MPIRace-Check: Detection of Message Races in MPI Programs*

Mi-Young Park[1], Su Jeong Shim[1], Yong-Kee Jun[2,**], and Hyuk-Ro Park[3]

[1] Chonnam National University, Gwanju
openmp@korea.com, sjsim@chonnam.ac.kr
[2] Gyeongsang National University, Jinju
jun@gsnu.ac.kr
[3] Chonnam National University, Gwanju
South Korea
hyukro@chonnam.ac.kr

Abstract. Message races, which can cause nondeterministic executions of a parallel program, should be detected for debugging because nondeterminism makes debugging parallel programs a difficult task. Even though there are some tools to detect message races in MPI programs, they do not provide practical information to locate and debug message races in MPI programs. In this paper, we present an on-the-fly detection tool, which is MPIRace-Check, for debugging MPI programs written in C language. MPIRace-Check detects and reports all race conditions in all processes by checking the concurrency of the communication between processes. Also it reports the message races with some practical information such as the line number of a source code, the processes number, and the channel information which are involved in the races. By providing those information, it lets programmers distinguish of unintended races among the reported races, and lets the programmers know directly where the races occur in a huge source code. In the experiment we will show that MPIRace-Check detects the races using some testing programs as well as the tool is efficient.

Keywords: message-passing programs, debugging, message races, MPIRace-Check.

1 Introduction

In a distributed parallel program [1,4,9,14], processes communicate with each other through message-passing and those messages may arrive at a process in a nondeterministic order by variations in process scheduling and network latencies.

* This work was supported in part by Research Intern Program of the Korea Science and Engineering Foundation and in part by the 2th BK21.

** Corresponding author. Also involved in Research Institute of Computer and Information Communication (RICIC) as a research professor in Gyeongsang National University.

C. Cérin and K.-C. Li (Eds.): GPC 2007, LNCS 4459, pp. 322–333, 2007.

Nondeterministic arrival of messages causes nondeterministic executions of a parallel program [7,10,11]. If two or more messages are sent over communication channels on which a receive listens, and they are simultaneously in transit without guaranteeing the order of their arrivals, a message race [2,3,5,6,8,12,13] occurs in the receive event and causes nondeterministic executions of the program.

Message races, which can cause nondeterministic executions of a parallel program, should be detected for debugging because nondeterminism, intended or otherwise, makes debugging message-passing parallel programs a difficult task [7,10,11]. Even though some parallel programs are designed to have message races in order to improve their performance, detecting message races is critical in debugging parallel programs for two reasons. First, message races complicate debugging because their nondeterministic nature can prohibit equivalent re-execution of a program from being repeated [7]. Second, message races can prevent a program from being tested in all the possible executions of a program [7]. Therefore message races should be detected for debugging message-passing programs.

There are several tools for detecting message races such as MAD [8], MARMOT [5,6], and MPVisualizer [2,3]. However those tools are not practical for debugging message-passing programs because they do not provide practical information to locate and debug message races. Also some of them can not exactly detect race conditions because they detect message races just by identifying the use of wild card receives as sources of race conditions. Therefore, due to lack of information and wrong detection, programmers can be easily overwhelmed by the incorrect information or be incapable of finding where the races occurred in a huge source code.

In this paper, we present an on-the-fly detection tool, which is MPIRace-Check, for debugging MPI [14,15] programs written in C language. MPIRace-Check detects and reports all race conditions in all processes during an execution by checking the concurrency of the communications between processes. Also it reports message races with some practical information such as the line number of a source code, the processes number, and the channel information which are involved in the races. By providing those information, it lets programmers distinguish of unintended races among the reported races, and lets the programmers know directly where the races occur in a huge source code. In the experiment we will show that MPIRace-Check detects and reports the races using MPI_RTED [15] testing programs as well as this tool is efficient using a kernel benchmark program.

In the following section 2, we describe the notion of message races and explain the problem of the previous tools. In section 3 we explain the methods used in developing MPIRace-Check and then we show that the accuracy and the efficiency of MPIRace-Check using MPI_RTED testing programs and a kernel benchmark program in the experiment of section 4. In the last section we conclude this paper and discuss future work.

2 Background

In this section, we describe our model of parallel programs, and the notion of message races. Also we introduce the previous tools to detect the races and explain the problem of the previous tools.

2.1 Message Races

An execution of a message-passing program [1,10,11,13] can be represented as a finite set of events and the *happened-before* relations [4,9] defined over those events. If an event a always occurs before another event b in all executions of the program, it satisfies that a happens before b, denoted $a \rightarrow b$. For example, if there exist two events $\{a, b\}$ executed in the same process, $a \rightarrow b \vee b \rightarrow a$ is satisfied. If there exist a send event s and the corresponding receive event r between a pair of processes, then $s \rightarrow r$ is satisfied. We denote a message, sent by a send event s, as $msg(s)$. The binary relation $\rightarrow$ is defined over its irreflexive transitive closure; if there are three events $\{a, b, c\}$ that satisfy $a \rightarrow b \wedge b \rightarrow c$, it also satisfies $a \rightarrow c$. When an event a does not happen before an event b, we denote the relation between them as $a \nrightarrow b$.

A *message race* [2,3,5,6,8,13] occurs in a receive event, if two or more messages are sent over communication channels on which the receive listens and they are simultaneously in transit without guaranteeing the order of their arrivals. A message race is represented as $\langle r, M\rangle$: r is the first receive event and M is a set of racing messages toward r. Any send event s included in M, but not the one received by r, satisfies $s \nrightarrow r$ or $r \nrightarrow s$.

Even though some parallel programs are designed to have message races in order to improve their performance, detecting message races is critical in debugging parallel programs for two reasons. First, message races complicate debugging because their nondeterministic nature can prohibit equivalent re-execution of a program from being repeated [7]. Second, message races can prevent a program from being tested in all the possible executions of a program [7]. Therefore message races should be detected for debugging message-passing programs.

Figure 1 shows a partial order of events that occurred during an execution of a message-passing program. In the figure two processes P_3 and P_4 send two messages $msg(i)$ and $msg(k)$ to P_2. At this time two messages $msg(i)$ and $msg(k)$ are racing toward the receive event j of P_2 because the send event k satisfies $k \nrightarrow j$. Also the message $msg(m)$, which is sent by process P_5, is also racing toward j. Therefore the race, which occurs at the receive event j, can be denoted as $\langle j, J\rangle$: the first receive event j, $J = \{msg(i), msg(k), msg(m)\}$.

2.2 Related Work

There are several tools for detecting message races such as MAD [8], MARMOT [5,6], and MPVisualizer [2,3]. MAD offers a variety of debugging features such as placement of breakpoints on multiple processes, inspection of variables, an event manipulation feature, and a record&replay mechanism. MARMOT is to verify the standard conformance of an MPI [14,15] program automatically during runtime and help to debug the program in case of problems such as deadlocks, and race conditions. MPVisualizer includes a trace/reply mechanism, a graphical interface, and the engine of the tool which detects and notifies the occurrence of race conditions.

In case of MAD and MARMOT, those tools detect message races just by identifying the use of wild card receives, mpi_any_source, as sources of race conditions.

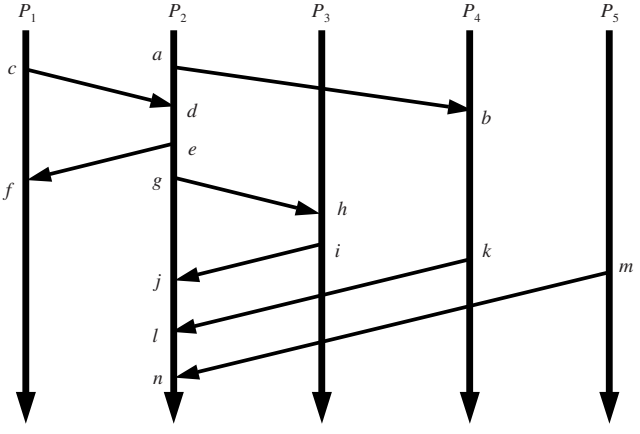

Fig. 1. An Example

In this case the detection result is not correct and also programmers will be overwhelmed by the vast and incorrect information.

Figure 2 shows the cases that there are no race conditions even though receive events are called with mpi_any_source. In Figure 2.(a), process P_1 sends a message to process P_2 with a tag (1). Also process P_3 sends a message to process P_2 with a tag (2). At this time, two receive events in process P_2 are called with mpi_any_source, but with different tags. In this example, even though two send events are concurrent, two messages being sent by processes P_1 and P_3 will be always received deterministically because of the different tags.

In Figure 2.(b), the second receive event in process P_2 is called with mpi_any_source and mpi_any_tag. In this example, however, two messages will be received deterministically because the first message being sent by process P_1 will be always received at the first receive event in process P_2.

In Figure 2.(c), two messages are sent from the same process P_1 and they are received in the process P_2. In process P_2, two receive events receive the messages respectively using mpi_any_source and mpi_any_tag. In this case, there are no race conditions if successive messages sent by a process to another process are ordered in a sequence and if receive events posted by the process are also ordered in a sequence.

As shown in Figure 2, there are no race conditions even though mpi_any_source or mpi_any_tag are used in the receive events. Therefore, if we detect race conditions just by identifying the use of mpi_any_source, that will include wrong detections of race conditions and then mislead programmers.

One the other hand, the method suggested by Nezer [12] can detect more exactly race conditions. This technique focuses on detecting unaffected races [12,13] so that it detects the first race in each process. For this, it requires two executions of a program. In the first execution it checks if a race occurs and

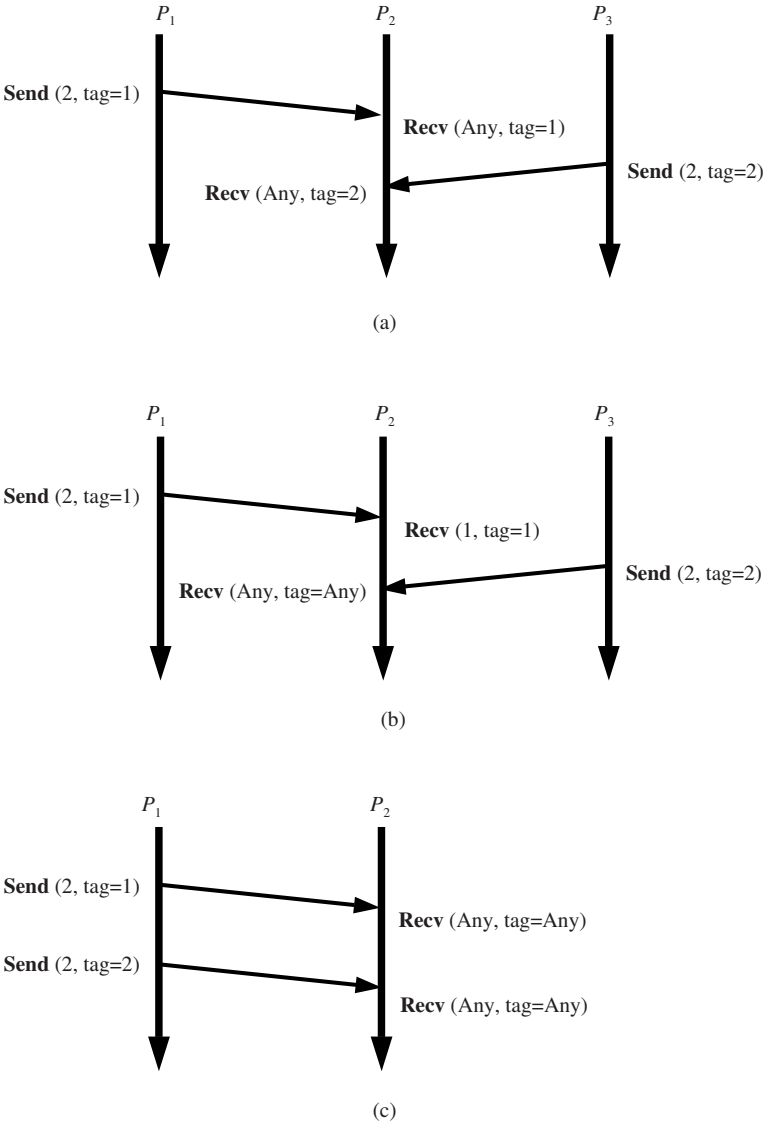

Fig. 2. No Race Conditions with MPI_ANY_SOURCE

identifies the location where the race occurs. In the second execution it halts the execution at the location where the race occurred and then detects racing messages. Even though this technique can detect race conditions more accurately, it is not efficient because it requires two executions of a program.

```
0  TimestampInit()
1  localclock := 0
2  for i from 0 to size do
3      timstamp[i] := 0
4      prevrecv[i] := 0
5      sender[i] := 0
6  end for
```

(a)

```
0  CheckConcurrency()
1  if prevrecv[pid] > sender[pid]
2      report this race
3  end if
```

(b)

```
0  TimestampInSend()
1  localclock := localclock + 1
2  timestamp[pid] := localclock
```

(c)

```
0  TimestampInRecv()
1  call CheckConcurrency()
2  for i from 1 to size do
3      timestamp[i] := max(timestamp[i],
4          sender[i])
5  end for
6  localclock := localclock + 1
7  timestamp[pid] := localclock
8  prevrecv := timestamp
```

(d)

Fig. 3. Algorithms for Timestamp

3 Race Detection

In this section, we explain the methods used in developing MPIRace-Check. First we explain several algorithms to maintain vector timestamps during an execution in order to detect race conditions. Also we show how the algorithms can be called inside of MPI profiling interface.

3.1 Concurrency Check

Vector timestamps [4,9] have been used to determine the "happened before" relations between two events during an execution. Each vector timestamp consists of n values, where n is the number of processes involved in an execution. In this paper, we use vector timestamps to check concurrency between send/receive events in MPI parallel programs. Figure 3 shows the algorithms for maintaining vector timestamps during an execution.

In Figure 3.(a), all variables are initialized with zero: *localclock, timestamp, prerecv,* and *sender*. In the algorithm, *size* is an integer variable and indicates the number of processes involved in an execution. *localclock* is an integer variable for counting the number of events which occurred in each process. This will be incremented by one whenever a send or a receive event occurs.

The variables *timestamp, prerecv,* and *sender* for maintaining the vector timestamps are an array which consists of n elements, where n is the number of processes. Whenever a send or a receive event occurs in a process, *timestamp* will be updated by the current *localclock* during an execution. Only one element of *timestamp*, corresponding to the process itself, will be updated. *sender* will be used for keeping a vector timestamp of a sender which sends a message to the current receive event. *prevrecv* will be used for keeping a vector timestamp of the previous receive event.

Figure 3.(c) shows the algorithm, **TimestampInSend()**, which will be called in each send event. The variable *pid* indicates the current process which sends a message. In each send event, it increments *localclock* by one and sets the element of *timestamp*, corresponding to the current process *pid*, equal to *localclock*. This *timestamp* will be attached to the outgoing message.

Figure 3.(d) shows the algorithm, **TimestampInRecv()**, which will be called in each receive event. In each receive event, first of all, it checks if a race occurs by calling **CheckConcurrency()**. In **CheckConcurrency()**, it checks if the element of *prevrecv*, corresponding to the current process *pid*, is greater than that of *sender*. If then, it means that the message, which was received in the current receive event, can be received in the previous receive event. In this case it reports that a message race occurs.

After calling **CheckConcurrency()**, it updates its *timestamp* using *sender*, which was attached to this incoming message, by the operation **max**(). And it increments *localclock* by one and sets the element of *timestamp*, corresponding to the current process *pid*, equal to *localclock*. For the next receive event, it copies *timestamp* into *prevrecv* because this receive event will become the previous receive event in the next receive event.

Figure 4 shows the vector timestamps in each event when we applied the algorithms to Figure 1. In the figure, *lc* means *localclock* in each event and each *timestamp* in each event is represented with "[]".

In the send event a in P_2, **TimestampInSend()** will be called and *localclock* will be incremented by one. And *localclock* will be set into the element of *timestamp* corresponding to the current process P_2. So *localclock* becomes 1 and *timestamp* becomes [01000]. In the receive event b in P_4, **TimestampInRecv()** will be called and *localclock* will be incremented by one. And *localclock* will be set into the element of *timestamp* corresponding to the current process P_4. Also it updates its *timestamp* using *sender* by the operation **max()**. So *localclock* becomes 1 and *timestamp* becomes [01010]. In this way *timestamp* will be updated and maintained in each event during an execution.

Let us show you how to detect race conditions using *timestamp* in each receive event. For example, in the receive event j of process P_2, **TimestampInRecv()** calls **CheckConcurrency()**. **CheckConcurrency()** compares *prevrecv*, which is the vector timestamp at d of P_2, with *sender* which is the vector timestamp of the send event i of P_3. In this case, *prevrecv*[*pid*], which is "2" from [12000] (*pid* is P_2), is not greater than *sender*[*pid*] which is "4" from [14200]. This means that the message, which was received by the current receive event j of P_2, is not racing toward the previous receive event d of P_2.

On the other hand, in the receive event l of process P_2, *prevrecv* is greater than *sender*. In case of the receive event l, *prevrecv* is at j which is [15200], and *sender* is at k of P_4 which is [01020]. Therefore, *prevrecv*[*pid*], which is "5", is greater than *sender*[*pid*] which is "1" (*pid* is P_2). This means that the message, which was received by the current receive event l of P_2, is racing toward the previous receive event j of P_2. So there is a message race. In this way we can detect message races.

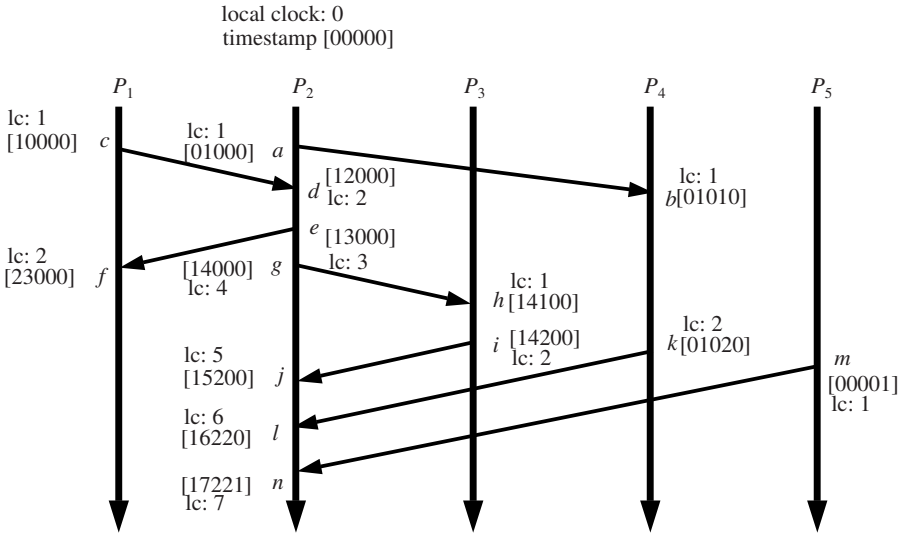

Fig. 4. An Example of Vector Timestamp

3.2 MPI Profiling Interface

MPI Profiling Interface included in MPI specification allows anyone to intercept every call to the MPI library and perform an additional action. For this, the MPI specification states that every MPI routine is callable by an alternative name; every routine of the form MPI_xxx is also callable by the name of the form PMPI_xxx, allowing users to implement and experiment their own MPI_xxx.

For implementing MPIRace-Check, we used MPI profiling interface and we wrapped all point-to-point functions. In each wrapped function, we used MPI_PACK in order to attach a vector timestamp to the outgoing message and we used MPI_UNPACK in order to detach a vector timestamp from the incoming message.

Figure 5 is an example of how we wrapped each function with the algorithms explained before. Figure 5.(a) shows the wrapped MPI_Send function. First it calls **TimestampInSend()** in line 2 and packs the user message(*buf*) and *timestamp* together using MPI_PACK in order to attach *timestamp* to the outgoing message in line from 4 to 5. After that, it calls PMPI_Send.

Figure 5.(b) shows the wrapped MPI_Recv function. First it received a message by calling PMPI_Recv and unpack the message into *sender* and *buf* in line from 4 to 5. After that, it calls **TimestampInRecv()** in order to update its *timestamp* and check if a race occurs.

In this way, we wrapped all point-to-point functions so that users can apply our tool to their programs without modifying their code.

```
0  MPI_Send(buf, count, datatype, dest, tag, comm)
1  {
2      TimestampInSend();
3
4      MPI_Pack(timestamp, size, MPI_INT, buffer, buffersize, pos, comm);
5      MPI_Pack(buf, count, datatype, buffer, buffersize, pos, comm);
6
7      PMPI_Send(buffer, pos, MPI_PACKED, dest, tag, comm);
8  }
```

(a)

```
0  MPI_Recv(buf, count, datatype, source, tag, comm, status)
1  {
2      PMPI_Recv(buffer, buffersize, MPI_PACKED, source, tag, comm, status);
3
4      MPI_Unpack(buffer, buffersize, pos, sender, size, MPI_INT, comm);
5      MPI_Unpack(buffer, buffersize, pos, buf, count, datatype, comm);
6
7      TimestampInRecv();
8  }
```

(b)

Fig. 5. Examples of Wrapped MPI Functions: MPI_Send and MPI_Recv

4 Experimentation

We implemented MPIRace-Check as a library using C language and MPI Profiling Interface so that users can apply our tool to their programs without modifying their source code. Also we used *gdb* to provide detail information for debugging race conditions. When a race is detected, *gdb* will be called within MPI Profiling Interface. To enable this, users have to use the compiler option '-g' when they compile their programs.

In this experiment we evaluated the accuracy and the efficiency of MPIRce-Check. For evaluating the accuracy of race detection, we used MPI_RTED [15] testing programs written in C language. MPI_RTED was developed to evaluate MPI debugging tools. So some of them were designed to have message races to evaluate the ability of detection of race conditions.

Table 1 shows all test programs and the detection results when we applied our tool to MPI_RTED programs. In the table, we can see each name of tested programs, and MPI functions which are used in the testing programs. In those programs, MPIRace_Check detected all races as shown in the table.

Figure 6 shows an error message of our tool when it detects a race in a test program. In the first line, it shows the *localclock*s of the events, and the process number which are involved in the race: P_1 (1) and P_2 (1). In the second line, it shows the channel information, the program name, and its line number: $-2 - 1$, 'c_B_1_1_a_M1.c' and 76. In the third line, it shows the source code

Table 1. The Result in MPI_RTED

Name	MPI Functions	Detection
c_B_1_1_a_M1.c	MPI_RECV	Yes
c_B_1_2_a_M1.c	MPI_RECV	Yes
c_B_1_1_b_M1.c	MPI_SENDRECV	Yes
c_B_1_2_b_M1.c	MPI_SENDRECV	Yes
c_B_1_1_c_M1.c	MPI_SENDRECV_REPLACE	Yes
c_B_1_2_c_M1.c	MPI_SENDRECV_REPLACE	Yes
c_B_1_1_d_M1.c	MPI_IRECV	Yes
c_B_1_2_d_M1.c	MPI_IRECV	Yes
c_B_1_1_e_M1.c	MPI_RECV	Yes
c_B_1_2_e_M1.c	MPI_SENDRECV	Yes
c_B_1_1_f_M1.c	MPI_RECV	Yes
c_B_1_2_f_M1.c	MPI_SENDRECV_REPLACE	Yes
c_B_1_1_g_M1.c	MPI_RECV	Yes
c_B_1_2_g_M1.c	MPI_IRECV	Yes

```
WARNING(RaceCondition):
The message which was sent at '1' from 'P_2' is racing toward '1' receive event in 'P_1'
(the current channel is -2-1)  at c_B_1_1_a_M1.c:76
>>          MPI_Recv(&recvbuf_2, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Fig. 6. An Example of Error Messages

Table 2. Overhead in MPIRace-Check

The number of Send/Recv	Original Run Time (s)	Monitored Run Time (s)	Slowdown
10000	0.168	0.212	26%
100000	1.673	2.234	34%
1000000	16.399	22.034	34%
10000000	164.471	221.736	35%

which is involved in the race: 'MPI_Recv(&recvbuf_2, . . ., &status)'. Using those information, programmers can easily notice whether the race was intended or not, and they can directly modify the bug because they know where it occurs in their source code.

For estimating the efficiency of our tool, we wrote a simple kernel benchmark program. This benchmark program consists of MPI_Send() and MPI_Recv() operations and users can change the number of those operations in the command line. In this program, only a process with the rank 0 receives any messages with mpi_any_source and the other processes send a message to the process with rank 0. To measure the slowdown of MPIRace-Check, we used MPI_Wtime() in the benchmark program.

Table 2 shows the slowdown of MPIRace-Check. For example, when we set the number of send/recv operations 10000, it took 0.168 seconds without our tool. However, the monitored execution by our tool took 0.212 seconds so that

the slowdown is 26%. As we increase the number of send/recv operations, the slowdown does not change proportionally. The worst case in the table shows only 35% slowdown when the number of send/recv operations is 10,000,000. Therefore our tool is efficient as an on-the-fly detection tool.

5 Conclusion

In this paper, we have presented an on-the-fly detection tool, which is MPIRace-Check, for debugging MPI programs written in C language. MPIRace-Check detects and reports all race conditions in all processes during an execution by checking the concurrency of the communications between processes. In our experiment, we showed that MPIRace-Check detects and reports message races using MPI_RTED testing programs as well as our tool is efficient using a kernel benchmark program.

Also our tool provides useful information for debugging such as the line number of a source code, the processes number, and the channel information which are involved in the races. By providing those information, it lets programmers distinguish of unintended races among the reported races, and lets the programmers know directly where the races occurred in a huge source code. Therefore this tool will be useful to develop and debug MPI C parallel programs. In the future we will expand MPIRace-Check to cover all collective routines of MPI-1.

References

1. Cypher, R., and E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives," *8th Intl. Parallel Processing Symp.*, pp. 729-735, IEEE, April 1994.
2. Cláudio, A.P., J.D. Cunha, and M.B. Carmo, "MPVisualizer: A General Tool to Debug Message Passing Parallel Applications," *7th High Performace Computing and Networking Europe*, Lecture Notes in Computer Science, 1593:1199-1202, Springer-Verlag, April 1999.
3. Cláudio, A.P., J.D. Cunha, and M.B. Carmo, "Monitoring and Debugging Message Passing Applications with MPVisualizer," *8th Euromicro Workshop on Parallel and Distributed Processing*, pp.376-382, IEEE, Jan. 2000.
4. Fidge, C. J., "Partial Orders for Parallel Debugging," *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194, ACM, May 1988.
5. Krammer, B., K. Bidmon, M.S. Müller, and M.M. Resch, "MARMOT: An MPI Analysis and Checking Tool," *In proceedings of PARCO'03*, 13:493-500, Elsevier, Sept. 2003.
6. Krammer, B., M.S. Müller, and M.M. Resch, "MPI Application Development Using the Analysis Tool MARMOT," *4th International Conference on Computational Science*, Lecture Notes in Computer Science, 3038:464-471, Springer-Verlag, june 2004.
7. Kranzlmüller, D., and M. Schulz, "Notes on Nondeterminism in Message Passing Programs," *9th European PVM/MPI Users' Group Conf.*, Lecture Notes in Computer Science, 2474:357-367, Springer-Verlag, Sept. 2002.

8. Kranzlmüller D., C. Schaubschläger, and J. Volkert, "A Brief Overview of the MAD Debugging Activities," *4th International Workshop on Automated Debugging* (AADEBUG 2000), Aug. 2000.
9. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7):558-565, ACM, July 1978.
10. Lei, Y., and K. Tai, "Efficient Reachability Testing of Asynchronous Message-Passing Programs," *8th Int'l Conf. on Engineering of Complex Computer Systems* pp. 35-44, IEEE, Dec. 2002.
11. Mittal, N., and V. K. Garg, "Debugging Distributed Programs using Controlled Re-execution," *19th Annual Symp. on Principles of Distributed Computing*, pp. 239-248, ACM, Portland, Oregon, 2000.
12. Netzer, R. H. B., T. W. Brennan, and S. K. Damodaran-Kamal, "Debugging Race Conditions in Message-Passing Programs," *SIGMETRICS Symp. on Parallel and Distributed Tools*, pp. 31-40, ACM, May 1996.
13. Park, M., and Y. Jun, "Detecting Unaffected Race Conditions in Message-Passing Programs," *11th European PVM/MPI User's Group Meeting*, Lecture Notes in Computer Science, 3241:268-276, Springer-Verlag, Sept. 2004.
14. Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1996.
15. HPC Group, MPI Run Time Error Detection Test Suites: http://rted.public.iastate.edu/MPI/, Iowa State University, USA, 2006