An Implementation of Race Detection and Deterministic Replay with MPI

C. Clémençon[1], J. Fritscher[1], M.J. Meehan[2] and R. Rühl[1]

[1] CSCS–ETH, Centro Svizzero di Calcolo Scientifico,
6928 Manno, Switzerland
[2] Dept. of Computer Science, Univ. of North Carolina,
Chapel Hill, NC 27599-3175, U.S.A

Abstract. The *Parallel Debugging Tool* (PDT) of the *Annai* programming environment is developed within the *Joint CSCS-ETH/NEC Collaboration in Parallel Processing* [1]. Like the other components of the integrated environment, PDT aims to provide support for application developers to debug portable large-scale data-parallel programs based on HPF and message-passing programs based on the MPI standard. PDT supports MPI event tracing for race detection and deterministic replay for manually parallelized MPI programs as well as for code generated with the advanced techniques of a data-parallel compiler. This paper describes the tracing and replaying mechanisms included in PDT as well as their efficiency by presenting execution time overheads for several benchmark programs running on the NEC Cenju-2/3 distributed-memory parallel computers.

1 Introduction

The lowest-level programming paradigm employed often on Distributed-Memory Parallel Processors (DMPPs) is pure message passing. Until recently, proprietary message-passing libraries have been used a majority of the time, which has made porting of applications from a given DMPP to a system of another vendor difficult. In addition, message passing has been recognized to be tedious and error prone. To provide a higher-level programming interface, several data-parallel languages have been proposed. To allow for portability, the low-level Message-Passing Interface (MPI) and the high-level High Performance Fortran (HPF) have been defined and both have been accepted as standard by most DMPP vendors. HPF allows for the integration of MPI primitives into high-level source code through extrinsic procedures.

Many problems still remain regarding the usability of DMPPs. Debugging of DMPP programs for correctness is difficult because of possible deadlocks and non-determinism. Non-determinism may be either unintended or intended. Unintended non-determinism is a programming error and should be detectable by a DMPP debugger. The intended non-determinism used in some programming models (such as the *master/worker* model described in Section 2) can cause problems for the debugger because it might not be possible to re-produce an error condition if it is dependent on the execution order of a non-deterministic section of a program.

We are currently developing a fully interactive, source-level debugger for large-scale DMPP programs. The debugger is called Parallel Debugging Tool (PDT) [2], and

is integrated into the programming environment *Annai* [3]. Annai also includes a Parallelization Support Tool (PST) [4], a Performance Monitor and Analyzer (PMA) [5] and an OSF/Motif-based graphical User Interface (UI). PMA supports performance debugging, while PDT supports debugging for correctness. Among the major design objectives for all tools are application-oriented design, portability, scalability, and support of both MPI at the low level and extended HPF at the high level. A group of application developers is continuously evaluating tool prototypes and providing feedback for functionality enhancements.

This paper reports results about the race detection and deterministic replay facilities built into PDT. Although first DMPP debuggers have been built which include deterministic replay features, they are either research systems targeting at a single communication interface and lacking the portability of MPI, or they emphasize other features than replay efficiency and scalable interactive debugging. In contrast to these systems (see also Section 3.1), PDT provides portable scalable interactive debugging support for both low and high-level programming paradigms. This includes a complete MPI instrumentation for the detection of non-determinism and for deterministic replay.

The next section gives an example of the two main types of non-determinism which can occur in MPI programs. Then we summarize related work and describe the integration of the various Annai components in more detail. After a summary of the algorithms used for race detection and deterministic replay, we present measurements of tracing and replay overhead on NEC Cenju-2/3 DMPPs.

2 An MPI Example

In Fig. 1 we show a simple code segment with calls to MPI which cause races. These races are due to blocking receives which use the `MPI_ANY_SOURCE` wild-card for matching the source processors of incoming concurrent messages. Although the code segment has no application and was constructed solely for benchmarking purposes (see also Section 5.2), it illustrates one kind of race possible in MPI programs: $p-1$ processors send messages "concurrently" to the same processor which accepts messages using a wild-card for increased parallelism and efficiency. In the benchmark this is repeated $n \times (p-1)$ times (loops at lines 5 and 6) and each processor plays the role of a receiver n times.

The situation is again depicted in Fig. 2 in a space-time diagram for four processors. Throughout the paper we denote a message as MSG(a, b), where a defines the id of the source processor of a message, and b a serial event counter on that source. Analogously, the respective receive events are denoted as RECV(a, b), where the serial number b is computed on the receiver.

Fig. 2 shows a triple race between MSG(0, 1), MSG(2, 1) and MSG(3, 1). If replay information about MSG(0, 1) and MSG(3, 1) (the first two messages to be received) is stored in a first execution of the code segment and during re-execution the order of these two messages is guaranteed, then MSG(2, 1) goes to RECV(1, 3) as desired and the first execution is deterministically replayed.

A second type of non-determinism can occur in MPI due to the use of the non-blocking probe operation `MPI_Iprobe` (further on called *iprobe*). With iprobes,

```
Void RECEIVEBENCHMARK(Int n, MPI_Comm c)
 1:   MPI_Status s;
 2:   Int i, j, k, x, p, myid;
 3:   MPI_COMM_SIZE(c, &p);
 4:   MPI_COMM_RANK(c, &myid);
 5:   for i := 1 to n
 6:     for j := 0 to p − 1
 7:       if j = myid
 8:         for k := 1 to p − 1
 9:           MPI_RECV(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG, c, &s);
10:         end for
11:       else
12:         MPI_SEND(&myid, 1, MPI_INT, j, TAG, c);
13:       end if
14:     end for
15:   end for
```

Fig. 1. RECEIVEBENCHMARK is a typical example MPI code segment with races due to receives with wild-card source specification.

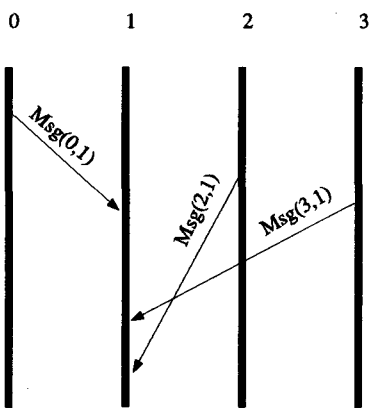

Fig. 2. Simple three message race.

non-determinism can occur on as few as two processors, because when replaying, depending on communication delays, the same sequence of iprobes may result in a non-deterministic sequence of booleans indicating the success of the operations. The example in Fig. 3 shows how a user can emulate a blocking receive using iprobes. For repeated executions of a program using EMULATEDRECV, with the same input, the loop at line 3 of Fig. 3 may be executed a different number of times. Note that in our implementation of MPI, `MPI_Recv` is actually based on non-blocking probes similar to what is shown in the figure.

```
Void EMULATEDRECV(Void *a, Int n, Int src, Int tag, MPI_Comm c, MPI_Status *s)
1:   Int true_src;
2:   Boolean f;
3:   repeat
4:     MPI_IPROBE(src, tag, c, &f, s);
5:   until f;
6:   MPI_GET_SOURCE(s, &true_src);
7:   MPI_RECV(a, n, MPI_INT, true_src, tag, c, s);
```

Fig. 3. EMULATEDRECV emulates a blocking receive using non-blocking probe operations.

3 Background

3.1 Related Work

The first step towards correct tracing and replay of communication events of a DMPP program, is the definition of the order of the events. Lamport [6] has first defined the *happens before* (or *causality*) relation "$\rightarrow$", an irreflexive, partial order on a set of events E. For two events $e_1, e_2 \in E$, $e_1 \rightarrow e_2$ is the smallest transitive relation satisfying the following conditions: either e_1, e_2 are happening on the same processor and e_1 precedes e_2, or e_1 is a send event and e_2 is the respective receive event on the destination of e_1. If two events do not causally affect each other, they are called *concurrent*. A time stamp added to each message can be used to compute on each processor for the events e_1, e_2 logical clocks $C(e_1), C(e_2)$ which satisfy the *clock condition*:

$$e_1 \rightarrow e_2 \;\Rightarrow\; C(e_1) < C(e_2)$$

Fidge [7] has first used *vector clocks* to define the exact causal order of events in DMPPs. On a DMPP with p processors, vector clocks have p elements, and their computation in general requires appending of a vector time stamp of length p to each message. For the many DMPP programs which feature communication locality, however, such overhead can be drastically reduced by only communicating incremental vector clock changes [8] which typically adds constant overhead independent of the number of processors.

Netzer and Miller [9] have first described how to use vector clocks to trace and replay *frontier races*. A *frontier* is drawn across a space-time diagram of a DMPP program, which divides communication send and receive events into two sets such that (1) two (or more) sends are just after the frontier, (2) a receive that could have received either of the respective messages is just after the frontier, and (3) all receive events before the frontier also have their senders before the frontier. For simple message patterns, frontier races are traced and replayed with minimum overhead. An evaluation of six programs

on a 32 node iPSC/2 is carried out which shows that only 1-2% of all message receives need tracing, and that tracing execution time overhead is less than 14%.

Damodaran-Kamal and Francioni [10] present the mdb debugger which allows the user to detect races using *controlled execution*: at certain points, the running program is suspended and the user is allowed to permute messages in send or receive queues. A set of commands is presented which can be applied to a suspended program; among others a sequential debugger can be used to inspect processor states. mdb does not support fully interactive low-level debugging in a scalable fashion. However, it supports the debugging of portable PVM programs.

Leu and Schiper [11] have built a system which integrates replay of non-determinism with ParaGraph-like visualization of message-passing programs. On the one hand, because the same trace format is used for visualization, race debugging and deterministic replay, both global and local behavior can be observed simultaneously. On the other hand, complete event tracing is necessary which introduces large trace overhead.

May and Berman [12] describe Panorama, a portable extensible system for both performance and correctness debugging of DMPP programs. A graphical interface runs on a user workstation and drives a vendor base debugger on the actual parallel platform. Deterministic replay is provided using complete message tracing. Measurements are shown on both an iPSC/860 and on an nCUBE, and for small messages, tracing accounts for an execution-time overhead of 7-55% on the iPSC and of 40-90% on the nCUBE, respectively, independent of the machine size.

3.2 Annai and its Parallel Debugging Tool (PDT)

Annai accepts high-level extended HPF programs and low-level message-passing source code. PST acts as a compiler for both paradigms. We are separately developing two other tools, one for correctness debugging (PDT) and one for performance analysis and tuning (PMA). For the reduction of trace overhead, each tool uses independent trace information, i.e. our basic communication platform MPI includes two orthogonal instrumentations.

The overall debugging support of Annai is split into two components, the Tool Services Agent (TSA) and PDT. TSA constitutes the machine interface of Annai, and provides a collection of basic debugging functions, for loading, executing, halting and inspecting a parallel program. TSA can be viewed as the back-end, low-level debugger of Annai and parts of it run on the target platform. PDT runs on the user workstation and its role is to provide more elaborate, higher-level debugging functions built from sequences of TSA commands. The high level debugging functions supported by PDT include global displays of distributed data structures and various breakpointing mechanisms. PMA uses TSA for interactive instrumentation of the target program. TSA is currently based on the GNU gdb debugger from the Free Software Foundation and plays a role similar to the basic vendor debugger underlying Panorama: while at the application level, portability is supported through the use of MPI and HPF, at the system level, TSA eases porting of the whole tool environment. Currently, we support the NEC Cenju-2, Cenju-3, and a DMPP emulator running on a Solaris workstation.

4 MPI Race Detection and Deterministic Replay

MPI provides three different kinds of tags as a basis for receiving a message: the source, the communicator (thus defining a context or group), and a conventional tag. Except for the communicator, there are wild-cards available for these tags. The communicator allows grouping of processes. Intra-group and inter-group communication is only possible through different communicators.

In a correct implementation of MPI, the order of messages with matching tags, sources and communicators is preserved (FIFO). Messages with different tags are allowed to overtake, however, even if they come from the same sender.

As already pointed out in Section 2 two classes of races can occur in MPI programs: races due to blocking receives and probes with equal tags, communicators and message-source wild-cards (for instance with `MPI_Recv`, `MPI_Probe` and `MPI_Wait`), and races due to asynchronous probe operations which non-deterministically test for message arrival in the system's message buffer (for instance with `MPI_Iprobe` and `MPI_Test`).

4.1 Blocking Probes and Receives

Race Detection and Tracing We use the same technique as Netzer and Miller to determine causality between messages and to trace racing receive primitives. Each processor maintains a vector time stamp and appends it to each message sent. On each processor, an internal scalar logical clock is incremented by one upon each send event. On processor i, the i-th element of the vector time stamp is equivalent to the processor's scalar clock. The remainder of the vector is determined by doing a component-wise maximum on the current time stamp and any time stamp received at the end of an incoming message.

Given two incoming messages a and b, the first arriving from processor p_a the second from p_b and their vector time stamps V_a and V_b, one can determine if they race by comparing the p_a-th value of the vector time stamps. If the p_a-th value of V_a is larger than the p_a-th value of V_b, then the two messages are concurrent and race. For message a to happen before b, the p_a-th value (p_a's internal clock) would have to be incorporated into message b's time stamp, because the p_a-th value would be passed along through the chain of messages linking the two.

Fig. 4 (a) demonstrates this. MSG(2, 1) ($p_a = 2$) arrives before MSG(0, 1) ($p_b = 0$) so the third (p_a-th) value of their time stamps must be compared. The third value of MSG(2, 1)'s time stamp is 1 and the third value of MSG(0, 1)'s time stamp is 0, therefore the two messages race. Considering MSG(2, 2) and MSG(0, 1), we see that the first value of the MSG(0, 1)'s time stamp is 1 and the first value of MSG(2, 2)'s time stamp is also 1, so the first message *happens before* the second message.

A *block race* is a collection of special frontier races which denotes races between a message and a set of messages. If the same message is concurrent to more than one message from the same processor, then it races with all of them. Fig. 4 (b) shows a block race. MSG(2, 1) is concurrent to both MSG(0, 1) or MSG(0, 2), therefore it races with both of them. Since MSG(2, 1) races with these messages, trace information about RECV(1, 1) and RECV(1, 2) must be stored. There also exists a block race between MSG(0, 3) and the pair MSG(2, 2) and MSG(2, 3) which results in storing information about RECV(1, 3) and

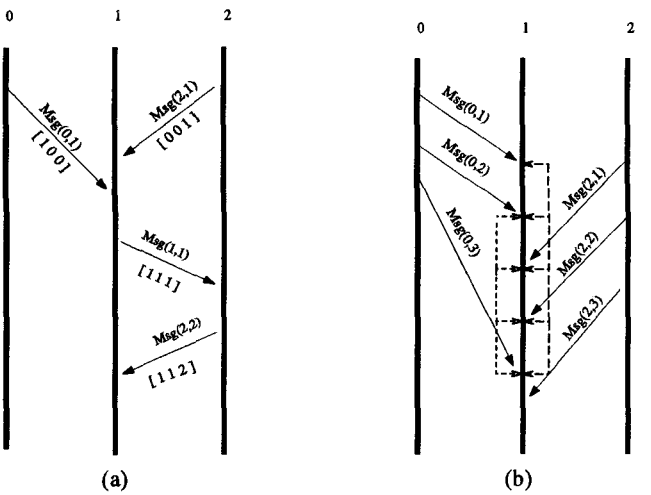

Fig. 4. On the left we show how vector time stamps are used to identify message races. On the right a block race is depicted.

RECV(1, 4). Due to the FIFO nature of the MPI communication channels, it is possible to store block races as single trace file entries. In our MPI implementation, block races can occur when long messages are chopped into pieces to fit into communication buffers of limited size.

To detect block races, a buffer of time stamps stemming from recently received messages must be stored for each tag. The buffer does not have to store all past time stamps. A time stamp can be taken out if a message of its tag has been received from every processor (excluding the time stamp's source and receiving processor). 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 in the trace file and the old time stamp is removed from the buffer. If the new message does not race with the old, then the new one's time stamp is added to the buffer and the old one remains.

In MPI, blocking and non-blocking probes can affect a program's execution when polling for racing messages. If any of the information from the probe is used in the program (for instance, existence of the message, the message's source or length), then the race affects the program.

Blocking probes can be traced in the same manner as receives. To trace them correctly, all of the operations for tracing a receive must be performed during the probe except for the removal of the message from the system buffer. The incoming time stamps 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.

When a blocking probe is traced, the same actions performed for an equivalent

receive are performed, including incrementing the same internal clock for each occurrence. If on a processor p there have been two blocking receives and three blocking probes, then the internal clock is incremented by five.

Deterministic Replay To replay a race condition, 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, one needs to read in the trace file which will indicate all critical receives and the correct messages for each.

Because FIFO message passing is assumed, the order of arrival of messages from a given processor is guaranteed. Therefore, the only piece of information necessary to determine the correct message for a critical receive is the sending processor's number. Assuming that all messages up to 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.

For a processor to replay deterministically, 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. Since the internal clock is incremented in the same manner as in the original execution, the algorithm can compare its current internal clock and the numbers of the critical receives to determine which receives must be controlled.

To implement re-execution, the program reads in the trace file and determines what the next critical receive is. The program executes without any interference until the first critical receive (since all of the preceding receives are already deterministic). Upon the first critical receive, the program blocks 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. Netzer and Miller have proposed a handshake protocol for the correct replay of message arrivals. However, the MPI user is only affected by different message arrivals if message buffer overflows occur. In our MPI implementation, buffers are fairly large, and we refrained from implementing the above handshakes because of efficiency considerations.

4.2 Non-blocking Probes

Non-blocking probes (*iprobes* as defined above) can also cause non-deterministic behavior. As with blocking probes, if any of the information from an iprobe is used in a program (for instance existence, source or message length), it can cause non-determinism. Iprobes cannot be traced in the same manner as blocking probes and receives, since they are simply checks for existence of messages. Instead, the outcome of every non-blocking probe must be recreatable from a trace file.

Keeping the trace information for every iprobe would be expensive. One would have to store each iprobe's outcome (true or false) and the probed message's source if the iprobe was successful and was of a class that can detect messages from more than

one source. For instance, in the Traveling Salesman Problem considered in Section 5.3, less than one percent of the iprobes executed return successfully. In such an application, tracing only successful iprobes can significantly reduce tracing overhead.

For deterministic replay of the iprobes, the traces are read into a buffer. Every time an iprobe is called, a respective counter is incremented. This iprobe event counter and trace files are separate and distinct from those used for the blocking receives and probes. Regardless of the existence of messages in the buffer, the replay mechanism forces the iprobe to return value FALSE until the iprobe counter matches the value for the next true iprobe. When the counter does match, value TRUE must be returned. To re-execute properly, the iprobe blocks until it has received the correct message before it returns a true value. Once again, only the source number of the message needs to be known for the critical iprobes.

5 Performance Measurements

5.1 The NEC Cenju-2 and Cenju-3

The NEC Cenju-3 installed at CSCS is configured with 128 processing nodes, each comprising a 75 MHz VR4400SC RISC processor, 32 Kbytes primary on-chip cache, 1 Mbyte of secondary cache and 64 Mbytes main dynamic memory. Processors communicate via a packet-switched multi-stage interconnection network composed of 4×4 crossbar switches. At CSCS, a 16 processor Cenju-2 (a predecessor of the Cenju-3) is also installed. The Cenju-2 features a similar network but is based on the slower 25 MHz MIPS R3000 processor. Some of the measurements shown in this paper were collected on the smaller Cenju-2, because it features a less intrusive timing function. The back-end C compiler used on both systems is the GNU `gcc` compiler, version 2.5.7.

5.2 Single Communication Primitives

To measure the impact of tracing and deterministic replay on the execution time of single MPI primitives we run the code segment of Fig. 1 ("Receive Benchmark") with $n = 500$ on our Cenju-2. The system timer on that machine features low intrusiveness and therefore average execution times of single MPI primitives can be measured reliably. To measure tracing and replaying overhead for iprobes, the benchmark was modified by replacing the `MPI_Recv` (Fig. 1, line 9) with the emulated receive of Fig. 3 ("Iprobe Benchmark"). Fig. 5 shows results of our performance measurements by breaking down the execution time spent in the three MPI primitives. Only the access time to trace buffers in memory is included in the measurements, but not the I/O necessary for writing and reading these buffers to and from disk.

The measurements show that with no instrumentation approximately half the time for a receive is spent waiting for the message arrival inside a loop of non-blocking probes. The other half is spent in receiving the message, which consists of bookkeeping and copying from system to user space. When tracing races, the overhead introduced in `MPI_Recv` is more expensive than for iprobes, because of the respective complexities of either tracing algorithm. When racing messages are traced in the receive benchmark,

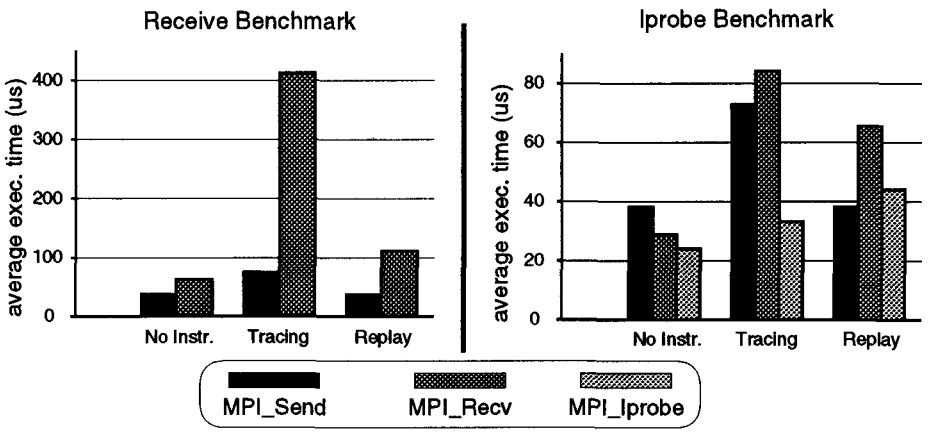

Fig. 5. Race detection and deterministic replay of single communication primitives on a 16 processor Cenju-2 in the benchmark of Fig. 1.

sends and receives are slowed down by approximately a factor of 2 and 6, respectively. Racing receives are replayed with less than a factor of 2 overhead with no noticeable performance difference for the respective sends. The relative execution-time overhead during replay is similar for both benchmarks.

5.3 Applications

We benchmarked two example programs:

TSP solves a traveling salesman problem on 18 randomly distributed cities using a parallel branch-and-bound algorithm. TSP is implemented in C with explicit MPI calls using a master-worker execution model (see also Section 2): the master processor constructs a queue of tasks which then are dynamically distributed to worker processes. The most important communication routine used for master-worker communications is `MPI_Iprobe`.

The **BiCGSTAB** solver from SPARSKIT [13] is applied to a simple sparse matrix, namely a banded random matrix of 16384 rows and a total bandwidth of 201. Our results refer to the performance of the iterative solver in steady state. BiCGSTAB is implemented in HPF with PST extensions, and compiled by PST, which intentionally introduces races for increased efficiency.

Fig. 6 shows the results of our measurements regarding the overhead of race detection and replay for TSP and BiCGSTAB, respectively. Both benchmarks were run on different Cenju-3 configurations without instrumentation, with race detection and with deterministic replay instrumentation. For TSP we show parallel execution times compared to an equivalent sequential program run; for BiCGSTAB MFLOPs are given. In TSP, the main source of non-determinism is the use of MPI non-blocking probe primitives. As the results show, replay is significantly slower than tracing: in the worst case performance is reduced by a factor of two compared to the non-instrumented code. In BiCGSTAB, few races are introduced by PST in blocking receive primitives. Our

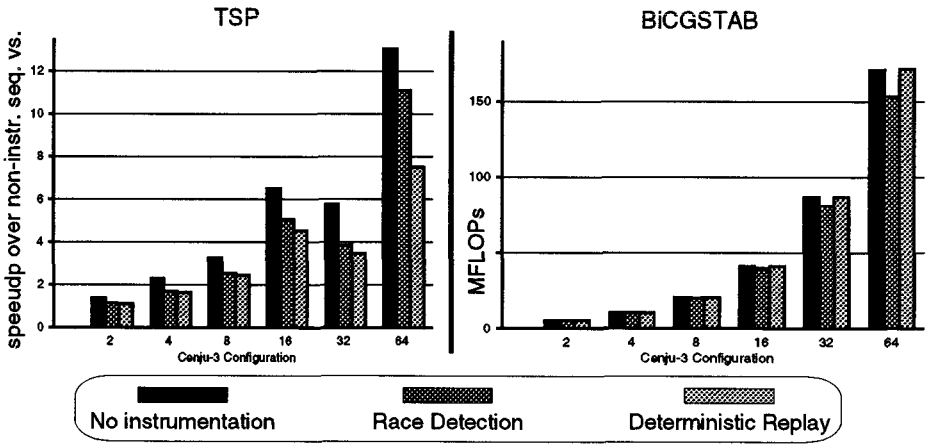

Fig. 6. The performance of TSP (left) and BiCGSTAB (right) without instrumentation, with race detection and with deterministic replay instrumentation is measured on several Cenju-3 configurations.

measurements show that only the performance of race detection is affected for large machine configurations, because the instrumentation requires longer messages to be communicated. The performance difference between non-instrumented and deterministically replayed program runs is negligible.

6 Conclusions

We have summarized the functionality of the Annai tool environment which integrates a compiler for extended HPF, a performance monitor and a parallel debugger. Since the whole tool environment aims at portable support of both extended HPF and MPI, one of the main features of the parallel debugger is race detection and deterministic replay. We have implemented such replay functionality in MPI with support for both frontier races of message receives with source-id wild-cards and for non-blocking probes (which are another type of race and require separate treatment).

We have measured the overhead introduced by both tracing and replay in a worst-case artificial benchmark where all messages race, as well as in two application programs, one parallelized using a master/worker model and non-blocking probes, and another data-parallel program where races are introduced for increased efficiency by the high-level compiler. We believe that the measurements show that the overhead introduced is acceptable, even for the debugging of large scale programs on many processors.

While the individual components of the debugger are not new to the research community, we believe we are the first to have integrated efficient race detection and deterministic replay into MPI and a tool environment which is portable and supports application portability.

Acknowledgments

The development of Annai has been a joint effort with A. Endo, A. Müller, and B. Wylie. Other project members, V. Deshpande, N. Masuda, W. Sawyer, and F. Zimmermann have generously been patient while evaluating our prototypes. We are grateful for many useful comments and careful proofreading of K. M. Decker and the EURO-PAR'95 reviewers.

References

1. C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, B. J. N. Wylie, and F. Zimmermann. Application-Driven Development of an Integrated Tool Environment for Distributed Memory Parallel Processors. In R. Rao and C. P. Ravikumar, editors, *Proceedings of the First International Workshop on Parallel Processing (Bangalore, India, December 27–30)*, 1994.
2. C. Clémençon, J. Fritscher, and R. Rühl. Execution control, visualization and replay of massively parallel programs within Annai's debugging tool. In *Proc. High Performance Computing Symposium, HPCS'95, Montréal, CA*, July 1995.
3. C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proc. of the 28th Hawaii International Conference on System Sciences, Volume II (Maui, Hawaii, USA, 3–6 January, 1995)*, pages 242–251. IEEE Computer Society Press, January 1995.
4. A. Müller and R. Rühl. Extending HPF for the Support of Unstructured Computations. In *Proc. ACM International Conference on Supercomputing, ICS'95, Barcelona, Spain*, July 1995.
5. B. J. N. Wylie and A. Endo. Design and realization of the Annai integrated parallel programming environment performance monitor and analyser. Technical Report CSCS-TR-94-07, CSCS, CH-6928 Manno, Switzerland, November 1994.
6. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
7. C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):183–194, January 1989. Published in ACM SIGPLAN Notices.
8. M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(10):47–52, August 1992.
9. R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.
10. S. K. Damodaran-Kamal and J. M. Francioni. *mdb*: A semantic race detection tool for PVM. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 702–709, May 1994.
11. E. Leu and A. Schiper. Execution replay: A mechanism for integrating a visualization tool with a symbolic debugger. In *Proceedings of CONPAR '92*, pages 55–66, September 1992.
12. J. May and F. Berman. Panorama: A portable, extensible parallel debugger. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, San Diego, California, May 1993.
13. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computation. CSRD Technical Report 1029, University of Illinois, IL, August 1990.