Proceedings of International Conference
on Computer Architecture

The Analysis and Detection Method for Nondeterminacy in Parallel/Distributed Program

Li-Wei Chu and Tsung-Chuan Huang
Department of Electrical Engineering
National Sun Yat-Sen University
Kaohsian 80424, Taiwan, R.O.C.
tch@mail.nsysu.edu.tw

Abstract

Debugging parallel programs is more difficult than debugging traditional sequential programs. The difficulty is mostly due to the synchronization and communication between components of a parallel program. The concurrent components of a parallel program communicate with each other via either shared variable or message passing. Without proper synchronization between these communications, nondeterminacy may arise. In shared variable model, nondeterminacy is caused by data race, and in message passing model it is caused by message race.

In this paper, we presented a graph based analysis technique suitable for message passing system to detect message race. Since this technique can be applied on trace analysis(post-mortem analysis), it can help the programmer find all potential races and the errors caused more efficiently. The new detection method will enhance parallel debuggers the capability of race detection and nondeterminacy elimination.

1. Introduction

Parallel processing, the method of having many small tasks solve one large problem, has emerged as a key enabling technology in mordern computing. As more and more organizations have high-speed local area networks interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. We will call the parallel programs that execute in such environment and communicate via message passing as *parallel/distributed programs* [8].

A parallel program exhibits nondeterminacy when it gives different results on different runs, given the same input [4]. When processes communicate via messages, variations in process scheduling and message latencies can cause race condition. The race condition is a source of nondeterminacy.

The classic approach to debugging sequential programs involves repeatedly stopping the program during execution, examining the state, and then either continuing or reexecuting in order to stop at an earlier point in the execution. This style of debugging is called *cyclic debugging*. Unfortunately, because of nondeterminacy, parallel programs do not always have reproducible behavior; the undesirable behavior may not appear when programs are reexecuted.

The current methods for determining potential races in parallel programs can be roughly divided into three groups:compiling time analysis [9,1], run time analysis [3,10], and trace based analysis [4,5].

This paper describes a mechanism that can find errors caused by message race and allow cyclic debugging to be applied during reexecution. In section 2, we will give message race a formal definition.

Our approach to the detection of message races is based on trace analysis. The trace analysis works by firstly obtaining a process communication trace of the program for a set of input values, then analyzing this trace output for possible races using the algorithms presented in this paper. For the sake of simplicity, all programs in this paper are considered to have a fixed number of processes and all receive events in the programs are to be blocking, i.e, the receive returns only when the data is in the receive buffer.

The trace file records the information of communication events(send events and receive events) among the concurrent processes during the execution of parallel/distributed program. For convenience of explanation, we will use process-time diagram [8] to describe these information.

2. Race Condition

It is frequently possible that more than one message is available for one receive event in a process of parallel/distributed program if no proper synchronization. A process-time diagram is a convenient way to describe the communication of a message passing parallel program. In a process-time diagram, the process's execution is represented as vertical lines; time flows down from the top of each line. When there is a message from process P_i to process P_j, we draw an arrow from P_i to P_j at the corresponding send event and receive

event. For example, in Figure 1, there are four processes:P_1 to P_4, a is a message sent from the send event s_1 in P_1 to the receive event r_2 in P_2, and so on.

Definition 2.1 *Reachable Set:* In a process, *reach(r)* is the set of messages which their corresponding send events can be concurrent with receive event r ■

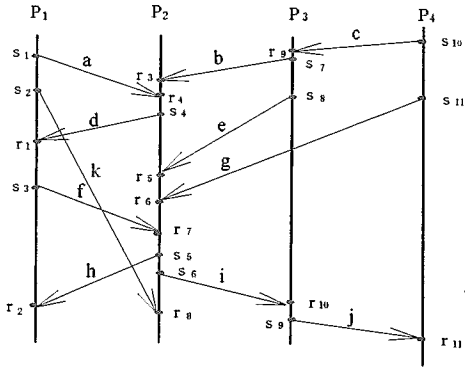


Figure 1. Process-time diagram.

When the computation reaches a receive event, it is not possible for messages not in the reachable set to be received at that receive event. For the example of Figure 1, *reach(r_1)* is {a, b, e, g, k}(their corresponding send events are s_1, s_2, s_7, s_8, s_{11}). Note that, f is not in *reach(r_3)* because before r_3 is done message d can not be sent to r_1; Before r_1 is done message f can not be sent to r_7. That is, s_3 can not be concurrent with r_3.

In PVM(Parallel Virtual Machine), the send event of a message can specify which process to send and label the message with an integer identifier *msgtag*. For a receive r in process P, if a message is sent to P and its *msgtag* is the same with the label that r specifies, then we say that this message is acceptable at r.

Definition 2.2 *Message Race:* A *message race* occurs at a receive event r, *iff reach(r)* consists of at least two messages p and q, such that p and q are acceptable at r ■

If we can create the reachable set for each receive event, we are able to test whether there exists a message race at that receive.

During the execution of a program, it is possible that, for a particular process, before a send event is done all the messages appearing in the reachable set of the receives following this send event can not be sent. In other words, under this circumstances, the send events sending the messages in the reachable set of the receives following this send event are controlled by this send, and the number of the messages controlled by this send event is equal to the number of receives following this send.

But how can the send event in process P control the messages that are sent to P? We know that, in parallel programs, the synchronization between concurrent processes is through communication. Thus, by sending messages to other processes, the send events in P can control the execution of those send events in other processes which will send messages to P.

Definition 2.3 *Control Point:* A control point in a process P is a point locating at the send event *s* that occurs immediately after a receive *r* in the process-time diagram, such that the number of the messages controlled by *s* is equal to the number of the receives following *s*. ■

For example, in Figure 1, process P_1 contains a control point at s_3, and process P_2 contains another control point at s_7. Definition 2.3 implies that the number of messages received at the receives above a control point is equal to the number of these receives. In addition, only messages that are received between two consecutive control points of a process may cause race.

3. Task Graph

Since we concentrate on trace analysis, it is much easier to prove certain properties about the execution of message passing programs when the execution is represented as a graph. For this purpose, a graph is constructed from the significant events that are recorded in a log file by executing an instrumented version of the program being debugged.

Theorem 3.1 In a process, for any two receives, r_1 and r_2, if there is no any send existing between them, then *reach(r_1) = reach(r_2)*.

proof: Let r_2 locate at the position behind r_1 in process-time diagram. If there exists a message m such that $m \in reach(r_2)$ but $m \notin reach(r_1)$, then there must exist a send between r_1 and r_2 such that m is controlled by this send. This contradicts with the condition given above.

But, is it possible to exist a message n such that $n \in reach(r_1)$ but $n \notin reach(r_2)$? This condition also can not be true because if $n \notin reach(r_2)$ then before r_2 is done n can not be sent. This means that when the execution reaches r_1, n must have not been sent. But r_1 is before r_2, $n \notin reach(r_2)$ implies $n \notin reach(r_1)$. This contradicts with $n \in reach(r_1)$.

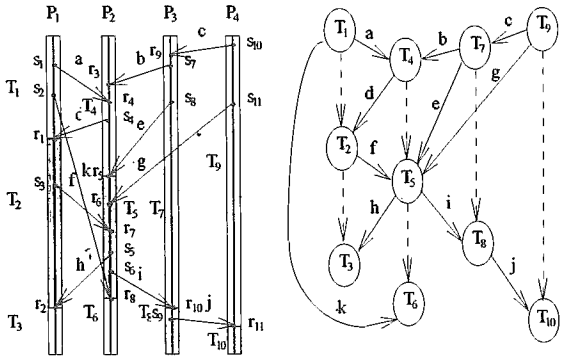
From the above discussion we can conclude that $m \in reach(r_2)$ *iff* $m \in reach(r_1)$. ■

In a task graph, the vertices set is formed by the collection of tasks, and the transformation from a process-time diagram to its corresponding task graph is defined as follows:

Definition 3.1 *Process-Time Diagram to Task Graph Transformation* [2]:

1. First Task: For each process represented in the process-time diagram, the process segment in the diagram starting with the beginning of the process and terminating with either the instruction immediately followed by a receive, or termination of the process, forms a task.
2. Other tasks: A process segment in the process-time diagram which is not included in any task, starting with the receive following an instruction that terminates an existing task and ending with either the instruction immediately followed by the first receive after a send or the termination of process, forms a task.
3. Edges: All messages in the process-time diagram are preserved in the task graph, and we call such edges the message edges. In addition, the edges from a task to the immediately following task are called the task edges. ■

For example, the process-time diagram in Figure 1 can be transformed into its corresponding task graph shown in Figure 2. In the process-time diagram, tasks are represented by shadows labeled with T_i, which form the vertices of task graph. We can see that T_1 starts with the beginning of process P_1 and terminates at the instruction before r_1. T_2 starts with r_1 and terminates at the instruction before r_2, and so on. Note that, we represent each task edge with a broken line.



(a) Process-time diagra (b) Task graph

Figure 2. Process-time diagram and its corresponding task graph.

From Theorem 3.1 and the definition of transformation from process-time diagram to task graph, we can see it is true that all the receives in a task have the same reachable set. This implies that when we are detecting message races, we do not have to analyze each receive's reachable set one by one, but can just do it task by task. Therefore, for simplicity, we will use *reach(t)* to represent the reachable set of every receive in task t, because the reachable sets in task t are all the same.

4. Message Race Detection

Definition 4.1 In a task graph, all the tasks, in a process, locating between two consecutive control points are said to be at the same control point level. If a process contains no control point, then all the tasks are at control point level 1. The *control point level receive* of a control point level i is defined as the set of messages appearing in the reachable sets of receives at control point level i, denoted by *CPLR(i)*. ■

Thus, all of the tasks from the first task in a process to the task containing the first control point are at control point level 1. For example, in process P_1 of Figure 3, because T_2 contains a control point, tasks T_1 and T_2 are at control point level 1 and T_3 is at control point level 2. Because process P_2 contains no control point, all the tasks in P_2 are at control point level 1. In addition, the contents of *CPLR(1)* in P_1 is {d}.

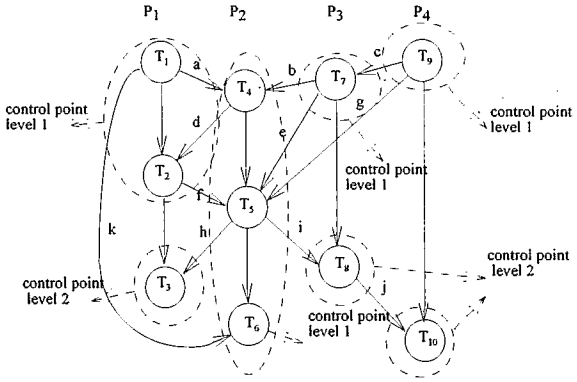


Figure 3. Control point level in task graph.

Theorem 4.1 For a particular process, let i be a control point level. If a message is in *CPLR(i)*, then it will not be in the *reachable set* of receives at other control point levels.

proof: For two control point levels i and j in the same process P, let i < j. If a message m ∈ *CPLR(i)* then, from the definition of control point, we can get that m can not be in the reachable sets of receives in control point level j. Because there is a control point existing at control point level i, m ∈ *CPLR(j)* means that m is controlled by that control point. This contradicts with m ∈ *CPLR(i)*, therefore m ∉ *CPLR(j)*.

In another aspect, definition 2.3 implies that the number of messages received at the receive events above a control point is equal to the number of these receive events. Thus, for any message in *CPLR(j)*, it must have been consumed when the execution reaches the receives at control point level j. So, if message n ∈ *CPLR(j)*, it follows that n ∉ *CPLR(i)*. From the above discussion, the theorem follows. ■

Theorem 4.1 indicates that, in a process, only messages received at the same control point level can race with each other. Since the messges received at control point level i can never be received at another control point level j, they will not be in the reachable set of the receives at control point level j. Thus, when building reachable set, we can just analyze those messages at the same control point level to find potential races rather than analyzing the messages received at all tasks in that process.

Definition 4.2 *Control Set:* The control set of a task *t* contains all the messages controlled by task *t* and we denote it as *cs(t)*. ■

Let task *t* be at control point level j. Definition 4.2 implies that *cs(t)* is equal to $\bigcup_{i<j} CPLR(i)$. Since the messages received at the receives above task *t* can not be controlled by *t*, the content of *cs(t)* must be the messages received by all the receives below *t*. For example, in Figure 3, the contents of $cs(T_1)$ is {d, h}.

The following algorithm can build the control set for every task by tracing a matrix, sm, which keeps a record of the communication between tasks. If there exists a message sent from task i to task j, then sm[i][j] will increase by one.

〖 Algorithm 4-1 〗 Building control set

Input:

Sm: Matrix that keep a record of the communication between tasks.

Tskcnt: The chararcteristic table for every task.

Process_char: The characteristic table for every process.

All_tasks: The total number of tasks in a task graph.

Output:

CS: the control set for every task.

Method:

```
/* This procedure is used to trace the sm, if sm[i][j] = 1,
   then it denotes that task i send a message to task j */
procedure find_send(ct, target);
 {
  /* check the current task ct and others tasks that it
     sends messages to them*/
  i = 1;
  while (i != (all_tasks+1))
  /* check every element in sm */
    {
     if (sm[target][i] != 0) /* if task target sends a
                                message to task i */
     /* if task i and task ct is in the same process */
     if (tskcnt[ct].process == tskcnt[i].process)
       put_message(ct,target,i);
     else
       find_send(ct,i);
     i++;
    }
  /* check the tasks that follows the current task in the same
     process */
  x = tskcnt[target].process_index+1;
  y = tskcnt[target].process;
  if (x <= process_char[y].total_task)
    find_send(ct,process_char[y].tasks[x]);
 }
/* MAIN PROGRAM */
 {
 /* begin building the control set with the last task*/
  for (every process)
    {
     current_task = process_char[j].last_task;
     while(current_task>=process_char[j].first_task)
      {
       i = 1;
       while (i != (all_tasks+1))
        {
         if (sm[current_task][i] != 0)
           find_send(current_task,i);
           i++;
        }
       current_task = current_task - 1;
      }
    }
 }
```

For a task t, messages received at upper control point level and in cs*(t)* will not race with the messages in *reach(t)*. Theorem 4.2 will show this.

Theorem 4.2 In a process, if task t is at control point level i then reach(t) = {all messages in this process} - (CPLR(1) ∪ CPLR(2) ∪ ⋯ ∪ CPLR(i-1)) - CS(T).

proof: From Theorem 4.1 and the definition of control set, the theorem follows. ■

Applying this theorem we can find the reachable set of every task as follows:

〖 Algorithm 4-2 〗 Building reachable set

Input:

Process_char: The characteristic table for every process.

CS: The control set of each task.

Tskcnt: The chararcteristic table for every task.

Output:

Reach: The reachable set of each task.

Method:

```
/* MAIN PROGRAM */
{
 for every process
  {
```

```
current_task = process_char[j].first_task; /* from
                    the first task in a process*/
below_cp = process_char[j].first_task;
cp_level = 1;
while (current_task<=the last_task in process)
 {
  index = 1;
  i = current_task + 1;
  tskcnt[current_task].cp_level = cp_level;
  /* set the total receive number below
    current_task */
  for every task below the current_task
    for ( x=1;x<=tskcnt[i].total_recv;x++)
      index++;
  tskcnt[current_task].total_under=index-1;
  /* if the number of cs is the same with the
    number of receives below current_task, then it
    is a control point*/
  if(cs[current_task].total_message==
    tskcnt[current_task].total_under)
    {/* for every control point, build the control
      point table*/
      index = 1;
      for all messages in the control point level
        {/* put all the messages in the control point
          level into cpl to get CPLR(cp_level)*/
          cpl[j][cp_level].message[index]=
            rsm[tskcnt[x].recv[y]];
          index++;
        }
    }
  current_task++;
 }
for every task in this process
 {
  index = 1;
  for every message received in this process
    {/* if this message is not in the cs( ) of the
      task*/
      if (in_cs == 0)
        { /* if the task is in control point level 1*/
          if (tskcnt[this task].cp_level == 1)
            /* reach(task) = total_message of a
              process - cs(task)*/
            reach[this task].message[index] = this
              message;
          else /* if the task is not in control point
            level 1 */
            if (in_cplr == 0)/* if this message is not
              in upper control point level*/
              reach[this task].message[index] = this
                message;
        }
    reach[this task].total_message = index - 1;
  }/*end ot this task*/
}
```

According to the definition of message race, we can see that only when the number of messages accepted at the currently analyzing receive are larger than or equal to 2, message race may happen at that receive.

【 Algorithm 4-3 】 Detecting message race

Input:

Reach: The reachable sets of receives for every task.

Output:

Information about messages possible to arise racing.

Method:

```
/* MAIN PROGRAM */
for every process
 if(reach[current task].total_message >= 2 )
  for every receive in current task
   {
    index = 1;
    for every message in reach(current task)
     if ((the tag of the send is the same with the receive) ||
       (the tag of the receive is any))
       {
         accept[the current
           receive].message[index]=reach[the current
           task].message[z];
         index++;
       }
    accept[the current receive].total_message = index-1;
    if (accept[the current receive].total_message >= 2)
      {
        report the identifier of receive to user;
      }
   }
  }     /* for every task */
 }      /* for every process */
```

5. Simulation

The simulation of the message race detection is done by first getting a PVM trace file as the input to build the process-time diagram and the tag-table, and then uses these tables to construct the task graph. After constructing the task graph, we can then build control set for every task and by theorem 4.2 build the reachable set.

The process-time diagram obtained from the trace file in our simulation is shown in figure 4.

For each send event, the first argument specifies which process to send and the second argument specifies the message tag. Similarly, the first argument of the receive event specifies from which process the receive event can receive and the second argument specifies the message tag. For example, $send_1(2,3)$ specifies the message with tag=3 is sent to process 2; $recv_5(1,-1)$ specifies this receive event is waiting for a message with any tag number to be sent from process 1.

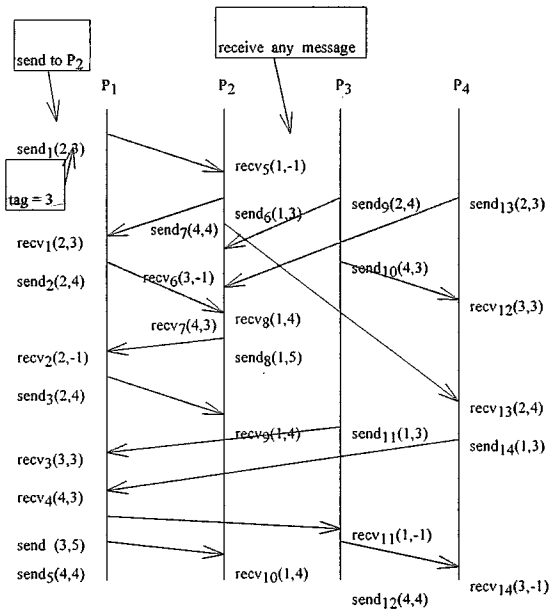


Figure 4. The process-time diagram for our simulation.

The information obtained from process-time diagram is as follows :

1. The receive and send events in a process :

process 1 P_1 : $send_1$ $recv_1$ $send_2$ $recv_2$ $send_3$ $recv_3$ $recv_4$ $send_4$ $send_5$

process 2 P_2 : $recv_5$ $send_6$ $send_7$ $recv_6$ $recv_7$ $recv_8$ $send_8$ $recv_9$ $recv_{10}$

process 3 P_3 : $send_9$ $send_{10}$ $send_{11}$ $recv_{11}$ $send_{12}$

process 4 P_4 : $send_{13}$ $recv_{12}$ $recv_{13}$ $send_{14}$ $recv_{14}$

2. The send events and their corresponding receives :

In process 1, the send events are :

<send 1> -- corresponding to --> [recv 5] {in process 2}
<send 2> -- corresponding to --> [recv 8] {in process 2}
<send 3> -- corresponding to --> [recv 9] {in process 2}
<send 4> -- corresponding to --> [recv11] {in process 3}
<send 5> -- corresponding to --> [recv10] {in process 2}

In process 2, the send events are :

<send 6> -- corresponding to --> [recv 1] {in process 1}
<send 7> -- corresponding to --> [recv13] {in process 4}
<send 8> -- corresponding to --> [recv 2] {in process 1}

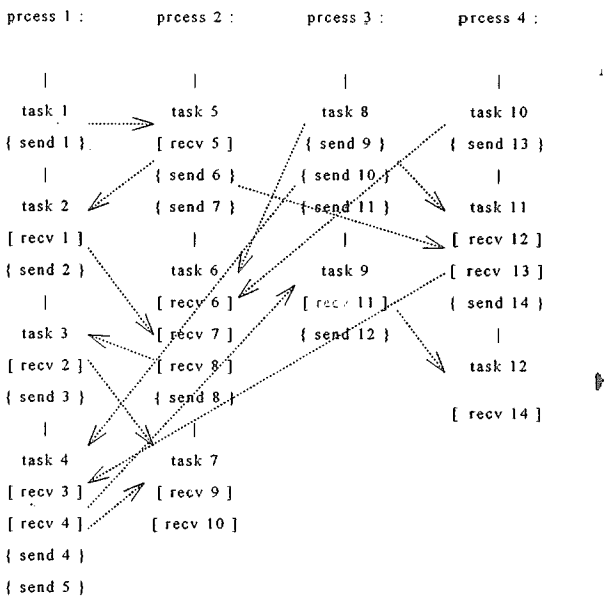
In process 3, the send events are :

<send 9> -- corresponding to --> [recv 6] {in process 2}
<send10> -- corresponding to --> [recv12] {in process 4}
<send11> -- corresponding to --> [recv 3] {in process 1}
<send12> -- corresponding to --> [recv14] {in process 4}

In process 4, the send events are :

<send13> -- corresponding to --> [recv 7] {in process 2}
<send14> -- corresponding to --> [recv 4] {in process 1}

The task graph obtained from the process-time diagram is :



There are 12 tasks in 4 processes

After the analysis, the send-metrix is :

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	1	0	0	0	0	0	0	0
2	0	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	1	0	1	0	0	0
5	0	1	0	0	0	0	0	0	0	0	1	0
6	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	1	0	1	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	1	0	0	0	0	0	0
11	0	0	0	1	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0

For each task, we build its reachable set as follows where reach(i) represents the reachable set of task i :

----------task 1----------
Task 1 has no receive instruction.
----------task 2----------
message:send 6----------> recv 1
message:send 11----------> recv 3
message:send 14----------> recv 4

There are total 3 messages in reach(2).

For each receive event in a task, we will check if every message in its reachable set can be received by it.

*****Analyzing recv1 (tag = 3):*****

message : send 6-----> recv 1 is accepted at recv 1

No message race here.

----------task 3----------
message:send 6----------> recv 1
message:send 8----------> recv 2
message:send 11----------> recv 3
message:send 14----------> recv 4

There are total 4 messages in reach(3).

*****Analyzing recv2 (tag = -1):*****

message : send 6------> recv 1 is accepted at recv 2
message : send 8------> recv 2 is accepted at recv 2
message : send11-----> recv 3 is accepted at recv 2
message : send14-----> recv 4 is accepted at recv 2

There are 4 messages racing with each other (in recv 2) : 6, 8, 11, 14

----------task 4----------
message:send 6----------> recv 1
message:send 8----------> recv 2
message:send 11----------> recv 3
message:send 14----------> recv 4

There are total 4 messages in reach(4).

*****Analyzing recv3 (tag = 3):*****

message : send11-----> recv 3 is accepted at recv 3

No message race here.

*****Analyzing recv4 (tag = 3):*****

message : send14-----> recv 4 is accepted at recv 4

No message race here.

6. Conclusion

In this paper we describe a graph based technique which can detect message races in a parallel program. In PVM, we can use the libpvm trace facility to trace the events we are interested in. By editing the trace mask, we can select the events to be collected into the trace file. The trace output contains all the information needed to build a task graph.

We have demonstrated algorithms that can identify which messages at a receive will race with each other. An advantage of our approach is that it does not place any restriction on the control structure of the programs. Iterative control structures such as loops, which are difficult to analyze statically, can easily be traced by our approach. In addition, our approach have a better performance than run time analysis. Because the time delay of run time analysis may be caused by CPU scheduling or network delay, it is even possible for a process to take several hours to wait for the arriving of a message. Thus, our approach have a averagely better performance than run time analysis.

Since our technique can be applied to parallel debugger, we can help users who are troubled by nondeterminacy to isolate errors caused by message race when writing and debugging parallel programs.

References

[1] D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," in Proc. Second ACM SIGPLAN Symp. Principles and Practice of Paralleled Programming(PPOPP).

[2] Damodaran-kamal, S. K., and Francioni, J. M. "Nondeterminacy: testing and debugging in message passing parallel program," in Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, May 1993.

[3] A. Dinning and E. Schonberg, "An expirical comparison of monitoring algorithm for access anomaly detection," in Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming(PPOPP), 1990.

[4] P. A. Emrath and D. A. Padua, "Automatic detection of nondeterminacy in parallel programs," in Proc. Workshop Parallel and Distributed Debugging, May 1988, pp. 89-99.

[5] P. A. Emrath, S. Ghosh, and D. A. Padua, "Event synchronization analysis for debugging parallel programs," in Proc. Supercomputing'89, Reno, NV, Nov. 1989.

[6] Gait, J. "A debugger for concurrent programs," Softw. Pract. Exper., Vol 15, No. 6, pp. 539-554.

[7] German S. Goldszmidt, and Shaula Yemini. "High-level lanaguage debugging for concurrent programs," ACM Transactions on Computer Systems, Vol. 8, No. 4, November 1990.

[8] Ming-Jer Lee, "The design and implementation of a parallel debugger," Master Thesis EE. NSYSU. June 21, 1995.

[9] C. E. McDowell, "A practical algorithm for static analysis of parallel programs," J. Parallel and Distributed Comput., June 1989.

[10] Netzer, R.H.B., and Miller, B.P., "Optimal tracing and replay for debugging message-passing parallel programs," in Proc. Supercomputing, November 1992, pp. 502-510.

[11] P. Bates and J. Wileden, "High-level debugging of distributed systems: The behavioral abstraction approach," The Journal of Systems and Software

1983, Vol. 3, No.4, pp. 255-264.

[12] R. S. Side and G. C. Shoja, "A debugger for distributed programs," Software-Practice and Experience, Vol. 24, No. 5, pp. 507-525, May 1994.

[13] T. J. Leblance and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," IEEE Trans. Comput., Vol. C-36, No. 4 pp. 471-482, Apr. 1987.