based on some previously observed program execution. According to Leu et al, an equivalent execution can be defined as follows [10]:

Two executions of a process p are considered to be equivalent, if the process p receives the same information from the other processes at the same instants. The instant of an arbitrary event is defined by the interval in which only this event takes place. This means that two executions of a parallel program will be considered to be equivalent if the execution of each of its processes is equivalent

For shared variable programs it suffices to make sure that the variable reading events obtain the same value during the two executions, while for message passing programs, two executions are equivalent if all message receiving events get the same message in the two executions.

3 1-n Clock Description

3.1 Vector clock

A vector clock (VC) system is a timestamp mechanism that exactly tracks causality among events produced by a distributed computation. Vector clock is introduced in 1988[11][12]. Whether the corresponding events are or not causally related can be indicated by comparison of their vector timestamps. The vector clock is the extension of Lamport logical clock by assigning a vector of integers $VC_i[1\ldots n]$ to each process P_i, and its properties are listed following:

For two events e and f, $e \in P_i$, $f \in P_j$,

(1) If e and f has precedent relation, then the following condition is true:

$$e \rightarrow f \Leftrightarrow V_i(e)[i] \leq V_j(f)[i]$$

(2) if e and f happens concurrently, noted by $e \| f$, the following condition is true:

$$e \| f \Leftrightarrow \begin{cases} V_j(f)[i] < V_i(e)[i] \\ V_i(e)[j] < V_j(f)[j] \end{cases}$$

3.2 1-n Clock system

The most serious defect of a vector clock sytem is its scalability, for its size equals to the number of processes. This situation will affect the efficiency of the record phrase. For long-running programs, the trace files with large volume are generated. Here we propose a compound clock system, which combine logical clock (Lamport clock) and VC. Our method can compress the size of trace file sharply and the equivalent execution can be reconstructed accurately according the clock of event.

Definition 1 A non-communicational event is such an event, which it operate on local data set, neither send or receive data with events with other processes.

Definition 2 A communicational event is such an event, which will exchange data with events in other processes when its local operation is complete.

Our idea comes from the following observation: the i_{th} component of a VC corresponds to the process P_i in the execution of a parallel program P. If only a local non-communicational event happens, the value of i_{th} component of a VC just increases by one, no operation to other n-1 components. The values of other n-1 components are updated only when communicational events happen. If the local non-communicational events happen continuously, the large amount redundant clock information is stored in the trace file. Here is the problem.

Our solution is to timestamp the local non-communicational events with only one integer, and timestamp communication events with n-length vector clock. We call it 1-n clock system. Figure 2 shows a1-n clock system.

In a execution of P with n processes, let k_1 be the number of local non-communicational events, and k_2 be the number of communicational events, every clock value is store with a integer, then the Compression Rate (CR) of 1-n clock compared with traditional VC can be obtained as following:

$$CR = 1 - \frac{k_1 + nk_2}{n(k_1 + k_2)}$$

3.3 Record phase

In order to obtain these events for a particular program run, a program's source code is instrumented and re-execution is initiated. The events (and corresponding data) are stored in trace files. The data are afterwards used to determine suitable recovery lines, and to force equivalent execution of the (nondeterministic) message-passing program.

For communication events, the primary method of generating trace information does not require any changes to the user's MPI program. The user can link against the instrumented MPI library to automatically generate trace output that marks each MPI call as a phase. This library makes use of a standard MPI feature specific to profiling: each MPI routine has an alternative name characterized by a name-shift. MPI calls are replaced with wrapper routines that output trace information, and invoke MPI through the name-shifted interface. We utilize of characteristic of the PMPI library to trace the events. When a message-passing operation happened, the clock is generated. The clock is appended to the message sent to other processes. The clock system can be constructed.

For there are too many kinds of non-communicational events, we only concern limited types of events and define several corresponding macros to record its happening.

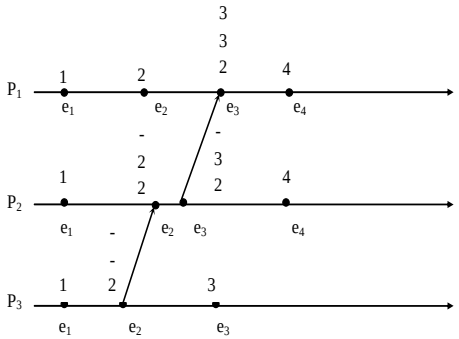

Figure2: 1-n Clock system in a Parallel Program Execution

3.4 Replay phase

During the replay phase, to construct an equivalent execution, the order of event should be assure strictly. As we have given a compress clock system, the problem which should be addressed in replay phase is how to reconstruct the traditional vector clock for 1-n clock.

In fact, the strategy is very simple. The communicational event has been time stamped by vector clock. For those non-communication events, its vector clock can be obtained by the most recent VC of communication events. For example, the VC of e_4 in P1 can be construct from e_3, the result is

(4,3,3). The clock can be reconstructed conveniently and correctly, so the equivalent execution can be obtained.

Now we will give the properties of 1-n clock system formally, 1-n clock is represented by 1NC, assume two events are e and f, and $e \in P_i$, $f \in P_j$. P_i and P_j are two processes:

1 Both e and f are non-communicational events:

(1) $e \to f \Leftrightarrow 1NC(e) < 1NC(f)$
(2) $e \| f \Leftrightarrow 1NC(e) = 1NC(f)$

2 Both e and f are communicational events
The same rule as Vector clock

3 Without loss of generality, let e be a non-communicational event and f be a communication events f,

(1) $e \to f \Leftrightarrow 1NC_i(e) < 1NC_j(f)[i]$

(2) $e \| f$, otherwise (1).

From the discussion above, we can see that the casual order can be described by the 1-n clock system. The accuracy of replaystage is proved.

4 Experiment Results

To evaluate our ideas, we analyzed execution of several message-passing programs.

First we instrument the execution of a parallel application named “nbody”, with traditional VC and our 1NC. We run “nbody”four times with iteration tmes 6, 10, 15, 20 respectively. The comparison of size of trace files generated is shown in Figure 3. We also obtain the result of other parallel applications. The comparison results are listed in Table 1

Now we will analysis these experiment results. Token Ring just passes token between processes, there is no local non-communication events, so the CR is 0. Nbody, N-Queen and Integer Sort are both iterative application, some local computation events happened, so the 1-n clock can generate the corresponding CR. The conclusion can be drawn that the 1-n clock is more appropriate for those loosely synchronization application, not for the communication intensive applications.

5 Conclusion Remarks

Execution replay is a technique developed to facilitate cyclic debugging of nondeterministic programs. The trace file can be used to force the replay of the parallel program with the same input. The size of trace file is very important to evaluate the scalability of record&replay scheme. This paper proposes an improved logic clock, called 1-n clock. By combining local logic clock and vector clock, 1-n clock can compress the size of trace file. The parallel programs can be replayed accurately based on the compressed trace file.

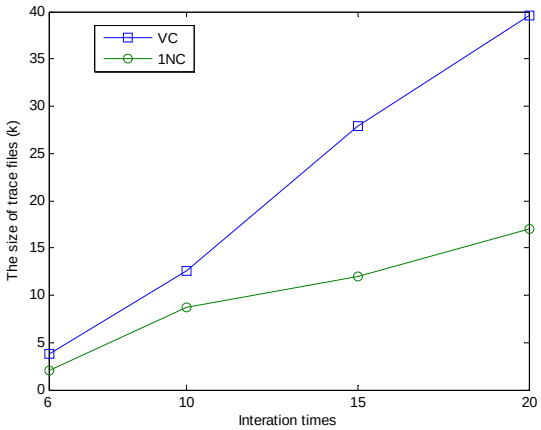

Figure3 Comparison of size of trace files of nbody using VC and 1NC

Programs(processes)	Trace file size (1-n clock)(kB)	Trace file size (traditional VC)(kB)	CR(%)
N-Queen(8)	2.5	3	16.7
IS(Integer Sort)(8)	11.2	13.9	19.4
Token Ring (16)	0.3	0.3	0

Table 1: Comparison of CR between 1-n Clock and Traditional VC

Acknowledgments

The author wishes to thank the anonymous referees for their careful reading of the manuscript and their fruitful comments and suggestions.

References:

[1] C.E. McDowell, D.P. Helmbold, Debugging Concurrent Programs, *ACM Computing Survey*, Vol.21, Issue 4, 1989, pp.593-622.

[2] R. Curtis, L. Wittie, BugNet: A Debugging System for Parallel Programming Environments, *Proc. of the 3rd Intl. Conf. on Dist. Computing Systems*, 1982, pp.394-399.

[3] E.T. Smith, Debugging Tools for Message-Based, Communicating Processes, *Proceedings 4th Intl. Conference on Distributed Computing Systems*, 1984, pp.303-310.

[4]T.J. LeBlanc, J.M. Mellor-Crummey, Debugging Parallel Programs with Instant Replay, *IEEE Transactions on Computers*, Vol.36, Issue 4, 1987, pp.471-481.

[5]C. Clemencon, J. Fritscher, R. Ruhl, Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool, *Proc. High Performance Computing Symposium, HPCS '95, Montreal, Canada*, 1995, pp. 393-404.

[6]J. Xiong, D. Wang, W. Zheng, M. Shen, Buster: An Integrated Debugger for PVM, *Proc. ICAPP '96, 2nd Intl. Conference on Algorithms and Architectures for Parallel Processing*, 1996, pp. 124-129,.

[7]F. Cornelis, A. Georges, M. Cristiaens, M.Ronsse, T. Ghesquiere, K. De Bosschere, A Taxonomy of Execution Replay Systems, *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003, CD-ROM paper 59.

[8]C. Clemencon, J. Fritscher, An Implementation of Race Detection and Deterministic Replay with MPI, *Technical Report CSCS TR-94-01*, Swiss Scientific Computing Center,1995.

[9]L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol.21, Issue 7, 1978, pp.558-565.

[10]Eric Leu, Andre Schiper, Abel Wahab Zramdini, Execution Replay on Distributed Memory Architectures, *Proc. 2nd IEEE Symposium on Parallel & Distributed Processing*, 1990, pp.106-112.

[11]C.J. Fidge. Timestamp in Message Passing Systems that Preserves Partial Ordering. *In Proc. 11th Australian Computing Conference*, 1988, pp. 56–66.

[12]F. Mattern. Virtual Time and Global States of Distributed Systems. *InCosnard, Quinton, Raynal, and Robert, editors, "Parallel and Distributed Algorithms" Conference*, 1988, pp.215–226.