Interlock Avoidance in Transparent and Dynamic Parallel Program Instrumentation Using Logical Clocks

W. Cai [a,1] K. Zhang [b] S. J. Turner [c] C. Sun [d]

[a] *School of Applied Science, Nanyang Technological University Singapore 639798*

[b] *Dept. of Computing, School of MPCE, Macquarie University Sydney, NSW 2109 Australia*

[c] *Dept. of Computer Science, University of Exeter Exeter EX4 4PT, U.K.*

[d] *School of Computing and Information Technology, Griffith University Brisbane, Queensland 4111 Australia*

Abstract

A fundamental problem with runtime monitoring of parallel programs is the intrusion introduced by instrumenting the original program. In order to minimize the amount of intrusion in monitoring parallel programs, the logical clock approach (LCA) was proposed. It uses logical clocks to time and control the ordering of communication events during monitoring, and to reflect the real execution behavior when running without monitoring. However, the main problem with LCA is that in the case of non-deterministic communication and when several processes wait on each other's logical clock to advance, an interlock situation may occur, where none of the processes can continue to execute. This paper presents a strategy to avoid the interlock situations, based on the concept of ready condition. How the logical clocks are updated and communications are controlled in order to maintain the ordering of events, using a relaxed communication model, will be described. Compared with the original logical clock approach, the new interlock avoidance approach is simpler and introduces less overhead. In addition, the modified logical clock mechanisms introduced in this paper are more general and applicable to a wider range of parallel computing systems.

Keywords: program instrumentation, dynamic monitoring, message-passing, intrusion, transparency, event ordering, logical clock, interlock avoidance.

---

[1] Contact Author – email: aswtcai@ntu.edu.sg; tel: 65 799 1273; fax: 65 792 6559

1 Introduction

Developing an efficient parallel program to achieve enhanced performance on a multi-processor or multi-computer system usually involves iterative processes of execution analysis and performance tuning. These can be a difficult and time-consuming process if the program being developed is relatively large and complex. Analyzing and visualizing the run-time behavior of a parallel program requires collecting information during its execution. This is usually done through program instrumentation that inserts trace recording instructions into the program. However, the intrusion 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 [7].

Several approaches to *postmortem* analysis of the intrusion have been proposed. Such approaches include the models of Malony and Reed [15,16] which adjust the trace event times to remove perturbation through event trace and execution time analysis, and the trace analysis technique of Helmbold, McDowell and Wang [11] which determines possible event orders by analyzing the trace information collected at runtime. However, postmortem approaches work well only if the program behavior has not been changed by the inserted code. *Dynamic* methods which aim at minimizing the intrusion at run-time are more promising in maintaining the original program behavior, and apparently more difficult.

One dynamic approach is to instrument a parallel program by modifying its binary image during the execution [12]. By dynamically deciding the instrumentation points and inserting lower level monitoring code into only the necessary parts of the program, such an approach can reduce the monitoring overhead considerably and make the trace file size controllable. However, the accumulated monitoring overhead, and thus the intrusion, is still dependent on the number of instrumentation points.

Gupta and Spezialetti also proposed a dynamic monitoring approach [10,20], which uses a message queue in each monitor to receive messages from other processes and decides the ordering of the messages that are available in the queue. In the case of a non-deterministic communication, an upper bound variable, `MAXWAIT`, is used to govern how long a receiving process has to wait for messages from the relevant sending processes. The value of `MAXWAIT` is determined by an estimation of the total intrusion overhead caused by the monitoring activities. When this value is inaccurate, the waiting time may not be long enough for all the relevant sending processes to send in their messages. In this case, the monitoring process needs to be redone after a proper adjustment of `MAXWAIT`.

In [5], Dodd and Ravishankar used a dedicated monitor processor to process monitoring data collected. In their system, the monitoring code is always running, thus, the interference is the same during normal execution as during program development. In addition, the monitoring interference in their system is low and predictable so that there is a minimum perturbation on the program's performance.

Cai and Turner proposed a *logical clock approach* (LCA) which uses logical clocks to control the inter-process communication [4]. The basic idea of LCA is to introduce a *logical clock* for each process that reflects the real time execution of that process when running without monitoring. Not relying on any special hardware, this approach achieves minimum intrusion and makes the monitoring effect on the original program completely independent from the amount of time spent on monitoring activities. Since the time information obtained using logical times reflects the real execution behavior, the logical clock approach can be used for both on-the-fly debugging and performance analysis. But, a main problem with LCA is that in the case of non-deterministic communication and when several processes wait on each other's logical clock to advance, an *interlock* situation in which none of the processes is able to execute may occur [2]. LCA uses a *carrier-null message algorithm* [1], originally developed for conservative parallel simulation [9], to break the interlock situation. However, in addition to the large communication overhead caused by sending extra protocol messages [2,19], the integration of the carrier-null message algorithm with logical clock updating and communication control mechanisms also makes the implementation of LCA more complicated [3].

This paper presents a new strategy that avoids the interlock situations in transparent and dynamic parallel program instrumentation using logical clocks. The new interlock avoidance strategy is based on the concept of *ready condition* [21]. In Section 4, the ready condition will be defined. We will prove that a non-deterministic communication can be committed safely, without violating the ordering of communication, once a ready condition is reached; and that the ready condition can be always reached.

The logical clock updating and communication control in the original logical clock approach are heavily influenced by the transputer architecture [17]. For instance, the hand-shaking synchronous communication model is assumed, in which message-passing only happens when both communicating partners are ready to communicate; and in selecting a communicating partner for a non-deterministic receive event, the process which becomes ready to send first is selected. These are in general not true for other parallel computing systems. In this paper, the logical clock updating and communication control mech-

[2] To avoid any confusion with possible deadlocks existing in a message-passing program itself, we refer to such a situation as *interlock* rather than *deadlock*.

anisms are modified for a more relaxed communication model. In Section 2, system assumptions of the communication model we adopt in this paper will be described. The modified logical clock mechanisms will then be introduced in Section 3. The interlock situation caused by the logical clock approach will also be described. As has been mentioned above, Section 4 will cover the new integrated interlock avoidance strategy based on the concept of ready condition. Section 5 will discuss briefly various implementation issues. Finally, Section 6 will conclude the paper with some suggestions on future work.

2 System Assumptions

We assume that the system under discussion consists of a number of processing nodes connected via either a network or a dedicated switch. There is no global shared memory in the system, each node can only access its own local memory. There is also no global clock in the system: each node has its own clock. We further assume that clocks of all processing nodes in the system are properly synchronized by a clock synchronization algorithm (e.g. [18]) so that there is no clock drift problem for any long running programs.

The programming paradigm is the *message-passing* programming model. A message-passing parallel program consists of two or more processes. Each process consists of a sequence of events and can communicate with other processes through message-passing. An event represents an atomic unit of computation meaningful to the programmer. This may be the execution of a single machine instruction or an entire procedure. The level of granularity is irrelevant [6].

A modified *space-time diagram* [13] is used to represent the computation in a message-passing parallel program (see Figure 1). The horizontal direction represents space (i.e., processes), and the vertical direction represents time which flows downwards. The dots denote events and the vertical lines denote processes. The dotted directed lines between vertical lines denote message-passing (that is, from the point where the message is sent to the point where the message arrives), and the solid directed lines between vertical lines denote the committed communication (that is, from the point where the message is sent to the point where the message is finally received).

We also assume that *synchronous* communication is the only medium for inter-process communication in the system. However, there is no hand-shaking requirement, that is, the message-passing can happen even if the receiving process is not ready yet to receive the message. The message will be sent to the receiving process immediately after the `send` routine is called, and the message will be temporarily buffered by the system if the receiving process has not yet issued the corresponding `receive` call when the message arrives. But,

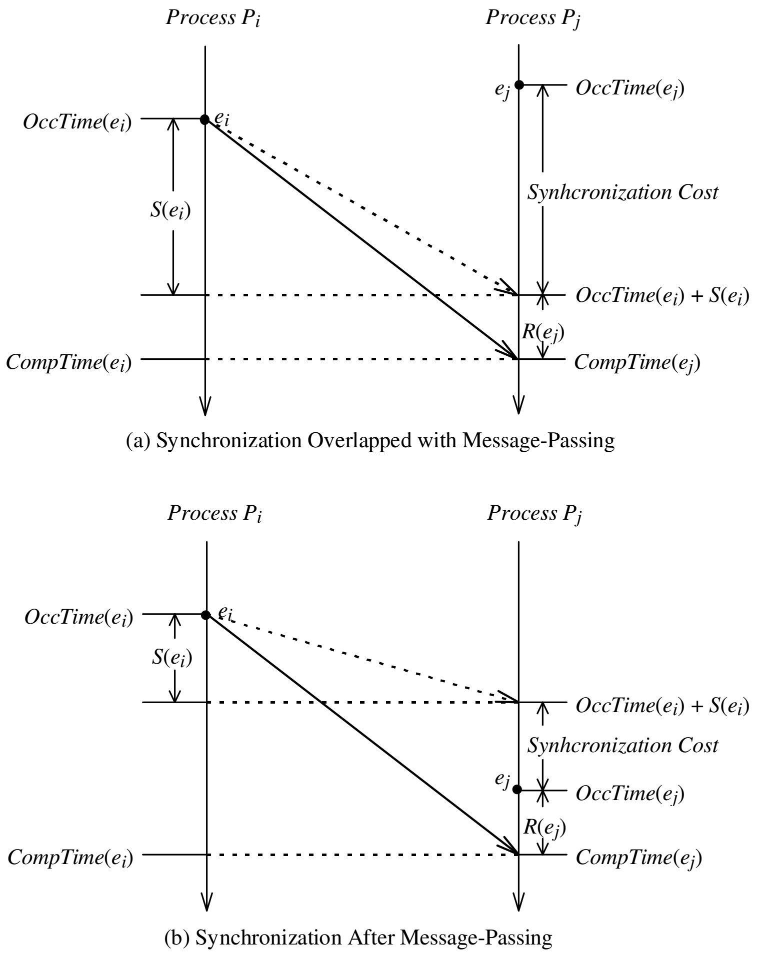


Fig. 1. Synchronization and Communication Delay

the sending process can only proceed to execute the next instruction after the receiving process calls the corresponding `receive` routine and receives the message. We say that a communication is committed only after the message has been received.

The following three communication routines are supported in the system:

Send which sends a message to a specified process. This primitive will not return until the receiving process has received the message.
Receive which receives a message from a specified process. Similar to the send, this primitive will not return until a message has been received.
Receive Any which receives a message from any of the other processes. The primitive will not return until the first available message from a sending process has been received.

We assume that `receive any` is the only source of non-determinism in our parallel programs, and that it will receive from the process whose message arrives first rather than the process that sends first.

To illustrate the above specifications, let us consider the case where event e_i of process P_i sends a message to process P_j, which is received by event e_j (see Figure 1). Suppose that $S(e_i)$ is the elapse time for sending a message from P_i to P_j, and that $R(e_j)$ is the elapse time for P_j to receive the message. If $OccTime$ represents the time at which an event happens and $CompTime$ represents the time at which an event completes, based on the above communication model assumption, we can have:

$$\begin{aligned} & CompTime(e_i) \\ = & CompTime(e_j) \\ = & \max(OccTime(e_i) + S(e_i), OccTime(e_j)) + R(e_j) \end{aligned} \tag{1}$$

If the receiving process issues the `receive` call before the message arrives (that is, $OccTime(e_i) + S(e_i) \geq OccTime(e_j)$, see Figure 1(a)), the delays experienced by the sending and receiving processes are

$$\begin{aligned} & S(e_i) + R(e_j) \ and \\ & (OccTime(e_i) + S(e_i) + R(e_j)) - OccTime(e_j) \end{aligned}$$

respectively. In this case, the receiving process needs to wait for the message to arrive, we say that the synchronization cost is $(OccTime(e_i) + S(e_i)) - OccTime(e_j)$. Otherwise, the delays are

$$\begin{aligned} & (OccTime(e_j) + R(e_j)) - OccTime(e_i) \ and \\ & R(e_j) \end{aligned}$$

for sending and receiving processes respectively. The arrived message will not be received by the receiving process immediately, and the synchronization cost is $OccTime(e_j) - (OccTime(e_i) + S(e_i))$ (see Figure 1(b)).

In a practical message passing system, an acknowledgment message will need to be sent from the receiving process to the sending process to indicate that the message has been received. This acknowledgment will take some time to reach the sending process, so the sending process may complete its send event after the receiving process. Thus, the actual completion time of the send event should be equal to the completion time calculated using the above formula plus the duration of transmitting the acknowledgment message. However, for the sake of simplicity, the acknowledgment message will be ignored in the following discussion.

The synchronous communication model described above is more general than the hand-shaking synchronous communication model in which synchronization must happen before message-passing. The model is meant for illustrating the principles involved. In fact, the logical clock approach can be easily extended to monitor other message-passing parallel computing systems, for example, the one based on asynchronous communications.

3 Logical Clock Approach

The logical clock approach is based on the concept of maintaining the correct ordering of non-deterministic communication events. Section 3.1 will briefly explain why it is necessary to preserve the ordering of non-deterministic communications; and Section 3.2 will show that how it can be done by keeping a logical clock for each process.

3.1 Event Ordering & Transparent Monitoring

In a message-passing parallel program, events can be ordered by the *happen-before* relation [13] defined as follows:

Definition 1 *Happen-Before Relation* (written as $\rightarrow$):

Given two events, e_i and e_j, $e_i \rightarrow e_j$ iff

- e_i and e_j are events in the same process, and e_j is the next event of e_i; or
- e_i is the sending of a message by one process and e_j is the receipt of the same message by another process; or
- $\exists$ events e_{l1}, ..., e_{lk}, so that there is a precedence chain $e_i \rightarrow e_{l1} \rightarrow ... \rightarrow e_{lk} \rightarrow e_j$.

For example, in Figure 1, the message sent by event e_i was received by event e_j. So, we have $e_i \rightarrow e_j$.

It is clear that the happen-before relation defines a partial order of all events in a message-passing parallel program [13]. So, a message-passing parallel program will have the same execution behavior provided that the partial order defined by the happen-before relation is preserved [6,8]. The execution ordering of events which are not related by the happen-before relation is not important.

The basic building blocks of a message-passing program are the sequential processes in the program, and it is the inter-process communication that achieves the coordination among the processes to perform parallel processing. Thus, the partial order of all events will be preserved provided that the ordering of communication events is kept unchanged [4].

According to the communication primitives described in the last section, we can identify the following communication events: *send event* (`SD`), *receive event* (`RV`) and *receive-any* (or *non-deterministic receive*) event (`NDR`). Furthermore, we define *matching events* as follows:

Definition 2 *Matching Events*:

> Two events are called *matching events*, if one is an `SD` event that sends a message and the other is either an `RV` or an `NDR` event that receives the message.

Since non-deterministic receive is the only source of non-determinism, if we can guarantee that for each `NDR` event it will receive the message from the same `SD` event both with and without instrumentation, the ordering of communication events will be preserved. For example, Figure 2(a) depicts a communication ordering of three processes without instrumentation, where events b_1 and b_2 of process B are `NDR` events:

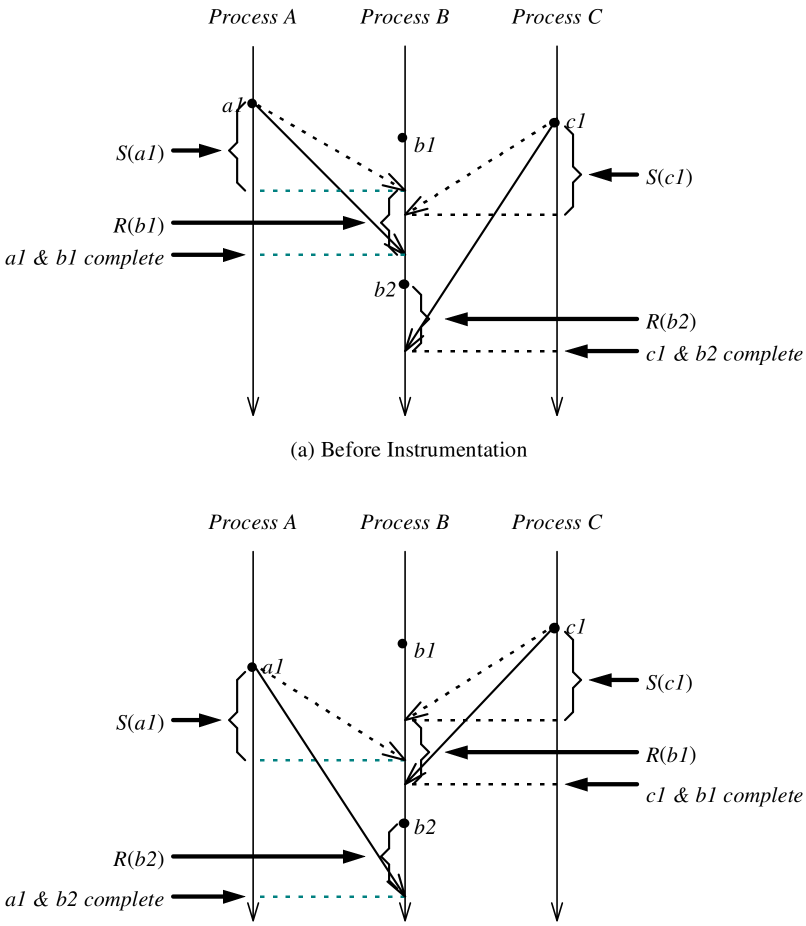
$$a_1 \rightarrow b_1;$$
$$c_1 \rightarrow b_2; \; and$$
$$b_1 \rightarrow b_2$$

If we can guarantee that when the program is executed under monitoring the two `NDR` events will still communicate with the same `SD` event, that is, b_1 receives from a_1 and b_2 receives from c_1, the above ordering of communication events will be kept unchanged. Thus, the same execution behavior would be observed no matter how the program is executed.

The objective of LCA is to preserve the execution behavior of a parallel program that is under monitoring. In other words, it tries to dynamically minimize the intrusion of execution behavior caused by program instrumentation. Therefore, to achieve this objective, we only need to guarantee that for each `NDR` event, it will receive the same message from the same matching event whether or not the program is instrumented. Hence, it is only necessary to instrument communication events.

3.2 Logical Clock Mechanisms

When a program is executed under monitoring, the instrumentation code may delay the occurrence of communication events. Thus, it is likely that, for example, the message sent by event c_1 arrives before the one sent by event a_1 as shown in Figure 2(b). As a result, the ordering of the communication events is changed (that is, b_1 now receives from c_1 and b_2 receives from a_1), and different execution behavior observed.



(a) Before Instrumentation

(b) Process *A* Instrumented

Fig. 2. Example of Communication Ordering

To preserve the ordering of communication events, we introduce a logical clock for each process, which ticks only when an event of the process is executed and is switched off when any monitoring activity (or instrumentation code) executes. We try to make sure that any event of a process that happens at time T without instrumentation will happen at logical time T with instrumentation [4]. Using the example in Figure 2(b), although physically the message from process C may arrive first when the program is instrumented, the logical arrival time of the message from process A will still be earlier than that of the message from process C. This is because the logical time does not include the delays incurred by monitoring activities. So, process B would still communicate with process A first. The ordering of communication events will be kept unchanged, and the same program execution behavior will be observed.

To enforce the above, the instrumentation system (or the *monitor*) needs to decide which communication can be committed (or permitted) and to update the logical clocks after the communication has been performed. Suppose that for an instrumented program $OccRT$ and $CompRT$ are the real time clock readings upon event occurrence and completion. $OccLT$ and $CompLT$ are the corresponding logical clock readings. A logical clock of a process P_i is written as $LC(P_i)$. Assuming that e_{i-1} and e_i are two consecutive instrumented events, our definition of logical clock implies that:

$$OccRT(e_i) - CompRT(e_{i-1}) = OccLT(e_i) - CompLT(e_{i-1})$$

since there is no monitoring activity between the completion of event e_{i-1} and the start of event e_i. All events occuring between e_{i-1} and e_i are not instrumented, so the logical elapse time is equal to the real elapse time during this period. Therefore, when event e_i happens, the logical clock of P_i, $LC(P_i)$, is set to:

$$OccLT(e_i) = CompLT(e_{i-1}) + (OccRT(e_i) - CompRT(e_{i-1})) \tag{2}$$

If e_i is P_i's first instrumented event, $LC(P_i)$ will be initialized to:

$$OccLT(e_i) = OccRT(e_i) - real\ clock\ value\ when\ program\ starts \tag{3}$$

The logical clock should advance during synchronization and communication. When an instrumented event completes, the logical clock needs to be updated again. Based on equation 1, we know that upon the completion of event e_i the logical clock of P_i needs to be set to:

$$CompLT(e_i) = \begin{cases} \max(OccLT(e_i) + S(e_i), OccLT(e_j)) + R(e_j), \text{ if } e_i \text{ is an SD event} \\ \max(OccLT(e_j) + S(e_j), OccLT(e_i)) + R(e_i), \text{ otherwise} \end{cases} \tag{4}$$

where e_j is the matching event of e_i, $S()$ represents the duration of sending a message and $R()$ represents the duration of receiving a message.

Figure 3 shows how a communication event is instrumented. The vertical line of real time represents the instrumented behavior of process P_i, where sections of solid lines denote the period when the logical clock should be switched on. Time on this line is obtained from reading the real time clock. Dotted vertical lines represent the instrumentation overheads, which are not included in the logical time (that is, the logical clock is switched off during such periods). The time on the vertical line representing logical time is obtained using the logical clock updating formulas described above (i.e., equations 2, 3 and 4). Clearly, we can see that the vertical line of logical time represents the original behavior of process P_i.

The decision to commit a deterministic communication can be easily made: if events `SD` and `RV` are matching events and the message has already arrived, the message sent by the `SD` event can then be received and the communication committed (see Figure 1). The problem is the handling of the non-deterministic

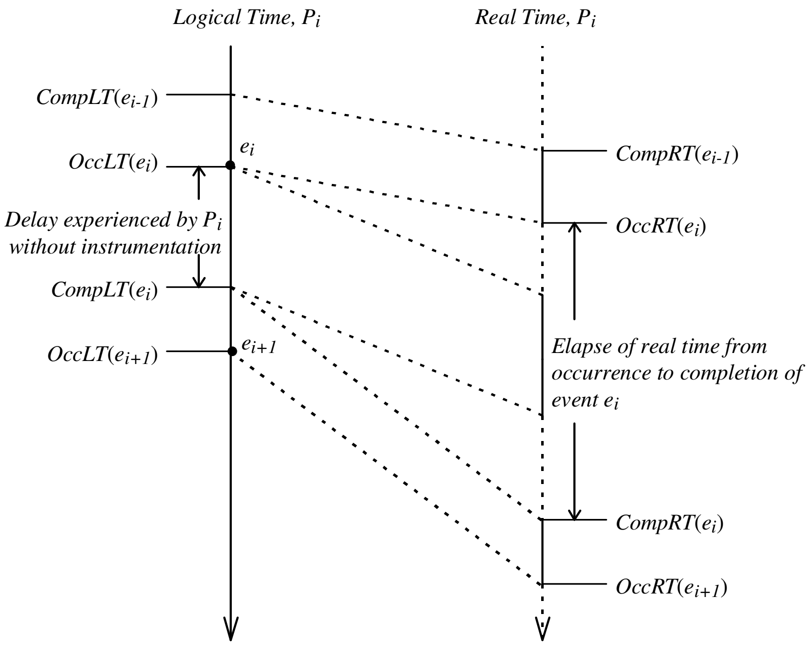


Fig. 3. Instrumentation of a Communication Event

receive events. When some processes are running behind in logical time, they should still be considered as the potential communication partners of the NDR event under consideration. Assuming that the current event of a process is the event being monitored, we define a *ready sending process* of an NDR event as follows:

Definition 3 *Ready Sending Process*:

> Suppose the current event of process P_i is an NDR event e_i that will receive a message from any other process. A *ready sending process* of P_i is defined as a process which has sent a message to P_i and is currently waiting for the completion of the SD event. The message it sent has already arrived at P_i, and is referred to as a *ready sending message* or *ready message*. The set of all ready sending processes of process P_i is denoted as N_{R_i}.

For an NDR event we have to decide which of the ready sending processes should provide the matching sending event and thus allow the communication to occur. This decision is based on the logical times of all the processes involved. In our communication model, the message that arrives first should be received. This is different from the original LCA that permits communication between an NDR event and an SD event of the process that is ready to send first [4]. Thus, the principle used to commit non-deterministic communication is as follows:

> *The communication between an* NDR *event and an* SD *event of the ready sending process whose message arrived at the earliest logical time* T *can be*

committed, if and only if there are no processes that can still send a message that may arrive at a logical time earlier than T.

If the logical time of a non-ready sending process is earlier than the earliest logical arrival time of the ready sending messages, there is still a possibility that this process may send a message that arrives earlier in logical time than any existing ready messages. So, it is not safe to commit the non-deterministic communicaton. Therefore, in the case that the logical time of a process, which is waiting on another process, is less than the logical arrival time of any of the ready messages, we may face a complex situation in permitting the communication. A simple policy, that permits the communication only when the logical times of all the non-ready sending processes are greater than the earliest logical arrival time of the ready messages, may lead to a situation where none of the processes is able to proceed.

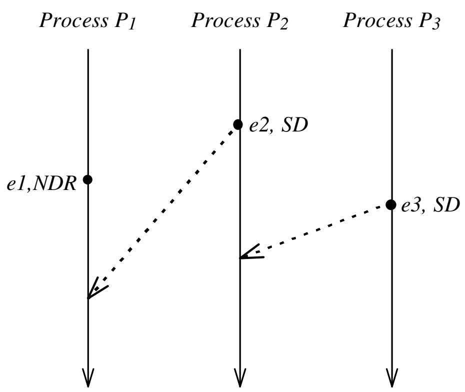


Fig. 4. Interlock Situation in Logical Clock Approach

Consider the example in Figure 4, which shows three processes involved in a non-deterministic communication. The vertical lines represent processes' logical times. Directed dotted lines between the vertical lines are used to represent message-passing as has been already described in Section 2. P_1 has an `NDR` event that is waiting for input from either process P_2 or process P_3. Process P_3 has sent a message to $P2$ and is waiting for the communication to commit. Its current logical time is the occurrence time of the `SD` event. Assuming that process P_2 has sent a message to P_1 and the message it sent has already arrived, so, according to the definition, P_2 is a ready sending process of P_1. The message it sent is ready to be received. Considering the `NDR` event of P_1, from Figure 4, it is clear that the logical time of P_3 is earlier than the logical arrival time of the ready message. In this case, using the simple policy, the communication between P_1 and P_2 is not allowed to commit because P_3's logical time is earlier than the logical arrival time of the ready message. A situation, in which none of the processes is able to proceed, occurs: P_3 is waiting for P_2 to receive its message; P_2 is in turn waiting for P_1 to receive its message; and P_1 is waiting for P_3 to advance its logical clock. In this paper, in order to avoid any confusion with possible deadlocks existing in a message-passing program itself, we refer to such a situation as *interlock*.

Definition 4 *Interlock*:

> Interlock refers to the situation where none of the processes can continue to execute because the monitor is unable to permit a non-deterministic communication. The processes involved in an interlock are waiting for either the completion of communication or the advance of logical times of other processes.

However, if we consider the above example carefully we can find that since process P_3 is waiting for P_2 to make progress, it in fact cannot send messages to process P_1 unless P_1 receives the message from process P_2 and commit the communication. So, if we can detect this condition, interlock situations in the logical clock approach can be avoided.

4 Interlock Avoidance

In this section, we will define the *ready condition* and prove that a non-deterministic communication can be committed once the ready condition is reached and that the ready condition can always be reached. Consequently, interlocks can always be avoided. Without losing generality, as has been mentioned in the last section, only the communication events, that is, `SD`, `RV` and `NDR` events, are considered in the following.

The definition of the ready condition is based on the concept of *wait-for* relation between processes. So, in the following, the wait-for relation and its properties will be introduced first.

Definition 5 *Wait-For Relation* (written as $\Rightarrow$):

Given two processes, P_i and P_j ($i \neq j$), $P_i \Rightarrow P_j$ iff:

- P_i either has sent a message to or is waiting to receive a message from P_j (that is, P_i has executed either an `SD` or an `RV` event), but P_j has not yet executed the corresponding matching event (that is, either `RV`, `NDR` or `SD` event); or
- $\exists$ processes P_{l_1}, ..., P_{l_k} so that there is a waiting chain $P_i \Rightarrow P_{l_1} \Rightarrow ... \Rightarrow P_{l_k} \Rightarrow P_j$.

If $P_i \Rightarrow P_j$, we say that P_i is waiting for P_j (or P_i is blocked by P_j). Obviously, the wait-for relation is transitive, that is, if $P_i \Rightarrow P_j$ and $P_j \Rightarrow P_k$, then $P_i \Rightarrow P_k$. Since our communication model is synchronous, based on the transitive property of the wait-for relation and the inequality $\max(a, \max(b, c)) \geq$

$\max(a, c)$, we can have the following lemma:

Lemma 1 *Suppose that the event currently being executed by P_i is e_i and the event currently being executed by P_j is e_j. If $P_i \Rightarrow P_j$, then the completion time of e_i is at least $max(OccLT(e_i), CompLT(e_j))$.*

If $P_i \Rightarrow P_i$, since we do not consider a process sending messages to itself, there must be a cyclic waiting chain involving more than two processes. According to the definition of the wait-for relation, if $P_i \Rightarrow P_j$, it means that P_i has executed either an `SD` or an `RV` event and is now waiting for P_j to proceed so that the communication can be eventually completed. It is clear that if there is a cyclic waiting chain of processes, none of the processes which are not in the waiting chain will communicate with any process in the waiting chain. Thus, the waiting cycle cannot be broken and processes in the waiting chain are deadlocked. This deadlock is inherent in the original program itself and is not caused by the logical clock approach. It will occur even if the program is not instrumented.

Lemma 2 *If there is a cyclic waiting chain, that is, $\exists$ a process P_i and $P_i \Rightarrow P_i$, then there will be a deadlock in the program itself involving all the processes in the waiting chain.*

Given a set of processes N, we denote the set of processes that are blocked by any process of N as N^-.

Definition 6 *Set of Blocked Processes of N* (written as N^-):

Given a set of processes N, N^- is defined as follows: for any process $P_i \in N^-$
- $P_i \notin N$; and
- $\exists P_k \in N$ so that $P_i \Rightarrow P_k$.

Obviously, $N \cap N^- = \emptyset$.

Now, we can define the ready condition as follows:

Definition 7 *Ready Condition*:

Given
- a set of processes N;
- a set of processes $N_{\mathtt{NDR}} \subset N$, such that $\forall P_j \in N_{\mathtt{NDR}}$, the event it is currently waiting for completion is an `NDR` event e_j;
- a set of processes $N_R \subset N$, such that $\forall P_j \in N_R$, $\exists P_k \in N_{\mathtt{NDR}}$ and $P_j \in N_{R_k}$, where N_{R_k} is the set of P_k's ready sending processes;
- for each process $P_j \in N_{\mathtt{NDR}}$, a set of processes $N_{L_j} \subset N$, such that $\forall P_k \in N_{L_j}$, $LC(P_k) > \min\{LC(P_t) + S(e_t) | P_t \in N_{R_j}\}$, where e_t is the SD event that process $P_t \in N_R$ is currently waiting for completion;

– a set of terminated processes $N_T \subset N$;

for process $P_i \in N_{\texttt{NDR}}$, e_i and N_{R_i} are said to be in a ready condition iff:
– N_{R_i} is not empty;
– $\forall P_j \in N_{\texttt{NDR}}$, $j \neq i$, $\max(LC(P_j), \min\{LC(P_t) + S(e_t) | P_t \in N_{R_j}\}) \geq \min\{LC(P_t) + S(e_t) | P_t \in N_{R_i}\}$; and
– $(N_{\texttt{NDR}} \cup N_R \cup N_{L_i} \cup N_T) \cup (N_{\texttt{NDR}} \cup N_R \cup N_{L_i} \cup N_T)^- = N$

Theorem 1 *Given an* NDR *event e_i that process P_i is currently executing and an* SD *event e_j that process P_j is currently waiting for completion, where $P_j \in N_{R_i}$ and $LC(P_j) + S(e_j) = \min\{LC(P_t) + S(e_t) | P_t \in N_{R_i}\}$, e_i should communicate with e_j if e_i and N_{R_i} are in the ready condition. Conversely, the ready condition will hold if the communication should be committed.*

Proof We first prove the first part of the theorem, that is, P_i can communicate with P_j if e_i and N_{R_i} are in the ready condition.

To prove P_i can communicate with P_j, we just need to prove that for any process $P_k (k \neq i, j)$, it will not send any message to P_i that arrives at a logical time earlier than $LC(P_j) + S(e_j)$.

– Clearly P_i will not receive any message from process P_k if $P_k \in N_T$ or $(P_k \Rightarrow P_s) \wedge (P_s \in N_T)$.
– From Lemma 1, it is also obvious that P_i will not receive any message at a logical time earlier than $LC(P_j) + S(e_j)$ from process P_k if $P_k \in N_{L_i}$ or $(P_k \Rightarrow P_s) \wedge (P_s \in N_{L_i})$.
– Because of the last condition of the ready condition, any other process P_k that is not included in the above cases must be in one of the following situations:
  · $P_k \in N_{\texttt{NDR}} \cup N_R$; or
  · $\exists\ P_s \in N_{\texttt{NDR}} \cup N_R$ and $P_k \Rightarrow P_s$.

  From the second condition of the ready condition, we know that for any process P_k, $(P_k \in N_{\texttt{NDR}} \cup N_R) \wedge (P_k \notin N_{R_i})$ it will not complete the communication earlier than $LC(P_j) + S(e_j)$. For $P_k \in N_{R_i}$, obviously, the logical arrival time of its message is no earlier than $LC(P_j) + S(e_j)$ since $LC(P_j) + S(e_j) = \min\{LC(P_t) + S(e_t) | P_t \in N_{R_i}\}$. So, we have proved that $\forall P_k$, $P_k \in N_{\texttt{NDR}} \cup N_R$, P_k will not send a message to P_i that arrives at a logical time earlier than $LC(P_j) + S(e_j)$. In addition, from Lemma 1, we know that any process P_s, $(P_k \Rightarrow P_s) \wedge (P_s \in N_{\texttt{NDR}} \cup N_R)$, will also not send a message to P_i that arrives at a logical time earlier than $LC(P_j) + S(e_j)$.

So, in summary, there is no process that can send a message which arrives at P_i at a logical time earlier than $LC(P_j) + S(e_j)$.

We now prove the second part of the theorem. Since P_i is allowed to communicate with P_j, P_i will not receive any message from any other process at a logical time earlier than $LC(P_j)+S(e_j)$. To prove the ready condition is met, we only need to prove that the three conditions of the ready condition are satisfied.

- It is obvious that N_{R_i} must not be empty.
- Clearly, $\forall P_k \in N_{\mathtt{NDR}}$, $k \neq i$, $\max(LC(P_k), \min\{LC(P_t)+S(e_t)|P_t \in N_{R_k}\})$ must be at least $LC(P_j)+S(e_j)$. Otherwise, if $\exists P_s \in N_{\mathtt{NDR}}$, $s \neq i$ and $\max(LC(P_s), \min\{LC(P_t)+S(e_t)|P_t \in N_{R_s}\}) < LC(P_j)+S(e_j)$, there is a possibility that P_s may send a message that arrives at P_i at a logical time earlier than $LC(P_j)+S(e_j)$. Recalling the principle used to commit non-determinstic communication in LCA, the non-deterministic communication between P_i and P_j cannot be committed, because the monitor cannot guarantee that there will be no more messages that may arrive at an earlier logical time than $LC(P_j)+S(e_j)$. This contradicts the assumption.
- If $\exists$ a process $P_s \in N$ but $P_s \notin (N_{\mathtt{NDR}} \cup N_R \cup N_{L_i} \cup N_T) \cup (N_{\mathtt{NDR}} \cup N_R \cup N_{L_i} \cup N_T)^-$, then P_s must
  - · not be involved in a non-deterministic communication [3];
  - · not be terminated;
  - · have an earlier logical time than $LC(P_j)+S(e_j)$; and
  - · not be blocked by any process in the set $N_{\mathtt{NDR}} \cup N_R \cup N_{L_i} \cup N_T$.

  So, there is a possibility that P_s may send a message to P_i that arrives at an earlier logical time than $LC(P_j)+S(e_j)$. Thus, if process P_i is allowed to communicate with P_j, the last condition of the ready condition must also be satisfied.

Therefore, all the three conditions are satisfied. e_i and N_{R_i} are in ready condition. □

So, the strategy to avoid interlocks is to commit a non-deterministic communication once a ready condition is reached. To detect whether or not a ready condition is reached, we need to test if all the three conditions of the ready condition are satisfied. The first condition can be easily tested. To test the second and the third conditions, the state and logical time information about all processes will be required. A brief discussion about various implementation issues can be found in Section 5.

Consider the example shown in Figure 4 again. P_1 is currently blocked on an `NDR` event e_1. P_2 has sent a message to P_1 and is now waiting for the completion of the `SD` event e_2. The message it sent has already arrived at P_1, so P_2 is a ready sending process of P_1. P_3 has sent a message to P_2 and is now waiting for the completion of the `SD` event e_2. Hence, we have

- $P_1 \in N_{\mathtt{NDR}}$;

[3] That is, P_s is not waiting for the completion of either an `NDR` event or an `SD` event that sends a message to an `NDR` event.

– $N_{R_1} = \{P_2\}$; and
– $P_3 \Rightarrow P_2$.

All the three conditions of the ready condition are satisfied for e_1 and N_{R_1}. Therefore, the communication between e_1 and e_2 can be committed.

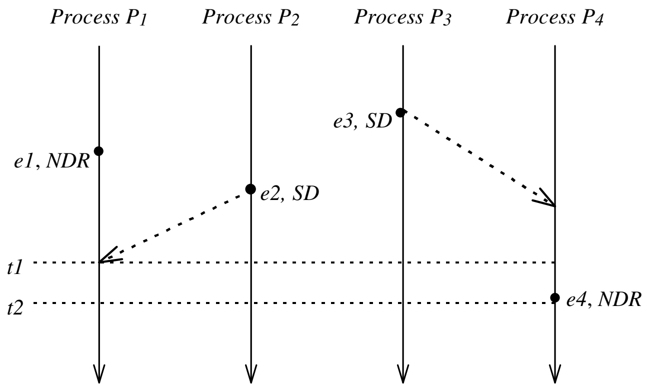


Fig. 5. Ready Condition Involving Two NDR Events

It is also possible that more than one non-deterministic communication can be committed at the same time. For example, consider the situation shown in Figure 5, where process P_1 is waiting for the completion of the NDR event e_1; process P_4 is also waiting for the completion of an NDR event (i.e., e_4); and processes P_2 and P_3 are the only ready sending process for P_1 and P_4 respectively (that is, $N_{R_1} = \{P_2a\}$ and $N_{R_4} = \{P_3\}$). It can be seen that for both e_1 & N_{R_1} and e_4 & N_{R_4} the ready condition is met. Since $t_1 = \max(LC(P_1), LC(P_2) + S(e_2)) > LC(P_3) + S(e_3)$ and $t_2 = \max(LC(P_4), LC(P_3) + S(e_3)) > LC(P_2) + S(e_2)$, the permission of one non-deterministic communication will not affect the choice of communication partner of the other non-deterministic communication.

Theorem 2 *Given an* NDR *event e_i that process P_i is currently executing and a set of ready sending processes N_{R_i}, a ready condition can always be reached if there are no deadlocks in the uninstrumented program itself.*

Proof To prove the ready condition can always be reached, we only need to prove that the three conditions of the ready condition will be eventually satisfied. We prove this in two steps: we first prove that given a set of processes which are waiting for an NDR event to complete, there must be at least one process for which the ready condition is reached; then prove that given any process P_i in $N_{\texttt{NDR}}$, the ready condition will be reached for this process.

Step 1: Given a set of NDR processes, i.e., $N_{\texttt{NDR}}$, there must be a process $P_q \in N_{\texttt{NDR}}$ such that

$$\forall P_j \in N_{\texttt{NDR}},\ j \neq q,$$

$\max(LC(P_j), \min\{LC(P_t) + S(e_t) | P_t \in N_{R_j}\}) \geq$
$\min\{LC(P_t) + S(e_t) | P_t \in N_{R_q}\}$

We prove that the ready condition for this process will be reached.

- N_{R_q} is obviously not empty.
- The second condition is also obviously satisfied for P_q.
- For any process P_k, $P_k \notin N_{\mathtt{NDR}} \cup N_R \cup N_T$, if it is not yet blocked by some other processes, it may advance so that eventually:
  · its logical clock passes the logical time $\min\{LC(P_t)+S(e_t)|P_t \in N_{R_q}\}$ and hence P_k becomes a member of N_{L_q};
  · it becomes a member of the set $N_{\mathtt{NDR}} \cup N_R \cup N_T$; or
  · it is blocked by a process P_s.

  In the case where process P_k becomes blocked by P_s, if $\exists$ a process $P_l \in N_{\mathtt{NDR}} \cup N_R \cup N_T$ so that $P_s \Rightarrow P_l$, then since the wait-for relation is transitive, we know that $P_k \Rightarrow P_l$; otherwise, since $s \neq k$ (because of no deadlocks in the uninstrumented program and Lemma 2), we can apply the same analysis again on process P_s. So, in summary, for any process P_k, $P_k \notin N_{\mathtt{NDR}} \cup N_R \cup N_T$, it will eventually satisfy one of the following conditions:
  · becomes a member of N_{L_q};
  · becomes a member of $N_{\mathtt{NDR}} \cup N_R \cup N_T$;
  · $\exists$ a process $P_s \in N_{L_q}$ so that $P_k \Rightarrow P_s$; or
  · $\exists$ a process $P_l \in N_{\mathtt{NDR}} \cup N_R \cup N_T$ so that $P_k \Rightarrow P_l$.

  The last condition of the ready condition is also eventually satisfied for this process.

Hence, the ready condition will be reached by this process P_q.

Step 2: Any process in $N_{\mathtt{NDR}}$ that has reached the ready condition will be removed from $N_{\mathtt{NDR}}$, so that for any process $P_i \in N_{\mathtt{NDR}}$, the condition

$\forall P_j \in N_{\mathtt{NDR}},\ j \neq i,$
$\max(LC(P_j), \min\{LC(P_t) + S(e_t) | P_t \in N_{R_j}\}) \geq$
$\min\{LC(P_t) + S(e_t) | P_t \in N_{R_i}\}$

will become true eventually. Note that a process that is removed from $N_{\mathtt{NDR}}$ may execute another `NDR` event and be added to $N_{\mathtt{NDR}}$ again (but the new `NDR` event will have a later logical time).

Therefore, the ready condition can be reached by an `NDR` event of an any given process P_i and its corresponding set of ready sending processes N_{R_i}. $\square$

From Theorems 1 and 2, we know that a non-deterministic communication can be committed once a ready condition is reached and that the ready condition can always be reached. This implies that interlock situations, where none of the processes is able to proceed due to the control of making a non-deterministic choice, will no longer occur during program monitoring using the logical clock approach. So, our strategy of committing non-deterministic communication once a ready condition is reached will avoid interlocks in parallel program monitoring using logical clocks.

Corollary 1 *The strategy of committing non-deterministic communication once a ready condition is reached will avoid interlocks in parallel program monitoring using the logical clock approach.*

5 Discussion on Implementation

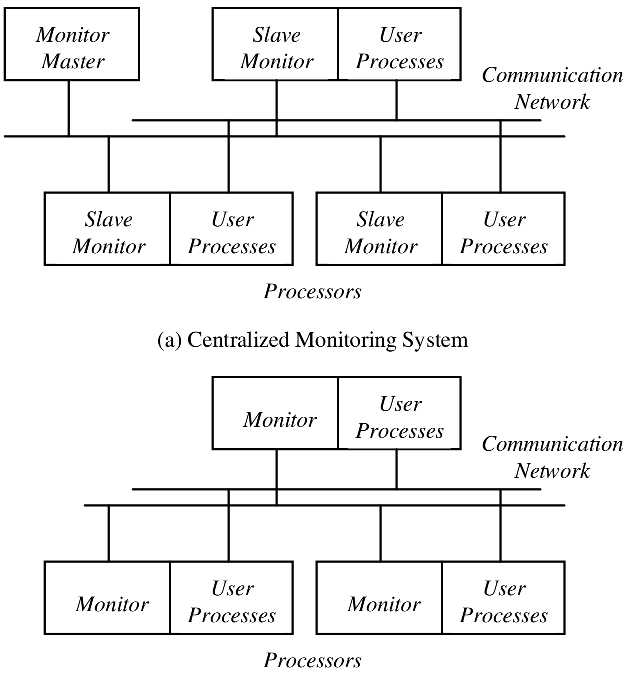
The basic mechanisms in the logical clock approach are the logical clock updating and the communication control. To implement the logical clock approach, we need to have ways of:

- instrumenting the programs;
- capturing the sending (i.e., $S(e_i)$) and receiving cost (i.e., $R(e_j)$) of a communication (see Section 2);
- keeping track of the logical time of each process; and
- keeping track of the state of each process (e.g., the current instrumented event of the process).

In this section, our discussion will focus on the infrastructure of the monitoring system that provides the above mechanisms rather than the detailed implementation.

In our modified logical clock approach, the control of non-deterministic communications is based on the detection of the ready condition. Since the detection of the ready condition needs the state and logical clock information of all processes under monitoring, the basic problem in the implementation of our modified logical clock approach is how the information can be made available for a monitor to make decisions.

The monitoring system can be implemented using either a centralized or a distributed scheme (see Figure 6). In a centralized scheme, there is a *slave monitor* at each processor, which collects the real-time information about the instrumented event and passes the information to the *monitor master*. The slave monitor can make the decision to commit a deterministic communication, since it does not require information about other processes. However, for a non-deterministic communication, the slave monitor must wait for the monitor master to make the decision. The monitor master, which may run on a dedicated processor, keeps global information about all processes under monitoring and makes the decision to commit the non-deterministic commu-



(a) Centralized Monitoring System

(b) Decentralized Monitoring System

Fig. 6. Centralized vs. Decentralized Monitoring System

nications.

The centralized scheme can be implemented straightforwardly, but may introduce bottle-necks in the communication network due to the large volume of information sent to the monitor master. However, the advantage of the centralized scheme is that it reduces monitoring overhead at each processor by dividing the communication control into two parts: the permission of deterministic communication which is less time-consuming and done locally by each slave monitor; and the permission of non-deterministic communication which is more time-consuming and done by the monitor master.

In a distributed scheme (see Figure 6(b)), there will be no monitor master in the system. The monitors at each processor are symmetric, and the decision to commit a non-deterministic communication will also be made locally by the monitor. However, the global information about the status and logical time of all processes needs to be stored locally, and the problem is to update the local copy of the global information at each monitor.

A distributed monitoring system, which keeps a copy of global information locally at each monitor, has been implemented for an nCube 2 multiprocessor machine [21]. Broadcasting is used to update the information in the implementation. The experiments showed that the monitor introduces very minimum intrusion in terms of program performance. There is a minimal difference between the performance data collected with monitors using logical time information and that collected without monitors using the system's real-time clocks.

6 Conclusions And Future Work

Debugging and performance tuning are important activities in parallel program development, both of which require a monitoring facility to collect program execution information. However, a fundamental problem with runtime monitoring of parallel programs is the intrusion introduced by instrumenting the original program. An intrusive monitor may alter the order of program events and distort the timing information. This may result in a change in the probability of making particular non-deterministic choices, and thus affecting the program behavior. To minimize the amount of intrusion in monitoring parallel programs, the logical clock approach (LCA) was proposed.

This paper presents a modified logical clock approach and an integrated strategy for its interlock avoidance. In the communication model adopted in this paper, the requirement that synchronization must happen before message-passing is removed. Message-passing now can happen before or during synchronization. In addition, in our model, we assume that a non-deterministic receive will communicate with the process whose message arrives first rather than the process that sends first. Hence, the communication model adopted in this paper reflects common characteristics found in most of message-passing based parallel computing platforms. To support our new communication model, the logical clock updating and control mechanisms in LCA are modified accordingly.

One major problem of LCA is that in the case of non-deterministic communication and when several processes wait on each other's logical clock to advance, an interlock situation in which none of the processes is able to continue its execution may occur. So, the other main contribution of this paper is an integrated interlock avoidance strategy based on the technique that commits non-deterministic communication once the ready condition is satisfied. The interlock problem is described. The ready condition is also clearly defined. Furthermore, we also prove that a non-deterministic communication can be safely committed once a ready condition is reached and that the ready condition can always be reached.

The ready condition concept proposed in this paper and the interlock avoidance strategy based on this concept are novel. In addition, the interlock avoidance strategy can be easily integrated with the logical clock updating and control mechanisms in LCA. The initial investigation and performance study of the strategy was carried out on an nCube 2 multiprocessor machine [21], and a variation of the strategy was also proposed and implemented on a CM5 [14]. This paper gives a thorough systematic treatment of the ready condition concept and the interlock avoidance strategy based on this concept, and focuses more on the correctness of the strategy.

The existing implementation of LCA can be optimized in many ways. As has been mentioned in the last section, the major issue is to reduce the total communication overhead introduced by the monitor. With a logical clock implemented for each process, the monitor can perform many different types of probing activities during run-time without affecting the program behavior. One of the useful activities is the dynamic flushing and garbage-collection of unnecessary trace data. The trace file can, therefore, be controlled under a manageable size. Another useful application of the scheme is to embed an intelligent decision-support system in the monitor for making run-time decisions on performance tuning and debugging. Thus, our future work includes the optimization of the existing implementation and the enhancement of the monitoring system with the aforementioned capabilities.

References

[1] Wentong Cai and S. J. Turner, An algorithm for distributed discrete-event simulation - the "carrier-null message" approach, in Proc. of *SCS Multiconference on Distributed Simulation*, pp. 3-8, San Diego, USA, Jan. 1990.

[2] Wentong Cai and S.J. Turner, Experimental studies of conservative distributed discrete-event simulation on transputer networks, In S. J. Turner edited *Tools and Techniques for Transputer Applications*, pp. 138-147, IOS Press, 1990.

[3] Wentong Cai, Parallel program monitoring - the logical clock approach and its deadlock avoidance, PhD Thesis, Dept. of Computer Science, University of Exeter, UK, 1991.

[4] Wentong Cai and Stephen J. Turner, Approach to the run-time monitoring of parallel programs, *The Computer Journal*, Vol.37, No.4, pp. 333-345, 1994.

[5] Paul S. Dodd and Chinya V. Ravishankar, Monitoring and debugging distributed real-time programs, *Software – Practice and Experience*, Vol.22, No.10, pp. 863-877, Oct. 1992.

[6] Colin Fidge, Logical time in distributed computing Systems, *IEEE Computer*, pp. 28-33, August 1991.

[7] Colin Fidge, Fundamentals of distributed system observation, *IEEE Software*, pp. 77-83, Nov. 1996.

[8] E. Fromentin, N. Plouzeau and Michel Raynal, An introduction to the analysis and debug of distributed computations, in Proc. of *IEEE First Int. Conf. on Algorithms and Architectures for Parallel Processing* (ICA3PP), pp. 545-553, April 1995.

[9] Richard Fujimoto, Parallel discrete-event simulation, *Communications of the ACM*, Vol.33, No.10, pp. 31-53, Oct. 1990.

[10] R. Gupta and M. Spezialetti, Dynamic techniques for minimizing the intrusive effect of monitoring actions, in Proc. of *15th IEEE Int. Conf. on Distributed Computing Systems*, pp. 368-376, Vancouver, Canada, June 1995.

[11] David Helmbold, Charles E. McDowell and Jingzhong Wang, Analyzing traces with anonymous synchronization, In Proc. of *1990 Int. Conf. on Parallel Processing*, pp. II.70-II.77, August 1990.

[12] J.K. Hollingsworth, B.P. Miller and J. Cargille, Dynamic program instrumentation for scalable performance tools, in Proc. of *1994 Scalable High Performance Computer Conf.*, pp. 841-850, Knoxville, USA, May 1994.

[13] Leslie Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, Vol.21, No.7, pp. 558-565, July 1978.

[14] K.C. Li and K. Zhang, Instrumenting parallel programs based on a virtual clock approach, in Proc. of *Int. Conf. on Parallel and Distributed Processing Techniques and Applications* (PDPTA'97), pp. 31-40, 1997.

[15] A. Malony, A. Hammerslag and D. Jablonowski, Traceview: A trace visualization tool, *IEEE Software*, 8(5), pp. 19-28, Sept. 1991.

[16] A. Malony, D.A. Reed and H. Wijshoff, Performance measurement intrusion and perturbation analysis, *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.4, pp. 433-450, July 1992.

[17] D.A.P. Michell, J.A. Thompson, G.A. Manson and G.R. Brookes, *Inside the Transputer*, Blackwell Scientific Publications, 1990.

[18] A. Olson and K. G. Shin, Fault-tolerant clock synchronisation in large multicomputer systems, *IEEE Trans. On Parallel and Distribributed Systems*, Vol.5, pp. 912-923, 1994.

[19] D.A. Reed and A. Malony, Parallel discrete event simulation: the Chandy-Misra approach, In Proc. of *SCS Multiconference on Distributed Simulation 1988*, pp. 8-13, San Diego, USA, 1988.

[20] W. Wu, M. Spezialetti and R. Gupta, On-line avoidance of the intrusive effects of monitoring on runtime scheduling decisions, in Proc. of *16th IEEE Int. Conf. on Distributed Computing Systems*, pp. 216-223, Hong Kong, May 1996.

[21] K. Zhang, C. Sun, and K.C. Li, Collecting timing information while preserving events ordering in parallel program instrumentation, Technical Report, Report No. C/TR95-07, Department of Computing, Macquarie University, December, 1995.