BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M11

"The Performance of Two Tracing and Replay Algorithms for Message-Passing Parallel Programs"

by

Weihua Yan

The Performance of Two Tracing and Replay Algorithms for Message-Passing Parallel Programs

Weihua Yan

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirement for the Degree of Master of Science in the Department of Computer Science at Brown University.

May 1995

Professor Rob Netzer
Adviser

The Performance of Two Tracing and Replay Algorithms for Message-Passing Parallel Programs

Weihua Yan
Department of Computer Science
Brown University
Providence, RI 12912-1910
why@cs.brown.edu

May 26, 1995

Abstract

Debugging parallel message-passing programs is complicated by the non-determinism that is inherent in those programs. Cyclical debugging, which is a proven method for sequential programs, often fails when debugging parallel programs because different executions of the same program may exhibit different behaviors due to non-determinism. Some approaches have been studied to remedy this problem. We implemented and compared two algorithms, both are termed Trace and Replay. The first algorithm we study here traces the order of all messages delivered in a program. Using this information, it forces a program to re-deliver a message to the same receiver as in the traced execution. The second algorithm improves on the first algorithm in that it attempts to trace only those messages that race with others. Since in most programs, only a small percentage of messages race and cause non-deterministic execution, this algorithm will reduce the overhead of tracing that plagues most trace and replay schemes. Both algorithms are implemented in PVM (Parallel Virtual Machine) and transparent to the user.

1 Introduction

Cyclic debugging is a well understood methodology: we execute a program multiple times and examine the executions using breakpoints or print statements, until an error manifests itself. However, such a method often fails in debugging parallel programs. The inherent, unpredictable behaviors of parallel processes make one execution of a distributed program often differs from later executions, even on the same input. Thus, an error that occurs in an earlier execution may not manifest itself in later executions. Worse, attempts to examine the program execution by using breakpoint or similar method may contribute to the difficulty of reproducing the erroneous execution.

In programs that communicate by passing messages, for example, two messages intended to the same processes may be received in different orders; or messages that can be received by a number of processes may be delivered to different processes in different executions. This nondeterminism renders traditional debugging tools for sequential programs powerless in debugging parallel programs.

Many approaches have been studied on this subject. One of the common approaches is to record event histories[3]. The debugger does little but recording information on the running program. The recorded information can then be analyzed after the execution of the program. The amount of information needs to be collected depends on how it is going to be used. There are three levels of use: *browsing*, *replaying*, and *simulation*. The amount of information increase for each method above.

Browsing requires only minimal information about the program. Sometimes simply recording the kind of events executed by a program can lead to an error. One disadvantage with browsing is that the event histories often contains large numbers of events and it is difficult to locate the events of interest. Simulation is to debug a single process while simulating rest of the program. This is meaningful because the execution of a single process is deterministic given that rest of the program will behave

deterministically. This approach requires enormous amount of information. It requires more information than re-executing the entire system because we need to record all input and output contents[3]. The third approach, Replay, requires reasonable amount of information compared to simulation, and it is much more effective than simple browsing. Therefore, it is gaining more appeal recently. We will discuss two replay algorithms in the later sections.

Another approach is to record only the event histories of interests instead of the entire history[5]. The debugger takes a snapshot of the program's state and keeps only the information after the snapshot. This approach is useful for long running programs which would require recording too much information for the three methods discussed earlier. The disadvantage of this approach is that it is difficult to obtain accurate snapshots in distributed systems effectively[3].

We experimented two trace and replay algorithms. The first algorithm, termed Naive Algorithm, attempts to trace all messages passed in a program. The second algorithm, termed Adaptive Algorithm, only traces a message if it "races" with another message. We implemented both algorithms with PVM (Parallel Virtual Machine). We ran a suite of programs and compared their running time with non-instrumented versions. We also recorded number of messages traced by both algorithms. Our result shows that both tracing algorithms incur relatively small execution time overhead. However, the Adaptive Algorithm performs relatively better than the Naive algorithm for tracing executions. Replaying incurs similar run-time overhead. In general, tracing and replay executions have less than 10% of execution time overhead on all of our programs experimented. Thus, we conclude that both algorithms are very practical. Adaptive Algorithm also has advantage over the Naive Algorithm in terms of the size of the trace files. In our experiment, only a fraction of the total messages are traced in most of our programs. Since trace file size is often the bottleneck in such trace and replay scheme[1], the Adaptive Algorithm is of considerable improvement over the Naive Algorithm.

However, in our current implementation, to implement replay, the Adaptive algorithm must sort and collate trace files from each process. Further research is needed to solve this problem.

2 Replay

In message-passing distributed programs, processes communicate by passing messages. Thus variation in message latencies, operating system scheduling, and unpredictable network delays may cause multiple executions of the same program to exhibit different behavior, even on the same input[1]. Thus replaying is essential to debug distributed program.

2.1 What is replay?

Replay is the deterministic re-execution of a distributed program. The debugger records information during the execution of a program. It can then use this information to control the re-execution of the program later. This permits the traditional debugging techniques, such as breakpoints, stepping, state examination using print statement, etc. without changing the behavior of the program.

The general method is often called trace and replay. There are two stages in a typical replay algorithm. The first step is tracing. During tracing, messages that are delivered in all processes are traced and the result is written to a file. The second step is replay -- re-execution of the program being debugged. During replay, the traced information can be used to force messages to be delivered in the same manner, to the same process(es) or in the same order, as in the previous execution.

2.2 Previous work

A number of work have been done to solve the nonreproducibility of distributed program. One approach is to log the contents of messages in an event log when it is

received. The log can be reviewed to find errors, or better the log can be used as input to replay the execution of the program[4]. The disadvantage of such method is that the amount of data to record is often prohibitively large for long running programs. This often limits the use of such scheme in a long running program where many messages are passed. The second problem is that recording contents of all messages often disturb the execution of a distributed program, and therefore may hide some errors that related to messages. This is often refer to as "Probe Effect"[3].

Another approach, termed Instant Replay, recognizes the fact that to implement replay, one only has to make sure that the order of messages received in replay is the same as in the traced execution[2]. The contents of the messages will be automatically regenerated during the execution. Thus this approach only traces the order of messages received, and it is much more cost-effective than logging the contents of all messages since it only records a small amount of information for each message. Such an algorithm reduces both time and space overhead of tracing and replay, and also alleviates the "Probe Effect". This approach is used in our tracing algorithms.

3. Replay Algorithms

We implemented two tracing and replay algorithms. The first one is similar to Instant Replay algorithm described by LeBlanc and Melor-Crummy[2]. The second algorithm is based on Rob Netzer's Optimal tracing and replay algorithm[1].

The first algorithm attempts to trace all synchronizing events, namely, all messages that are received by a process. The traced information is written to a file. At replay execution, the trace file is read, and the a process will only accept a message if it agrees with the trace file; otherwise, it will buffer the current message and wait for next one. We will refer to this one as Naive algorithm.

The second algorithm we implemented is termed Adaptive algorithm. Instead of tracing all messages received by a given process, it attempts to differentiate messages that race from those that do not race. The non-racing messages will be delivered deterministically in every run, and therefore need not be traced. The racing messages, on the other hand, must be identified and traced so it will be delivered to the same receiver in every re-execution.

It has been shown that in many distributed programs, the number of racing messages are relatively low compared to total messages[1]. Thus the adaptive algorithm reduces the size of trace file and therefore reduces the space overhead of tracing. Since trace file size is often the bottleneck in tracing long running programs, the Adaptive Algorithm should alleviate the bottleneck and allow even long running programs to be replayed that could not have been replayed previously.

3.1 Naive tracing and replay algorithm

The Naive algorithm is adapted from Instant Replay algorithm by LeBlanc and Melor-Crummy. The Instant Replay algorithm is devised for closely-coupled share-memroy parallel programs, and it requires instrumenting user's code to effect the tracing and replay[2].

Figure 1 shows the pseudo code for the Naive Algorithm. The crux of the Naive Algorithm is that it only traces the order that messages are received by each process.

During replay, contents of the messages will be recomputed and therefore need not be recorded. Tracing is straightforward. We simply trace the sender of the message and message type onto the trace file. Each process will have its own trace file. To implement replay, at each receive operation for each process, we read from the trace file the next message and its message type that was received in the tracing execution. If current

Tracing

For each process, at receive event:
1. log the message source and type
2. Write the message source and type to trace file

Replay

For each process, at receive event:
1. Read the expected message source, s1, and message type, t1 from the trace file
2. Check the current incoming message's source, s2, and type, t2.
3. if s1 = s2 AND t1 = t2, then
   receive current incomingmessage
   else
   goto 1, and wait for next message.

Figure 1. Naive Tracing and Replay Algorithm

message matches the source and message type of the traced message, it will be accepted, otherwise, it will wait for next message. Thus replay execution will be deterministic.

3.2 Message Races

In Adaptive tracing and replay algorithm, we only trace messages that race. We first define what is a race.

Intuitively, two messages race if either can be accepted by one receive event, *Recv*. In that case, in one run, *msg a* may be received by *Recv* first, while in another run, *msg b* may be received by *Recv* first.

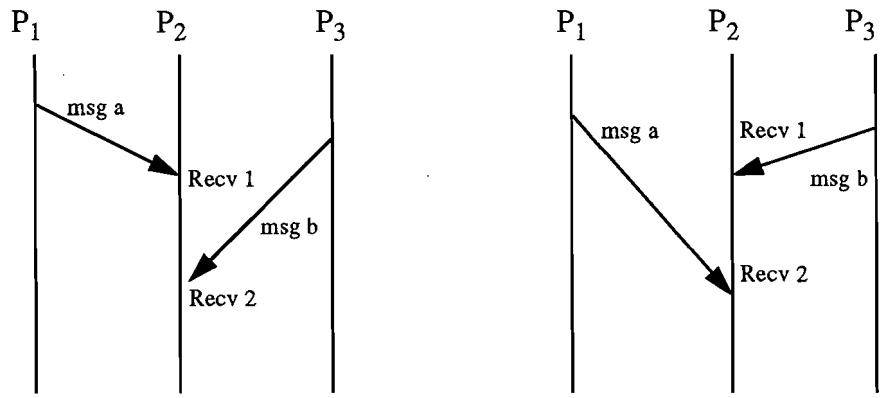


Figure 2. Two possible executions of a program

In Figure 2 [1], we see that in process 2, two messages can be received in different orders. While such nondeterminism may be intended, it causes problem in debugging since re-execution of the same program is not guaranteed to reproduce the original execution.

Netzer used the "Happened-Before" relation to define message race[1]. Formally, two messages race if one message does not "Happened-Before" the other message. The "Happened-Before" relation is the irreflexive transitive closure of the union of two other relations:($\xrightarrow{XO}$ U $\xrightarrow{M}$)$^+$. The $\xrightarrow{XO}$ relation is the order of events in the same process. The i^{th} event always executes before $i+1^{th}$ event in the same process because within a single process all events occur in a deterministic order. The $\xrightarrow{HB}$ relation shows the order that messages are sent. So $a \xrightarrow{M} b$ shows that *event a* sent a message that *event b* received. Thus, intuitively, $a \xrightarrow{HB} b$ is true iff 1) *a* and *b* are two events in the same process, and *a* executes before *b*, or 2) there are a sequence of messages that were sent from *a* to *b*[1].

Given this definition of race, we define two messages, *msg a* and *msg b*, race if

$a \xrightarrow{HB} b$.

3.3 Adaptive Tracing and Replay

The Adaptive Algorithm relies on the fact that not all messages race in a program. If we can identify those racing messages and only trace those messages, we can reduce the overhead of tracing. Furthermore, we only trace second message in a pair of racing messages[1], thus further reduce the overhead of tracing.

During tracing, we do a race check at each receive event to determine if the current message races with a previous received message. Using the $\xrightarrow{HB}$ relation described before, we can determine if there is a linear order between current message and previous receive. If there is a linear order, the delivery of the previous message must have occurred before the send event of current message, and no race exists. If the linear order can not be determined, there is a race, and we must trace the sender of the message, event serial number of the sender, and event serial number of the receiver. Since each process maintains a trace file, the receiver's event serial number is implicit and need not be traced.

To implement race check, we need to assign a serial number to each send and receive event. We use a local counter in each process for this task. The counter is incremented at each synchronization event.

During replay, we must ensure that the racing messages are delivered to the same recipient as well as in the same order as in the original execution. Since non-racing

1. For non-transitive races, we may trace more than 1 out of a pair of racing messages. See [1].

messages will be delivered to the same recipient, we need not do anything special for

Tracing

At each receive:

1. Send = event that sent current message, msg.
2. prevRecv = previous receive event in this process that are willing to receive this message.
3. Recv = current event receiving msg
4. if (prevRecv $\xrightarrow{HB}$ Send) Trace a msg is sent from Send to Recv

Replay

At each send:

1. Read the trace file and get the next traced message
2. If the current message matches the next traced message, tag the message

At each receive:

1. If the message is tagged, then
   receive only if it matches the local counter
   Else
   the message is not tagged, receive as norma

Figure 3. Adaptive Tracing and Replay Algorithm

them. At each *send* operation, we tag the message that is traced in the original execution with its intended receiver and the receiver's serial number. At each *receive* operation, we accept non-tagged message as normal. For the tagged message, we only accept if current process serial number matches that on the tagged message. The above trace and replay algorithm ensures that during re-execution every message is received by the correct event as that of the original execution.

4 Implementation

We implemented the Naive and Adaptive algorithms using PVM (Parallel Virtual Machine) on a network of Sun Sparc 10 Workstations running Solaris 2.4. PVM is chosen for its availability and extensibility.

4.1 PVM system

PVM is a message-passing programming system. It links separate machines to form a "Virtual Parallel Machine" and is designed to be portable. This "Virtual Machine" can be consisted of various types of machines that are physically apart and running variety of operating systems. Applications developed for PVM can be written in C, C++ or Fortran, and composed of any number of processes[6].

PVM is a set of software tools and libraries that emulates a general parallel computer on interconnected computers. A user configures a pool of host computers to be used in this "Virtual Machine". The computation model is process-based, and the unit of computation is a task. A user program first starts on a host machine. The program then spawns a number of processes, or tasks, to be scheduled by the host machine running on machines in the pool. Multiple tasks can exists within one machine.

Process-to-process communication is based on message-passing model. Each process is assigned a unique process id. Cooperation and synchronization are done by sending and receiving messages to one another, identified by task ids. Message size is not limited except by the availability of memory[6]. Each message contains the sender process id and receiver process id. It also contains its message type, thus a receiver must also check that the type is what it expects as well as that it is the right receiver. Message type can be a wildcard, in which case it can be received by a process expecting any type of messages. Receiver can also be a wildcard, and thus such message can be received by

any process.

PVM consists of two parts. The first part is a daemon that runs on each host machine included in the machine pool. Before running any PVM application, a user must first start PVM. This will set up a list of machines and the PVM daemon will be running on each host machine. Any application can then be started from any of these machines.

The second part of PVM is a library consisting of interface functions. A user program calls these functions for spawning processes, message passing, synchronizing processes, and modifying the virtual machines.

4.2 Modifications to PVM

Our algorithms are implemented by modifying the PVM libraries and daemon.

PVM daemon runs on each physical machine in the virtual machine. It spawns processes and assigns ids to the processes and schedule them on various machines. It also processes requests for communications between tasks within one machine and forward messages that are for processes on another machine. Task id is a unique integer that identifies the host machine it is running on and its sequence number among all tasks running under the same machine. Since PVM daemon generates different process ids within one session, process ids can be used to identify a task within the virtual machine.

Because we must identify the tasks in tracing and replay, we need to maintain that during replay, a message is sent from one process to another process, and these two processes are the same in both executions. Therefore, we must also trace the tasks spawned by the host machine so we can refer to the same process by its task id. We accomplish this by forcing PVM daemon issuing the same task id when spawning a new task at replay execution.

In PVM library, we changed several library functions to implement tracing and replay. The *recv* library function is modified so it will trace the delivery of messages. The race checking is implemented within the *recv* library function call. The spawn function is also modified so the processes spawned are traced. A number of other functions are changed to implement the tracing and replay algorithm.

Our algorithms are entirely implemented within PVM. The user does not need to know how it is implemented to use these algorithms. Two environment variables, TRACE_REPLAY and NAIVE_ADAPTIVE, are used to specify whether a user wants *trace* or *replay* (or none) and whether to use the *naive* or *adaptive* algorithm. The user program does not need to concern with the algorithm. Any PVM program can utilize our algorithm without any modification.

4.3 Implementation of the algorithm

4.3.1 Implementing the naive algorithm

Implementation of the naive algorithm is straightforward. For tracing, when *pvm_recv* function is called, we trace the sender task id of the message and its message type, and write them in the /tmp directory in the local disk of the local machine where the process is running. We maintain one trace file for each process.

During replay, at each *pvm_recv* call, we read from the trace file the sender task id and type of the message to be received. We compare the current message and only receive if it matches both the sender task id and message type. Otherwise, the message is buffered for future receive event.

4.3.2 Implementation of the adaptive algorithm

In the adaptive algorithm, we need the information on $\xrightarrow{HB}$ relations between two receive events. we use vector timestamps to provide such information. Each process has one vector timestamp associated with it as well as one local counter. The local counter is

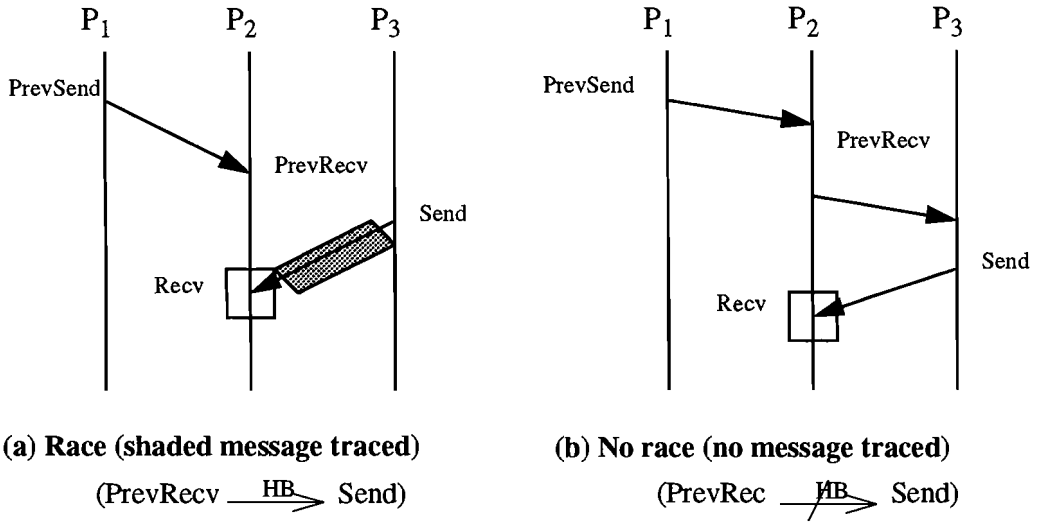
used to assign the serial number to each synchronization event and is incremented after each operation[1]. The vector timestamp is maintained by each process so that during the execution, its i^{th} component is the serial number of the last event in process i that happened before the most recent event in process p.

During the execution, each process appends its own timestamp onto the outgoing message. At the receiving end, each process updates its own timestamp by taking the maximum of each component with the timestamp from the incoming message.

The race check can then be simply performed using the timestamp. The sender's timestamp from the incoming message is compared to the serial number of the previous receive event that could have received this message to decide if the previous receive "happened before" the current event. The value of the p^{th} slot of the sender's timestamp equals to the serial number of the most recent event in process p that happened before the *send* event in sender. If the serial number of the previous receive is greater than this value, then the previous receive did not happen before the send event in sender, and these two events race. If a race exists, then current incoming message is traced. The sender's process id, its serial number and current process serial number is recorded to the trace file that is maintained for each process. Figure 4 illustrates the racing checking in the algorithm. In (a), PrevRecv did not "Happened Before" Send, therefore, message from *Send* to *Recv* is traced.

Since we only trace the second message in a pair of racing messages, we must do something special at send time in order to produce correct replay. To implement replay, we first sort the trace files by sender's task id. The files are collated according to the sender process. At each send event, a process will check its file to see if current message is traced. If it is, then it races with another message in the original execution and we will tag the message by the process id of its intended receiver and the receiver's serial number. Non-traced messages will be sent as usual. At each receive event, the received

will only accept a tagged message if its counter matches the counter in the message. Non-



(a) Race (shaded message traced)
(PrevRecv $\xrightarrow{HB}$ Send)

(b) No race (no message traced)
(PrevRec $\not\xrightarrow{HB}$ Send)

Figure 4. Example race check performed at boxed receive[1]

tagged messages are received as usual.

4.4 External environment

We assume that the original execution and the subsequent replays occur in equivalent virtual machine environment. This assumption should not affect general cases since to implement successful replay we only need to make sure that message deliveries in replay are the same as that of the original execution. As long as the behavior of the program execution does not depend on the value of the real time clock, our algorithms will successfully implement replay. Further, any cyclic debugging system for sequential programs will have to face this same problem of finding two equivalent virtual machine. We do not depend on a specific simulation of virtual machine, therefore any techniques developed for sequential program debugging can be used for distributed environment.

In our implementation, we assume that programs do not exploit the physical

characteristics of any resources allocated by the system. We only ensure that the amount of resources available during replay is at least the amount used by the program during the original execution.

5. Experiments and results

5.1 Experiment

We analyzed a collection of PVM programs obtained from colleagues. We measured their running time and trace sizes for the tracing executions on the non-instrumented PVM system and two trace and replay algorithms described above. We measured the number of all messages sent. This is collected by counting all the messages traced by the Naive Algorithm since it traces all messages delivered during an execution. We then measure the number of messages traced by the Adaptive Algorithm and the percentage is calculated and recorded. We measure the running time for the tracing executions with both algorithm, replay with the naive algorithm, and the running time with the original non-instrumented PVM system. The percentage of execution time increases by tracing and replaying over non-traced executions are recorded as overhead. We experimented on six programs:

FFT: computes the Fast Fourier Transformation of size 128.
CHOL: computes the Cholesky factorization of a random matrix of size 1000 by 1000.
BSORT: sorts a random array of floats of size 1,000,000.
N_PUZZLE: solves a puzzle of size 10x10 with maximum search depth set at 42.
HEAT: calculates heat diffusion through a substrate (a wire).
MMULT: computes the product of two randomly generated matrix,
each with size of number_processors x block size.

In all instances, we configured 9 machines to run each program. we ran each program 20 times and take the middle 10 runs as our reported results. Because these programs are

non-deterministic, and also because the erratic behavior of operating system and network, each run of a program will result in a very different time. This made comparing the performance of various runs difficult. However, by using a large number of executions of a single program, we hope that we will get average behavior.

The experiments are run on Sparc 10 running Solaris 2.4 connected by a local network. File system is SUN's NFS running on four servers. To see how the server behavior affects the computation, we also run the programs from the local disks along with all I/O.

5.2 Results

5.2.1 Performance

We first measured the running time of each algorithm. Table 1 shows the running time for each algorithm as well as replay. Time is in seconds and are for 10 runs of each algorithm.

Table 1 shows the running time of the six programs with no-trace, Naive, and Adaptive algorithm. Both Naive and Adaptive algorithm show increased running time over the non-instrumented version. The execution time overhead ranges from 0.2% to 17.7%. However, the Adaptive Algorithm shows somewhat better performance than the Naive Algorithm. In all but one program, the execution time overhead of the Adaptive Algorithm is lower than that of the Naive Algorithm. Nevertheless, execution time overhead is low even for programs that have large number of messages.

In one program, *n-puzzle*, the running time of the naive algorithm is slightly better than that of non-instrumented version. This is probably due to non-determinism of the program, unpredictable behavior of the operating system and network irregularities.

Running Time (Time are in seconds)

Program	No-trace (seconds)	Naive (seconds)	Adaptive (seconds)	Naive Replay (seconds)
N-Puzzle	304	300	311	297
FFT	878	951	936	943
CHOL	637	750	702	684
BSORT	237	276	253	258
HEAT	469	482	470	490
MMULT	201	210	205	208

Table 1. Running Time for Both Algorithms and Naive Replay

Time Overhead for Tracing and Replay (in percentage)

Program	Naive (%)	Adaptive (%)	Naive Replay (%)
N-PUZZLE	(1.3)	3.6	(2.6)
FFT	8.3	6.6	7.4
CHOL	17.7	10.2	7.3
BSORT	16.4	6.75	8.8
HEAT	2.8	0.2	4.0
MMULT	4.4	2.0	3.9

Table 2. Time Overhead for Tracing and Replay

Running Time and Standard Deviation

	Adaptive		Naive		No-trace	
	Time (sec)	std/avg (%)	Time (sec)	std/avg (%)	Time (sec)	std/avg (%)
N-PUZZLE	254.1	1.5%	266.5	1.7%	259.4	2.8%
FFT	92.9	3.4%	95.2	6.9%	91.3	6.3%
CHOL	70.3	4.3%	74.98	4.3%	74.8	4.2%
BSORT	31.7	27.9%	28.9	9.1%	29.2	13.6%

Table 3. Running Time and standard deviation

In all programs, the running time has a variation between 3%-15%. Table 3 shows a different run of some of the programs. In addition to the running time, we also included the ratio of the standard deviation of the running time to the average running time.

The variance for these programs are generally low. For long running programs like *n-puzzle*, the ratio of the standard deviation to average running time is only 2-3%[1].

We also compared the running time of four programs using Naive Algorithms to see if by running the programs on local disks we will gain more stable running time. This is achieved by copying the executables and data files to the local disk of each machine before running the programs, and redirect all I/O to the local disks. We first thought that by running the program form the local disk, we may reduce the influence of erratic behavior of the file server, which contributes to large variance in running time. Table 4 shows the result. The results are gathered for 10 runs of each program and are average for each program. Unlike our earlier results, we included all running times, as opposed to

1. The raw data of these running time are included in appendix.

taking the middle. This explains the increased *std/avg* ratio. The result indicates that the running time is very close for both local disk or file server. Further, the std/avg ratio is also too close to indicate any significance.

Local disk versus Server (Time in seconds)

	Avg Time (secs)	Max Time (secs)	Min Time (secs)	Std/Avg (%)
File Server				
N-PUZZLE	194	204.5	182.6	8.8%
FFT	86.7	96.5	80.3	15.4%
CHOL	71.7	81.3	61.3	23%
BSORT	32.9	40.34	25.8	36%
Local Disk				
N-PUZZLE	194.4	206.7	181.57	10.6%
FFT	85.5	96	80.5	15%
CHOL	69.5	80.3	57.3	26%
BSORT	32.96	37.3	26.3	29.5%

Table 4. Running time from Local Disk versus File Server

5.2.2 Tracing size

Table 5 shows the number of messages trace for each algorithm.

The number of messages traced are taken from 20 runs of each program. Some programs have large number of messages, e.g. *n-puzzle*, *chol*, and *heat*, while others have very few messages. As shown in table 5, the Adaptive Algorithm traced far fewer messages than the naive algorithm did in most programs. Percent of messages traced by the Adaptive.

Space Overhead

Program	Naive	Adaptive	Adaptive/Naive
N-PUZZLE	2,614,880	657,260	25.1%
FFT	13,200	11,100	84.1%
CHOL	423,000	230,020	54.9%
BSORT	1,520	860	56.6%
HEAT	240,750	218,830	89.9%
MMULT	10,250	1,520	14.8%

Table 5. Comparison of Number of messages traced

algorithm over the Naive algorithm ranges from 15% - 90%.

Originally, our algorithm will consider two messages from *process a* to *process b* will race. However, since PVM will preserve the message order in such a case, we changed the Adaptive Algorithm so that two messages will not race if they have the same sender, receiver and message type. This, however, does not seem to influence the number of messages traced by the Adaptive Algorithm in any significant way.

6. Discussion

6.1 Running time overhead

Both algorithms seem to do well. Performance did not degrade in any significant way.

As is expected, the runtime overhead of the Adaptive Algorithm is in general less than that of the Naive Algorithm. The Adaptive Algorithm produced better performance than

Naive tracing and replay. Its execution time overhead is generally very low, between 1-5%. This is well expected since much of the runtime overhead is related to writing traced information to the disk. Since the Adaptive Algorithm only traces the racing messages and therefore only a small fraction of the total messages are traced, the execution time overhead is lower.

However, the overall runtime overhead of both algorithms are rather low. The worst overhead figure is 17.7%, from CHOL, with the Naive Algorithm. Given that the program running time has 5-10% variation, this runtime overhead is well within the acceptable range.

This may come as a surprise. However, in our experiments, the longest running program has a running time in the hundreds of seconds and therefore not exactly a long running program. Further, the number of messages passed in any of our programs is less than 50,000. Most of our programs have only few hundred messages in each execution. Since the runtime overhead is mostly tracing the messages, these programs should not exhibit severe degradation in performance.

Although it is not evident from the data in the tables, one might expect that the runtime overhead will increase for longer-running programs and for those programs that pass significantly large number of messages.

In [1], Netzer reported that tracing all messages suffered severe performance degradation, even for programs with moderate number of messages. Slowdown of greater than 500% is recorded for those programs with large number of messages passed. Our programs do not seem to suffer from such performance degradation with the Naive Algorithm, even though some of our programs have more messages passed per program than those reported in [1]. In [1], the experiments are conducted on dedicated message-passing parallel machines. Our algorithms are implemented on top of the PVM system.

In PVM, messages passed between tasks running on different host machines incurs additional overhead because all communication must first go through the PVM daemon. In addition, there are frequent control messages and information messages between PVM daemons. These factors may hide some of the overhead in our implementation of the tracing algorithms.

We also discovered that running the same experiments under the same condition except with different pool of host machines may greatly affect the running time, regardless of the algorithms. Table 6 shows the running time for four programs with two different pools of host machines. Each data in the table is the average running time from 10 executions of each program.

Running Time Variation with Adaptive Algorithm

programs	Host Pool 1 (seconds)	Host Pool 2 (seconds)	difference (%)
N-PUZZLE	200.15	254.1	27%
FFT	87.6	92.9	6%
CHOL	74.4	70.1	(6.1%)
BSORT	31.6	31.7	--

Table 6. Running Time variation with different Host Machines

We do not know why the execution time fluctuates in such a great degree. In particular, the program *n-puzzle* exhibits 27% difference in average execution time with two different sets of host machines. The difference is nearly 10 times the standard deviation for each run.

Thus, we can only conclude that no definitive conclusion on relative performance of two algorithms implemented here can be drawn from our data. Although our results suggest

that Adaptive Algorithm performed better than Naive Algorithm, the variance in running time is too great to make any definitive assertion. In particular, in Table 4, we see that the Adaptive Algorithm performes better than the Naive Algorithm and the Uninstrumented Program. All programs are run under the same condition: same input and same host machines that are carefully selected to avoid contention with other user. We can only surmise that it is the result of unpredictable network traffic pattern and other external factors that are beyond our control.

6.2 Trace size

Although the Adaptive Algorithm traced a fraction of all messages, the number of messages traced by the Adaptive Algorithm as a percentage of total messages is rather high. Two programs, *n-puzzle* and *mmult*, traced only 14% and 26% of total messages respectively, others traced more than 50% of total messages. The worst figures are from *fft* and *heat* where both traced more than 80% of total messages.

Ideally, the Adaptive Algorithm traces only the racing messages, and further, it traces only one of the two racing messages. However, if a number of messages are involved in one race, the algorithm will trace all but the first message involved in the race. This seems to be the case in most of our programs. In *n-puzzle* for example, each child process performs computation on the current configuration of the puzzle and send results back to its parent at each search depth level. Since all messages passed to the parent are of the same type and could potentially raced, the Adaptive Algorithm must trace most of these messages. Although none of the messages between the child processes race and therefore not traced, the algorithm must still trace a large number messages because in *n-puzzle* these messages account for nearly half of all messages in a given execution.

Although our program did not achieve order of reduction in number of messages traced, we can still infer from the better performance of the Adaptive Algorithm in most

programs to conclude that the benefit of reduced tracing outweights the added cost for race checks and appending the timestamp to each message.

6.3 Overall

Because the external factors vary greatly during different executions of the algorithm -- the host machines we use, contention of machines with other users, network traffic and occasional but definite influence of the file server delays -- the results that we obtain could not precisely conclude the relative performance of the algorithms. However, based on the data, we can still conclude that neither algorithm suffer severe execution time degradation.

In our implementation of both algorithms, we did not optimize tracing. We used *fprintf/ fscanf* instead of direct *read/write* system calls. Further, we could buffer the traced data in 4K chunk thus reduce the number of read and write system calls further. These considerations should place our results as the upperbound of our tracing overhead, further indicates that the tracing overhead is low.

The Adaptive Algorithm suffers from additional overhead in preparing for replay. Before each replay execution, we need to sort and collate the trace files from all hosts machines. One way to simplify and reduce such work is to write all trace data in one central file instead of one file on local disk for each process. However, the performance of the tracing is poor. Netzer reported that writing trace data to one file rather than to local disk suffers 6-10 times of performance degradation.

7 Conclusion

We explored two trace and replay algorithms. Their performances are measured. The Adaptive Algorithm has lower runtime overhead and in general traces much fewer

messages than the Naive Algorithm. Overall, in our experiment, no program suffered severe performance degradation with either algorithm. This could be due to the fact that our example programs are small in terms of running time and number of messages passed.

One might very well expect that performance will suffer with longer-running programs. Since the Naive algorithms is simpler to implement replay, it may be a better choice for programs that do not pass many messages. However, for programs with many messages, the benefit of the Adaptive algorithm could outweight its disadvantage.

8. Acknowledgements

I would like to thank my adviser, Rot Nezter, for his guidance and help. I would also like to thank my wife, Sandra, for her understanding and support.

References

[1] Netzer, R and Miller, B, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," Supercomputing '92, pp. 502-511 Minneapolis, MN (November 1992).

[2] T. LeBlanc and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay,", IEEE Transaction on Computers C-36(4) pp.471-482 (April 1987).

[3] C. McDowell and D. Helmbold, "Debugging Concurrent Programs," ACM Computing Surveys 21, 4, pp.593-622, December 1989.

[4] Larry Wittie, "Debugging Distributed C Programs by Real Time Replay," SIGPLAN/ SIGOPS Workshop on Parallel and Distributed Debugging, pp.57-67, Madison, WI, May 1988.

[5] B. Miller and J. Choi, "A mechanism for Efficient Debugging of Parallel Programs," Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp.141-150, Madison, WI, May 1988. Appendix.

[6] A. Geist, A Beguelin, J. Dongarra, etc, "PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for networked Parallel Computing," The MIT Press, Cambridge, MA 1994

Appendix

A. How to use the tracing and replay algorithms

The algorithms are implemented by modifying the PVM daemon and PVM user interface libraries. Thus it is completely transparent to the user. User will write their programs as usual. Two environmental variable, TRACE_REPLAY and NAIVE_ADAPTIVE, can be used to set the mode for tracing, replay or none. TRACE_REPLAY can be set to "trace" or "replay" to set the system in tracing or replay mode. The default is none. NAIVE_ADAPTIVE can be set to "naive" or "adaptive" to choose the algorithm to be used. The default is Naive algorithm if either tracing or replay mode is selected.

Before you start the PVM system, you need to set the two environment variables to appropriate values to enable trace or replay. Nothing needs to be set if you do not need to do tracing or replay. Then you start the PVM system as usual. Because all traces files are written on the local machines, you must ensure that the program is run from the same host machine as in the tracing execution.

At present, to enable replay, you must restart the PVM system. This is because the system currently is not able to force the PVM daemon to regenerate the same task id number for the host process. However, once the host process starts, all subsequent processes will be assigned the same task id number as in the original execution. In the future, we could implement a mapping in the PVM daemon so the host process will be automatically mapped to the same process id, thus obviating the need to restart the system for each replay.

B. Where is the code?

The instrumented PVM system code is in /u/why/cs/pvm3. The programs that we experimented are in /u/why/cs/pvm_progs/. There are script files in each directory to

automatically execute the program once the appropriate environment variables are set.

C. What is modified from the original PVM system?

The modified PVM system is taken from PVM version 3.3.7. The files that are modified are:

PVM Daemon:

pvmd.c, tdpro.c, ddpro.c, task.c, global.h

PVM library:

lpvm.c, lpvmgen.c, lpvmpack, lpvmcat, timestamp.c, tevmac.h.

D. Data from the experiments[1]

D.1 Bsort, Adaptive Algorithm

43.291678
33.945922
38.545040
36.902532
29.972472
30.598765
33.303382
34.173307
26.634587
30.640031
28.157372
32.790708
34.739847
29.037777
27.756455
32.519174
33.600923
26.304627
29.832300
27.300312

total time: 316.241454. Max= 33.945922, min= 29.037777, avg=31.624145, std = 17.0%

1. All time is in seconds. Statistics are taken for the middle 10 executions.

D.2 chol, Adaptive Algorithm

58.028133
61.455418
85.054917
125.668490
71.261775
73.468440
76.702368
76.566791
73.707926
73.904314
69.956497
75.017055
73.358272
72.329975
72.109837
76.544592
79.427379
74.225289
105.929691
75.052517

total time: 744.175171. Max= 76.566791, min= 72.329975, avg=74.417517, std = 5.5%

D.3 fft, Adaptive Algorithm

99.113937
89.333321
85.532867
81.143198
93.407359
91.818161
84.978087
91.104765
85.265786
98.728473
86.432629
84.822053
79.975876
89.295378
90.502762
85.458936
86.467977
85.053684
98.067513
86.629449

total time: 876.023870. Max= 91%00000

D.4 n-puzzle, Adaptive Algorithm

232.872229
264.576185
272.846884
252.271892
256.729313
253.036242
256.465180
253.520723
254.408305
236.876953
250.926923
253.501780
258.705932
256.741392
254.853728
253.722332
254.648238
250.696765
259.511261
254.298406

total time: 2840.726826. Max= 256.465180, min= 252.271892, avg=254.072683, std = 1.5%

D.5 bsort, Naive algorithm

34.769579
23.234466
27.587271
38.090655
30.073580
29.516496
28.275545
32.706608
27.648343
29.667202
21.720104
27.330179
35.760064
28.941129
29.214589
27.426876
32.614825
27.971823
29.088019
20.962864

total time: 287.983997. Max= 30.073580, min= 27.587271, avg=28.798400, std = 9.1%

D.6 chol, Naive algorithm

77.155328
74.681972
74.156608
81.114729
88.184450
67.908086
77.161733
75.655169
75.941668
73.527625
74.214644
72.349508
78.296680
78.055926
69.449826
73.630100
75.415294
72.959497
73.907807
74.997908

total time: 749.756498. Max= 77.155328, min= 73.630100, avg=74.975650, std = 4.3%

D.7 fft, Naive algorithm

97.644032
97.479117
91.525584
87.998633
100.925594
93.294542
95.285635
84.435079
95.575461
99.911506
96.699087
91.420361
87.322639
97.789587
92.045126
96.957350
83.897580
95.318909
98.708737
99.429699

total time: 951.824843. Max= 97.644032, min= 91.525584, avg=95.182484, std = 6.9%

D.8 n-puzzle, Naive algorithm

257.513900
263.066369
266.729403
268.854797
269.099308
261.409555
265.857368
267.152830
266.644244
251.115282
263.531272
300.918096
267.646350
270.804361
268.554261
264.185995
266.904399
267.508314
257.082206
269.297473

total time: 2664.714436. Max= 268.554261, min= 263.531272, avg=266.471444, std = 1.7%

D.9 bsort, non-instrumented

25.126405
29.150500
28.998890
31.924465
31.802149
28.455499
32.276153
32.038201
26.670987
27.457082
27.290051
30.404276
30.299197
28.660107
28.524219
27.771932
31.992130
31.865930
27.277864
20.519185

total time: 291.523851. Max= 31.802149, min= 27.457082, avg=29.152385, std = 13.6%

D.10 chol, non-instrumented

81.196658
73.970186
76.572637
74.267838
73.600746
66.339077
75.294591
93.355159
79.792708
72.859203
73.384786
71.344483
75.419043
77.983726
68.009478
76.139829
73.878399
73.745944
74.717996
85.621290

total time: 747.607209. Max= 76.572637, min= 73.600746, avg=74.760721, std = 4.2%

D.11 fft, non-instrumented

100.451874
95.408041
95.897182
84.007822
97.060580
97.587604
90.380820
90.819315
85.631827
86.191289
98.571030
98.196410
91.342923
91.834627
94.012648
94.456435
94.558513
95.095089
98.666972
97.946485

Total time: 947.253642. Max= 97.587604, min= 91.342923, avg=94.725364, std = 6.3%

D.12 n-puzzle, non-instrumented

231.815824
258.078264
263.779654
266.359780
254.562539
254.930387
265.591321
258.339162
255.864773
240.607371
261.038288
256.213996
254.956663
260.435233
269.471254
260.870761
258.208765
261.270861
270.810242
288.326736

total time: 2594.099757. Max= 263.779654, min= 255.864773, avg=259.409976, std = 2.8%