Dynamically Instrumenting Message-Passing Programs Using Virtual Clocks

Kang Zhang[†] Chengzheng Sun[‡] Kei-Chun Li[†]

[†]Department of Computing, Macquarie University, Sydney 2109, Australia

[‡]School of Computing & Information Technology, Griffith University, Brisbane, Qld 4111, Australia

1 Introduction

Analysing and visualising the run-time behaviour of a distributed program requires collecting performance data during its execution. This is usually done through program instrumentation that inserts trace recording instructions into the program. The intrusiveness of inserted extra code to the original program may make the existing errors vanish, or can cause new errors to appear. It will not only change the real-time execution speed, but also change the probability of making particular non-deterministic choices. Cai and Turner proposed a logical clock approach (LCA) to control the inter-process communication [1]. It uses a logical clock in each process to reflect the real time execution of the process when running without monitoring. Not relying on any special hardware, this approach achieves minimum intrusiveness and makes the monitoring effect on the original program completely independent from the amount of time spent on monitoring activities. A main problem with LCA is that in an event of non-deterministic communication and when several processes wait on each other's logical clock to advance, a deadlock may occur. LCA uses a carrier-null message algorithm to break the deadlock situation. But sending extra null message will distort the event timing information. Therefore, LCA is only suitable for debugging rather than for performance tuning. We present a major improvement to LCA, which preserves the timing information without introducing any deadlock. Our improved approach can support both debugging and performance tuning. We call our clocks *virtual clocks* to avoid confusion with Lamport's logical clocks.

2 Instrumentation Monitor

Communication events. We identify five run-time events that capture the communication activities: Ready-To-Send (RTS), Ready-To-Receive (RTR), Non-deterministic Receive (NDR), Finished Send (FS), Finished Receive (FR), and an event that marks the termination of a process: Process Termination (PT). The events RTS and RTR/NDR (called *matching events*) of two communicating processes may arrive at different times. The difference in their arriving times is the *synchronisation delay*. The events FS and FR of two synchronous communication processes are assumed to occur at the same time after the communication. The *communication delay* measures the duration between the time when the matching events arrive and the time of their FS/FR.

Real Time and Virtual Time. Our approach uses virtual clocks to control the order of program events and hide the monitoring delay [2]. *Real time* stamps are generated by the local physical clocks. Each process executing on a processor keeps an array of *virtual time* readings. A virtual clock is maintained in each process. It ticks at the same rate as the physical clock during the execution of unmonitored portions of the program, and stops ticking during monitoring. The synchronisation and communication delays are recorded in real-time during monitoring and accumulated in the virtual time reading. We assume that the physical clocks inside multiple processors are properly synchronised by a clock synchronisation algorithm whenever needed so that there is no clock drift problem for any long running programs.

Ideally, the virtual time with monitoring should accurately reflect the real time when without monitoring. Figure 1 shows the relationship between real-time and virtual time. E1, E2, and E3 represent three execution phases which involve no monitoring activities, and thus are equivalent in both real time and virtual time. During the monitoring phases M1 and M2 when handling two communication events, the monitor captures the synchronisation and communication delays and counts them as parts of the virtual time. Let R and V be the real time and virtual time at the beginning of a new event, and R' and V' be the real time and virtual time at the end of the previous event, the above definition implies $V - V' = R - R'$.

For a non-deterministic receiving event, the monitor

decides which of the available sending processes should provide the sending event and thus allow the communication to occur. This decision is based on the virtual times of all the processes involved. In the case that the virtual time of a process which is waiting on another process is less than the virtual time of any of the available sending processes, the monitor faces a complex situation in permitting the communication. Such a situation was referred to as deadlock [1]. Consider the example in Figure 2, which shows four processes involving in a communication with non-deterministic choices. The vertical lines represent the processes' virtual times. P1 has a non-deterministic receiving event that is waiting for input from any of P2, P3, and P4. P2 is ready to send to P1, P3 is waiting to send to P2, and P4 is waiting for an input from P3. The virtual times of P3 and P4 are all less than that of P2. According to LCA, P2 will not be allowed to communicate with P1 because it is possible that either P3 or P4 may send a message to P1 at a virtual time earlier than P2's virtual time at RTS.

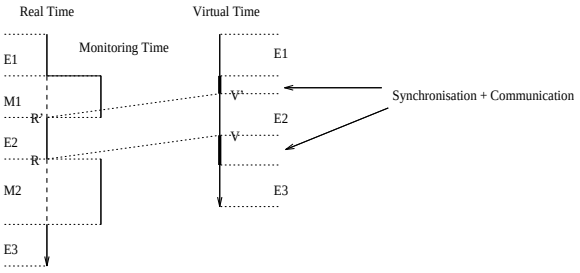

Figure 1: Real time and virtual time

We call the above interlocking condition, together with all the other cases where non-deterministic communications are permitted, as *ready condition*. Our monitoring algorithm [2] detects the ready condition, and allows the first available sending process (P2 in the example) to communicate with the NDR event. This is because that, even though the virtual times of P3 and P4 are earlier than that of P2, they are directly (in the case of P3) or indrectly (P4) waiting on P2's RTS event. Therefore, P2 is the one that is supposed to communicate with P1.

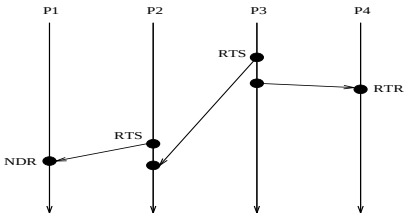

Figure 2: Non-deterministic communication

3 Experimental Results

We have implemented the monitoring algorithm on a nCube 2 multiprocessor machine and evaluated its performance by running two test programs. Each test program has a unmonitored version and a monitored version. The first test program involves a communication pattern with non-deterministic choices among eight processes. The results show that when introducing a fractional delay before the RTS event in some sending processes so that the NDR's selection choices change in the original program, the program behaves exactly the same by making the same selection choices in the monitored program. The monitoring algorithm does not introduce any deadlock.

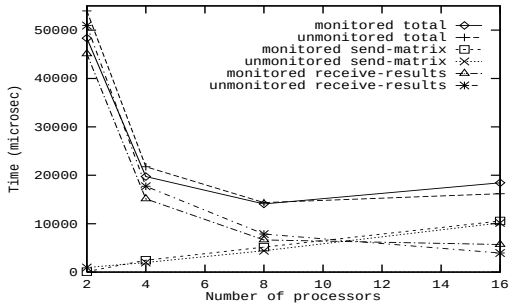

Figure 3: Divided times of the master process

The second test program is the matrix multiplication (MM) program for calculating $C[64][7] = A[64][15] * B[15][7]$. The purpose of testing this program is to evaluate the accuracy of the monitoring algorithm. The MM program consists of a master process executed on one processor, and a worker process executed on other processors for calculating the multiplication of different portions of the matrices. We measured the execution times of different parts of the master process and the worker process when 2, 4, 8, and 16 processors were used. Figure 3 shows the differences between the virtual times of the monitored version and the real times of the unmonitored version of the master process - about 4% with a maximum of 11% of the measured times. Those for the worker process are around 1.2% with a maximum of 2.6%. The differences arise mainly from the radically different ways in which the program is timed - one within the monitor and the other within the program itself.

References

[1] W. Cai and S.J. Turner, An Approach to the Run-Time Monitoring of Parallel Programs, *The Computer Journal*, 37(4), 1994, 333-345.

[2] K. Zhang, C. Sun, and K-C. Li, Collecting Timing Information While Preserving Events Ordering in Parallel Program Instrumentation, Technical Report C/TR95-07, Macquarie University, Dec. 1995.