Race-Condition Detection in Multicomputer Programs

SeRD-CSCS TN-94-12

June 1994

Abstract: In message-passing parallel programs a type of bug unique to parallel computing can occur. This type of error is due to multi-message *races*. A race occurs when messages are in transit at the same time and the corresponding receives can accept any of the messages. Most of the time, races are incorporated into programs for reasons such as enhancing load balancing. Sometimes a race causes erroneous execution that is hard to debug because of the non-deterministic nature of the program (i.e. if the messages in a race are not received in the same order as during the erroneous execution, the error might not occur). This paper addresses this problem by describing an efficient algorithm for tracing a given program execution and replaying that execution deterministically. The tracing and replaying algorithm will help the user reexecute a nondeterministic program deterministically. This reexecutability will allow a user to gather information on and facilitate removal of all non-deterministic errors.

Michael John Meehan
Computer Science Department
University of North Carolina
CB #3175, Sitterson Hall
Chapel Hill, NC 27599-3175
meehan@cs.unc.edu

SeRD-CSCS Technical Note

Table of Contents

List of Figures

1 Introduction

As part of the CSCS/NEC Joint Collaboration in Parallel Processing, CSCS's Section of Research and Development (SeRD) is developing a tool environment to ease parallelization of applications on distributed memory parallel processors (DMPPs or multicomputers) [?]. The tool environment, known as *Annai*, consists of a parallelization support tool (PST), a performance monitor and analyzer (PMA), a debugging tool for parallel programs (PDT), and a common user interface (UI). In collaboration with a team of researchers developing an HPF system at C&C Research Labs., NEC Tokyo, the tool environment allows profiling and debugging of "low level" MPI [MPI94] message passing programs, and "high level" High Performance Fortran (HPF) [HPF93] programs. In addition, the tools will support extensions to HPF to allow the effective parallelization of irregular finite-element based applications. The two-level profiling and debugging support is eased by the fact that the NEC HPF system and PST are in fact translators that generate Fortran, C, and MPI message passing constructs from high level language input programs.

The race condition detection algorithm described in this paper will be integrated in PDT, Annai's parallel debugging tool. PDT can debug on several platforms, among others our 128 node NEC Cenju-3 DMPP. PDT is currently based on the Free Software Foundation's GNU `gdb` debugger [Sta93], and already allows most of the functionality available in standard debuggers.

A common debugging strategy involves reexecuting a program (on a given input) over and over, each time gaining more information about bugs. Such techniques can fail on message-passing programs. Because of variations in message latencies, different runs on the given input may produce different results. This non-repeatability is a serious debugging problem, since an execution can not always be reproduced to track down bugs.

A message-passing program is non-deterministic if the order of message receipts can vary from one execution to another, i.e., either of two or more messages sent simultaneously can arrive first at some receive point. These messages define a race condition.

Because of such race conditions, a mechanism for tracing and replaying a program's execution is an essential part of a DMPP-program debugger.

This algorithm is a framework for (1) detecting race conditions in a message-passing program by tracing the order in which messages are delivered during a first execution, and for (2) reproducing equivalent executions of the program by using the traces to force each message to be delivered in the same order as during the first execution.

This paper describes an algorithm for tracing and replaying non-deterministic programs. The algorithm was implemented as part of the MPI message passing library. The library will trace the execution and replay it exactly as before. This algorithm involves tracing only the messages that race and storing only those needed to ensure exact replay (not all messages are needed).

The paper begins by giving examples of types of races and characterizes how various tracing and replaying algorithms would handle each. This section also discusses a functionality that exists only in this algorithm. The next section gives a more indepth look at races and and the effect they have on parallel programs.

Sec. 5 describes the tracing and replaying schemes, including the theory and implementation of each. The section describes how the message are traced and the process of determining races by deciding if messages of the same tag *know* of one another (i.e. a message (A) was sent by

a processor that received a message that was the end of a chain of messages that included the message which is begin tested with A). The replaying strategy is described and some additional considerations for MPI (which might be extended to other message passing interfaces) are discussed.

The last two sections of the paper have timing results and conclusions. Two applications and three race-intensive test programs were tested and the results are discussed.

2 Background

2.1 Related Work

There are three basic categories of race detection algorithms: after-the-fact analysis [HW93], compile-time analysis [CS88] and on-the-fly analysis [HMC90], [NM92a]. After-the-fact analysis involves tracing all of the messages in a program and checking for races after program execution. Compile-time analysis involves analyzing program semantics to determine potential races. On-the-fly strategies use information produced during execution to determine which message race.

The algorithm in this paper is an on-the-fly algorithm based loosely on Netzer and Miller's [NM92a] algorithm. Their algorithm uses vector timestamps appended to messages to determine where races occur. The major difference between the Netzer algorithm and the one described here is the FIFO assumption. Netzer and Miller do not assume a FIFO message passing system and therefore recognize some communication patterns as races that are not races if FIFO ordering is guaranteed. FIFO message passing guarantees that multiple messages from the same processor p_0 to the another processor p_1 arrive in the same order that they were sent.

2.2 Supported Platforms

2.2.1 MPI

The race detection algorithm discussed in this paper was implemented in CSCS's subset of MPI. The MPI subset is implemented in three layers. The lowest layer consists of three hardware-dependent basic functions, i.e., non-tagged send, receive and poll operations. The second layer consists of tagged point-to-point communication of contiguous data streams. On top of these, the third layer consists of collective communication routines.

The race detection functionality was implemented on the second layer of the MPI subset. The primitives involved in the race detection algorithm are `MPI_Send`, `MPI_Recv`, `MPI_Ssend` `MPI_Probe` `MPI_Iprobe`. `MPI_Isend` and `MPI_Irecv` also have potential nondeterminism due to a class of race known as *data races* [NM92b] but this is a different genus of races and is not considered in this algorithm.

Race detection and deterministic replay facilities will be included to support PDT. The strategy will be to incorporate deterministic trace and replay into our communication platform. Trace files are generated by the communication library in this format. The library uses these trace files to facilitate deterministic replay.

2.2.2 The NEC Cenju-2 and Cenju-3

The NEC Cenju-3 can be configured with up to 256 VR4400SC RISC processors, each comprising 64 Mbytes memory, 1 Mbyte of secondary cache, and 32 Kbytes primary cache mounted on-chip. The CPUs are MIPS compatible 64 bit processors and are clocked with 75 MHz. They communicate via a packet-switched multi-stage interconnection network composed of 4×4 crossbar switches. The machine is hosted by one or more VR4400SC-based workstations (NEC EWS4800). For more details on the Cenju-3 architecture see [?]. The Cenju-2 is an experimental prototype of the Cenju-3. It is based on MIPS R3000 processors and uses a similar network. The machine installed at CSCS features 16 processors and also 64 Mbytes memory per node.

Another supported platform is a simulator on Solaris based workstations. With this simulator true parallelism can only be achieved on multi-processor based Sparc systems. The Solaris simulation is very useful during the early stages of application development because direct access to the generally expensive parallel systems is not necessary.

3 Examples

To compare my algorithm with some previously developed, I will show two example races and how different algorithms would handle them. I will compare Miller and Netzer's [NM92a] frontier tracing and the tradition algorithm of tracing all messages with my own method. The first example will show that tracing all messages is sufficient to replay a race condition deterministically but is not necessary. The second will show that in message passing systems that allow message buffering that *block races* must also be taken into account. In both of the following examples, each message is acceptable for any receive. (i.e. the receives considered can accept from each of the sources and any of the tags of the racing messages).

3.1 Example 1: n messages from n processors

Fig. 1 depicts three messages racing for a set of receives. Processors 0, 2, 3 send messages `Msg0.1`, `Msg2.1`, `Msg3.1` respectively. The order that processor 1 receives the messages in this case is: `Msg0.1`, `Msg3.1`, `Msg2.1`. In the case of *all message tracing*, the replay information fora each of these messages would be stored. This will give correct replay results, but storing the information for all of these messages is unnecessary. The information for `Msg2.1` is superfluous. Upon replay, the algorithm will ensure that `Msg0.1` and `Msg3.1` are replayed in the correct order. This being the case, `Msg2.1` will automatically arrive at the correct receive. Therefore, there is no need to store replay information for the effectually deterministic `Msg2.1`. My algorithm and the frontier algorithm will trace this race and store replay information for only the first two messages, since only these are necessary for deterministic replay.

In general, if n messages from n processors are racing, then the information for the first $n-1$ messages is sufficient and necessary for deterministic replay.

3.2 Example 2: n messages from m processors $(n > m)$

Fig. 2 characterizes a type of race which occurs when multiple processors send multiple messages to a given processor. Processors 0 and 2 are each sending three messages to processor 1. FIFO

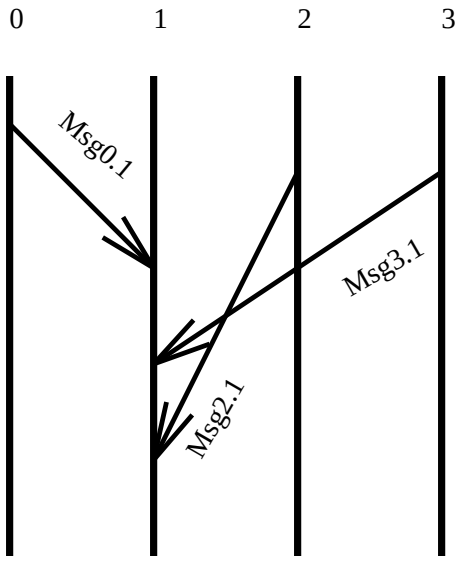

Figure 1: *Simple Three Message Race*

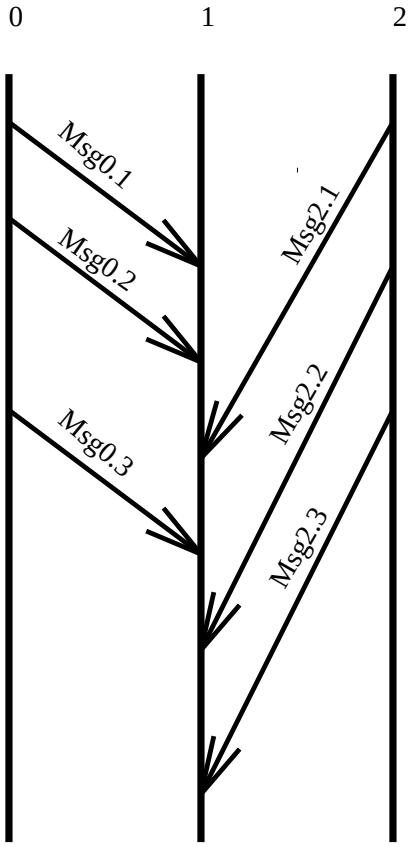

Figure 2: *Block Race Example 1*

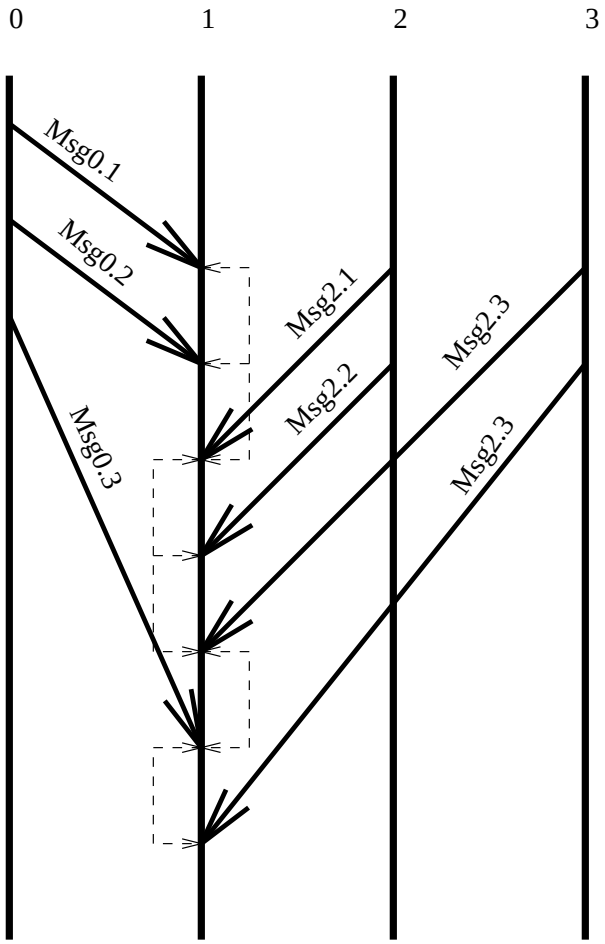

Figure 3: *Upper Bound: m=3, n=7, Num. Traced=6*

message control is assumed (i.e. messages of the same tag from the same processor to a given processor cannot overtake each other). Because the messages are FIFO, the order of arrival of messages from processor 0 is guaranteed (the same holds true for the messages from processor 2). The messages from a given processor have a guaranteed order, but the messages from different processors can be interleaved in any way.

I will not go over the traditional trace-all method as it has the same drawbacks described above. The frontier method will trace the messages and detect races in `Msg0.1` $\leftrightarrow$ `Msg0.2`, `Msg0.2` $\leftrightarrow$ `Msg2.1`, `Msg2.1` $\leftrightarrow$ `Msg0.3` and `Msg0.3` $\leftrightarrow$ `Msg2.2`. This would constitute storing the replay information for `Recv1.2`, `Recv1.3`, `Recv1.4` and `Recv1.5`. The algorithm described in this paper would detect that `Msg2.1` races with `Msg0.1` *and* `Msg0.2` (a *block race*) as well as the races between `Msg2.1` $\leftrightarrow$ `Msg0.3` and `Msg0.3` $\leftrightarrow$ `Msg2.2`. Therefore it would store the replay information for `Recv1.1` $\rightarrow$ `Recv1.4`, which is necessary and sufficient for correct reexecution. The algorithm would not store the information for `Recv1.5`. FIFO message passing guarantees that `Msg2.2` will arrive at `Recv1.5`.

In general, for n racing messages from m processors ($n > m$) it is necessary to trace at most $n - 1$ messages as shown in Fig. 3 and at least $m - 1$ as shown in Fig. 4.

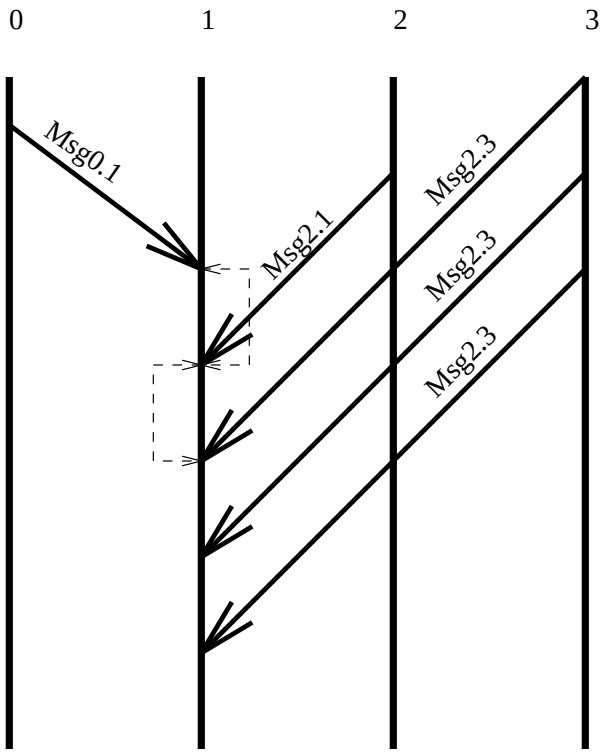

Figure 4: *Lower Bound: m=3, n=5, Num. Traced=2*

4 Characteristics

In this section, I will give a more formal definition of a race. The following subsections will describe what it means for messages to race, define some notation, and give a definition for a race.

4.1 Concept of a Race

A race occurs between messages if each of the messages could theoretically be accepted by the same receive. In a program that embodies a number of races, with each instance of the program's execution, the receives associated with the races might accept messages in a different order.

Fig. 5 is a pictorial example of racing and non-racing messages. In this figure, `Msg2.1` and `Msg0.1` race. That means that either `Msg2.1` or `Msg0.1` could have been received by either `Recv1.1` or `Recv1.2`.

`Msg2.2` on the other hand does not race with any other messages in this execution model. `Msg0.1` and `Msg2.1` have to be received by either `Recv1.1` or `Recv1.2` before `Msg1.1` is sent to `Recv2.1`. Only after `Recv2.1` can `Msg2.2` be sent. Therefore, `Msg2.2` will *always* be received after `Msg0.1` and `Msg2.1` and does not race with them.

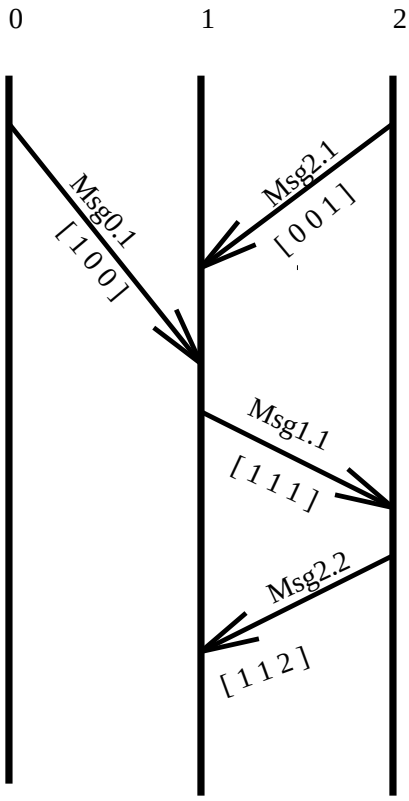

Figure 5: *Racing and Non-Racing Messages*

4.2 Race Notation

Here is described some notation used in this paper for more concise description of race detection algorithms. `Msg`p`.`n refers to the nth message sent from processor p. Similarly, the notation `Recv`p`.`n refers to the nth receive on the pth processor.

All figures used in this paper show races which occur at the beginning of the program's execution. In reality the choice of location is arbitrary. The figure can be taken as the beginning of the racing program's execution, or it could equivalently be anywhere in the execution as long as no messages before those depicted in the figure race with any of the messages in the figure. This is similar to Miller and Netzer's *frontier* concept.

4.3 Definition of a Race

All of the figures in this paper refer only to receives that can legally accept any of the messages sent. In the MPI message passing systems, messages have a tag. The tag is a user defined value that lets the receiver know what type of message it is. Receives not only have a restriction as to what tag they can receive, but also what processor they accept a message from.

In reality, for messages to race they must have the same tag, or the corresponding receives must be able to accept the tags of all racing messages. In MPI, messages of different tags can race if the corresponding receives have the `MPI_ANY_TAG` wildcard as its tag parameter. It is also

necessary for all of the receives associated with the race to have the ability to accept from any of the sources associated with the racing messages. If a receive can only accept messages from a particular source, then the FIFO-ality of the system will deterministically guarantee the arrival of a particular message, and the associated messages could not be part of a race.

Therefore, the definition of a race is simple: messages race if they can be simultaneously in transit and the associated receives can accept any of the messages. The method for determining if messages can be simultaneously in transit is described in Sec. 5. For a set of receives to be able to accept any of a set of messages, the receives must be able to accept messages from all of the *sources* of the racing messages and the receives must be able to accept messages having any of the *tags* associated with the given messages.

5 The Algorithm

This section describes the algorithm for run time race detection. The first subsection describes the method for detection of races. The second subsection describes how the information from tracing with vector timestamps is used for deterministic replay. The third describes additional tracing considerations for the MPI message passing interface.

5.1 Tracing and Detecting Races

5.1.1 Theory

For ease of explanation, the discussion of races will begin with examples involving only two messages and later be generalized to more messages. To determine if two messages race, one has to determine whether both messages can be accepted by the same receives and if they could be in transit at the same time. To determine the former, a check must be made to ensure that both of the receives could have theoretically accepted both messages. In MPI for example, both of the receives must use the `MPI_ANY_SOURCE` wild-card for its designated source. If communication blocks are implemented, then the two messages must also have the same communication block. (A communication block causes receives to accept messages from a subset of all of the processors). A check must also be performed to ensure that both receives can accept both messages with respect to tags. In MPI, this means that either the messages are of the same tag and both receives can accept this tag or both receives use the `MPI_ANY_TAG` wildcard as their tag designator.

To determine if two messages could be in transit at the same time one has to determine whether the second message received *knew* about the first. One message can be simultaneously in transit with another only if they do not know of each other.

For one message to *know* of another, its processor must have received the last of a chain of messages including the *known* message before it sends the *knowing* message. This is illustrated in Fig. 5. `Msg2.2` knows of `Msg0.1` because processor 2 received the end of a chain of messages (`Msg0.1`, `Msg1.1`) that included `Msg0.1` into `Recv2.1` before sending `Msg2.2`. Since `Msg2.2` knew of `Msg0.1` there is no race and no information is stored. It can be easily seen that `Msg0.1` and `Msg2.1` do race since the second does not know of the first.

As pointed out by Miller and Netzer [NM92a] only the information for the first of the two racing

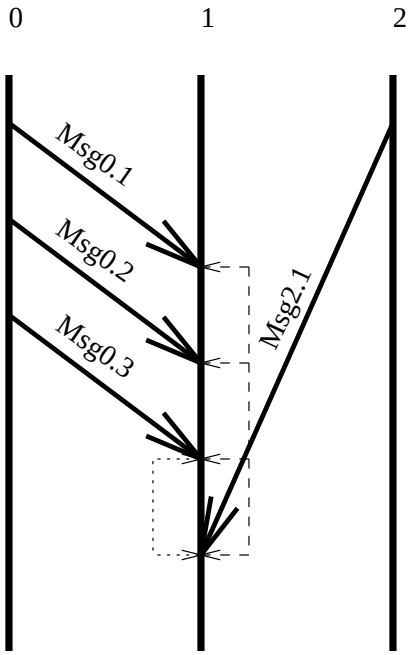

Figure 6: *Block Race Example 2*

messages needs to be stored for deterministic reexecution of the race. If the first of two racing messages is forced to the correct receive upon reexecution, then the second will automatically go to the correct receive (because there will be no other receive it can go to). In Fig. 5, if `Msg2.1` is forced to `Recv1.1` then `Msg0.1` has no option except to go (correctly) to `Recv1.2`.

For simple races involving n messages, only the information for the first $n-1$ need to be saved (for *block races* see below). Fig. 1 shows a triple race between `Msg0.1`, `Msg2.1` and `Msg3.1`. If the information for `Msg0.1` and `Msg3.1` (the first two messages to be received) is stored and the reexecution order of these two messages is guaranteed, then `Msg2.1` will go to `Recv1.3` as desired.

A *block race* is another class of races. Block races are races between a message and a set of messages. If a message does not *know* (in the same context as described above) of a set of messages, then it is in a race with them. Fig. 6 shows a simple block race. `Msg2.1` does not *know* about `Msg0.1`, `Msg0.2` or `Msg0.3`, therefore it is in a race with all of them. Since `Msg2.1` races with all of these messages, trace information `Recv1.1`, `Recv1.2` and `Recv1.3` must be stored.

It might not appear that the block race detection is necessary, but Fig. 7 shows a potential problem if only simple (non-block) races are traced. If only the simple races are traced, then the only race detected is the one between `Msg0.3` and `Msg2.1`. The only replay information stored would be that `Recv1.3` needs to receive `Msg0.3`. It is easy to see that since `Msg2.1` could be accepted by `Recv1.1` or `Recv1.2` which would would cause erroneous execution.

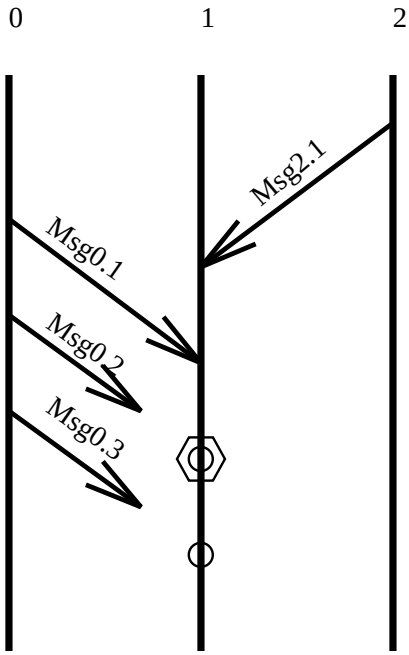

Figure 7: *Block Race Example 2: Erroneous Replay*

5.1.2 Implementation

To implement the concept of *knowledge* between messages I used a technique similar to Netzer and Miller's *vector timestamp* [NM92a]. They suggest that each message should have the sending processor's vector timestamp appended to the end of it. Each processor has an internal clock which increments upon each event in the processor. By definition, for a processor p, the pth value of the vector timestamp is equal to the internal clock. The rest of the vector timestamp for each processor is determined by doing a component wise maximum on its current timestamp and any timestamp appended onto an incoming message.

Given two incoming messages a and b , the first arrived from processor p_a the second from p_b and their vector timestamps V_a and V_b, one can determine if they race (i.e. the second message doesn't know of the first) by comparing the p_ath value of the vector timestamps. If the p_ath value of V_a is higher than the p_ath value of V_b, then the two messages race. This is easy to understand. For message b to know of message a, then the p_ath value (a's internal clock) would have to be incorporated into message b's timestamp (because the p_ath value would be passed along through the chain of messages linking the two).

Fig. 5 demonstrates this. `Msg2.1` arrives before `Msg0.1` so the 2nd value of their timestamps must be compared. The 2nd value of `Msg2.1`'s timestamp is 1 and the 2nd value of `Msg0.1`'s timestamp is 0, therefore the two messages race. Considering `Msg2.2` and `Msg0.1`, we see that the 0th value of the `Msg0.1`'s timestamp is 1 and the 0th value of `Msg2.2`'s timestamp is also 1, so the second message knows of the first one, therefore they do not race.

To detect block races, a buffer of timestamps from recently received messages must be stored. Such a buffer of timestamps must be stored for each tag. The buffer does not have to store all past timestamps. A timestamp can be taken out if a message of its tag has been received from

every processor (excluding the timestamp's source and receiving processor). Cause for exiting on this condition can be explained as follows. If a message of the same tag is received from another processor, then the message either races with the current message or it doesn't. If the new message races with the old, then the old message's trace information is stored and the old timestamp is removed from the buffer. If the new message does not race with the old, then the new one's timestamp is added to the buffer and the old one remains. If a message of the same tag has been received from every processor and the message is still in the buffer, then it is impossible for another message to arrive that would race with the message in question.

In addition to appending the timestamp, every message also has its sending processor number appended onto the end of the message. This information will be written to the trace file. Its use is described in the next section.

5.2 Deterministic Replay

If there is a bug in an application, the user would need to replay the program exactly as it played before in order to gather more information about the error. If the error is proceeded by a race condition, then successive replays may or may not recreate the bug. In order to guarantee identical execution, the program must be deterministically replayed (i.e. all races in the program must have the same order of arrival of messages as in the first execution). This can be accomplished by using the replay algorithm described in this section.

5.2.1 Theory

To replay a race condition exactly as before, one needs to know which messages should arrive at all critical receives and should be able to hold off reception until the correct message arrives. To determine which messages race, given that the original execution has been traced correctly (as described above), one simply needs to read in the trace file which will indicate all critical receives and the correct messages for each.

The only piece of information necessary to determine the correct message for a critical receive is the sending processor's number. Because we are assuming a FIFO machine, we are guaranteed the order of arrival of messages from a given processor. Assuming that all messages up a certain critical receive arrive in the correct order, then the next message to come from the critical receive's desired source will be the correct one.

5.2.2 Implementation

For a processor to know which message to receive, it needs to know which receives are critical and which messages should be received at each. To know which receives are critical, the trace file must (for each processor) supply the number corresponding to the internal clock of the original receive. To know which message should be received by a critical receive only the original sender's source number is needed.

To implement reexecution, the program reads in the trace file and determines what the next critical receive is. The program will execute without any interference until the first critical receive (since all of the preceding receives are already deterministic). Upon the first critical receive, the program will block until the correct message arrives. This process is repeated again

for the next, and all proceeding critical receives.

One should note that this does not guarantee the order of *arrival* of messages, it simply guarantees the order of reception of the messages. The messages can still arrive in any order and must be buffered until their correct reception time.

5.3 Considerations

Races can occur in more operations associated with message passing than just sends and receives. If messages are racing to a given processor and a probe is done on the receiving processor, then a race is actually occuring at that probe. If any of the information from the probe is used in the program (i.e. the messages source, length, et cetera), then the race affects the program.

Blocking probes are traced in the same manner as receives. To trace them correctly, all of the operations for tracing a receive must be performed except for the removal of the message from the message buffer. The incoming timestamps are compared to those of the receives and other blocking probes, stored and removed from the buffer upon the same conditions and stored identically in the trace file if they race.

If a user chooses to trace blocking probes, (the user is given the choice in my implementation) then the every action that is performed for a receive, is also performed for each blocking probe, including internal clock incrementation for each blocking probe occurrence. One should note that since the blocking probes are treated exactly as blocking receives are by the algorithm, the same internal clock is used for both. If on a processor p there have been two blocking receives and three blocking probes, then the internal clock would be equal to 5.

Non-blocking probes (such as `MPI_Iprobe` which I will refer to as Iprobe here) can be the cause of non-deterministic behavior. As with blocking probes, if any of the information from the Iprobe is used in the program (i.e. source, length, et cetera), it can cause non-deterministic behavior. Non-blocking probes cannot be traced in the same manner as blocking probes and receives, since they are simply checks for existence of message. Instead, the outcome for every non-blocking probe must be recreatable from a trace file.

Keeping the trace information for every Iprobe would be very expensive. One would have to store each Iprobe's outcome (true or false) and the message's source if the Iprobe was successful and was of a class that can detect messages from more than one source. Upon investigation it was discovered that for the set of programs that had Iprobes, less than one percent of those probes were successful. Therefore, it is most efficient to store only the information for the successful Iprobes.

For deterministic replay of the non-blocking probes, the Iprobe trace is read into a buffer. Every time an Iprobe is called, the Iprobe event counter is incremented. The Iprobe event counter and trace files are separate and distinct from those that used for the blocking receives and probes. Regardless of the existence of messages in the buffer, the replay mechanism will force the Iprobe to return a value of false until the Iprobe counter matches the value for the next true Iprobe. When the counter does match, and a true value has to be returned, the Iprobe will block until it has received the correct message, and then it will return a true value. Once again, only the source number needs to be known for the critical Iprobes. If the Iprobe has the correct source, then FIFO-ality will guarantee that the correct message arrives.

Something to take into consideration is the number of message timestamps allowed in the timestamp buffer. Since a comparison has to be made between the current timestamp and all old timestamps of its tag, keeping too many timestamps would slow down the trace execution of the program considerably.

For two racing messages to have a non-deterministic receive order, their order of arrival would have to vary from execution to execution. One might be able to assume this implies they would have to be in transit at nearly the same time during all executions. For two messages to be in transit at the same time, they must be in the message buffer at nearly the same time. By experiment, I have found that two times the number of messages allowed in the message buffer works as a timestamp buffer limit. Intuitively, this seems reasonable since races that need to be addressed should have receives that are relatively close. For the Cenju-2 and Cenju-3 the message buffer can hold 64 messages. This means 128 timestamps would be stored in the timestamp buffer. For the Solaris version of MPI, only 32 messages are stored in the message buffer.

This limitation is not based entirely on the logic of race detection, and there is a potential for important races to be skipped (in which case the buffer size should be increased). This is simply a method of reducing the number of message in the buffer, and therefore the execution time of tracing. In reality, messages (unless taken out because they race with another or because a message of their tag has been received from every other processor) could stay in the buffer for the duration of the program and continue to have potential to race with another message. If two messages are not received at nearly the same time, even if they race, they are not likely to be received in subsequent executions in a different order. In the race detection application, the number is automatically set to two times the length of the message buffer, but can be reset if desired.

6 Results

The race detection algorithm has been designed to complement the MPI message passing interface. All test programs were run using the modified MPI library. This section discusses the results of three race-intensive programs and two sample programs.

The three race-intensive programs were run on the NEC Cenju-2 using 4 processors. All three programs have the same functionality, but were implemented with a different series of MPI calls. The three programs use a different set of MPI calls to implement the functionality of an `MPI_Recv` with an `MPI_ANY_SOURCE` designator. The results shown here are averages of 5 test runs. In the programs, each processor sends 500 messages to each of the other three processors and receives 500 message from each of the other three processors. All 1500 messages received at each processor are involved in at least one race. From these 1500 messages, 1499 need to be traced.

6.1 Three race-intensive programs

The first implementation used an `MPI_Recv`.

```
MPI_Get_source(status, &src_now);
MPI_Recv(&recv[1], 1, MPI_INTEGER, MPI_ANY_SOURCE, DONE, MPI_COMM_WORLD, &status);
```

Implementation	Function	No Race Detect	Race Tracing	Race Replaying
MPI_Recv	MPI_Send	42	50	43
	MPI_Recv	120	249	195
MPI_Probe	MPI_Send	36	48	38
	MPI_Recv	35	82	51
	MPI_Probe	35	82	51
MPI_Iprobe	MPI_Send	38	50	38
	MPI_Recv	87	199	143
	MPI_Iprobe	8	15	18

Figure 8: *Average Times (in Micro Seconds) for MPI Calls in the Three Implementations of MPI_Recv*

Program	Timing	No Race Detection	Race Tracing	Race Replaying
TSP	Execution Time	5.115	5.786 (13%)	8.662 (70%)
	File I/O Time	0.0	4.5088 (88%)	5.161 (101%)
	Total Time	0.0	10.295 (101%)	13.823 (170%)
2DFFT	Execution Time	4.636	4.704 (1%)	4.642 (<1%)
	File I/O Time	0.0	3.464 (75%)	4.709 (102%)
	Total Time	0.0	8.168 (76%)	9.351 (102%)

Figure 9: *Average Execution Times (in Seconds) for Traveling Salesman Program and Two Dimensional FFT*

Fig. 8 shows the average execution times for the `MPI_Sends` and the `MPI_Recvs` for non tracing, tracing, and replying executions. The increase in the amount of time needed to implement a send is due to the appending of timestamps onto the end of each message. For replay, only a counter must be incremented and therefore not much additional execution time is needed. The additional time to receive during tracing is due to taking the timestamp off the message and checking for races. The additional time needed to receive during replay is simply due to bookkeeping and checks to see when the next racing message will arrive.

The second implementation used an `MPI_Probe` and an `MPI_Get_source` to find out where the next message was from and received it with an `MPI_Recv`:

```
MPI_Probe(MPI_ANY_SOURCE, DONE, MPI_COMM_WORLD, &flag, &status);

MPI_Get_source(status, &src_now);
MPI_Recv(&recv[1], 1, MPI_INTEGER, src_now, DONE, MPI_COMM_WORLD, &status);
```

The average send and receive times for the non-tracing, tracing and replay executions are similar to those listed above. The execution time differences for the `MPI_Recvs` and `MPI_Sends` for the three types of executions are the same for all three test programs, so they will not be mentioned

again.

Fig. 8 shows the time needed for the `MPI_Probes`. The additional execution time needed during tracing is due to the reception of the message and race checking with the timestamp. The message must actually be received (but not removed from the message buffer) so that the timestamp can be taken off. During replay, additional time is needed for execution because the probe must wait for the correct message from the correct processor before it can return (instead of returning as soon as any message arrives).

The third implementation used an `MPI_Iprobe` inside a while loop and an `MPI_Get_source` to find out where the next message was from and received it with a `MPI_Recv`:

```
MPI_Iprobe(MPI_ANY_SOURCE, DONE, MPI_COMM_WORLD, &flag, &status);

while (!flag) {
  MPI_Iprobe(MPI_ANY_SOURCE, DONE, MPI_COMM_WORLD, &flag, &status);
}

MPI_Get_source(status, &src_now);
MPI_Recv(&recv[1], 1, MPI_INTEGER, src_now, DONE, MPI_COMM_WORLD, &status);
```

Fig. 8 shows the time needed for the `MPI_Iprobes`. Even though the intrusiveness of the race detection algorithm is very small in the Iprobes, it is proportionally large because of the quick execution of the subroutine. The tracing overhead is due to the checking and storing of the results in the buffer. Most of the replay overhead is caused by waiting for the correct message when the Iprobe needs to return a true value.

6.2 Traveling Salesman Program and Two Dimensional FFT

Fig. 9 shows the slowdown of a parallel implementation of the traveling salesman program for 16 cities using 16 processors of the NEC Cenju-3. One can see that the majority of the overhead on tracing and replaying is due to file I/O. The additional 13% execution time for race tracing is due to appending, receiving and buffering timestamps and checking for races. The time to write the trace files adds an additional 88% to the execution time, producing a total slowdown of 101%.

The 70% slowdown in execution during replay is due to `MPI_Recv`s and `MPI_Iprobe`s waiting for the correct message to arrive. The file I/O adds another 101% to the execution time giving a total slowdown of 170%.

The TSP program is an extreme example of the slowdown due to the race condition detection algorithm. The parallel two dimensional FFT program has no races and no non-determinism. The slowdown data for this program is included to give users an idea of the performance for programs that have few or no races. Fig. 9 shows that the tracing and replaying have nearly the same execution times as the standard MPI library. The slowdown was 1% for tracing and less than 1% for replaying. The file I/O added 75% and 102% for tracing and replaying respectively. This file I/O time is constant for programs with no races (since no race data has to be written out), therefore if the problem size is increased, the percentage slowdown will decrease.

7 Conclusion

An algorithm for efficient race detection tracing and deterministic replaying has been described in this paper. The algorithm uses a method of vector timestamping to provide the information necessary for the algorithm to determine whether messages race with each other. Additional time is needed for execution of a program using race detection (either tracing or replaying). The additional time needed is a proportional to the number of races in the program being tested and in the worst case should not be enough to deter a programmer from using the algorithm's implementation as a debugging tool.

The algorithm assumes the message passing system is FIFO and uses that knowledge to trace the optimal number of messages.

8 Directions

This algorithm should be ported to the Annai debugging environment. It would also be useful to develop an incremental replay tool which uses the race detection tracing information in addition to computer state information to replay from a given point in a program's execution. A widget to show the critical program regions and the pattern for the racing messages for the UI would also prove useful for debugging.

9 Acknowledgments

I would like to thank Roland Rühl and of Christian Clémençon for all of their help and guidance through the development of this algorithm. I would like to thank Will Sawyer, Karsten Decker and all of the people at CSCS for making it possible for me to be involved in the research activities at CSCS. Finally, I would like to thank all of my SSIP piers for their academic and non-academic support over the duration of this program.

References

[CS88] D. Callahan and J. Sublock. Static analysis of low level synchronization. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[HMC90] Kennedy K. Hood, R. and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing '90*, November 1990.

[HPF93] HPFF (High Performance Fortran Forum). High Performance Fortran Language Specification: Version 1.0. *Scientific Programming*, 2(1&2), 1993. Special Issue.

[HW93] McDowell C. Helmbold, D. and J-Z Wang. Determining possible event orders by analyzing sequential traces. In *IEEE journal on Parallel and Distributed Systems*, 1993.

[MPI94] MPIF (Message Passing Interface Forum). MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, May 1994. Version 1.0.

[NM92a] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.

[NM92b] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.

[Sta93] R. M. Stallman. *GDB Manual: The GNU Source Level Debugger*. Free Software Foundation, 1993.