*MPL**: Efficient Record/Replay of nondeterministic features of message passing libraries

J. Chassin de Kergommeaux[1], M. Ronsse[2], and K. De Bosschere[2]

[1] ID-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, France
[2] ELIS, Universiteit Gent, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Abstract. A major source of problems when debugging message passing programs is the nondeterministic behavior of the promiscuous receive and nonblocking test operations. This prohibits the use of cyclic debugging techniques because the intrusion caused by a debugger is often large enough to change the order in which processes interact. This paper describes the solutions we propose to efficiently record and replay the nondeterministic features of message passing libraries (MPL) like MPI or PVM. It turns out that for promiscuous receive operations it is sufficient to keep track of the sender of the message, and for nonblocking test-operations to keep track of the number of failed tests. The proposed solutions have been implemented for an existing MPI-library, and performance measurements reveal that the time overhead of both record and replay executions is very low with respect to the (nondeterministic) original execution while the size of the log files remains very small.

1 Introduction

A program is called nondeterministic if two subsequent runs with identical user input cannot be guaranteed to have the same behavior. A program can be nondeterministic, even in the case when two runs produce exactly the same output.

Although nondeterminism is often a desired feature in dynamically evolving systems (e.g., for load balancing, scheduling, etc.), it makes debugging very cumbersome, especially in parallel and distributed programs. Indeed, the fact that one process is slowed down by a debugger will most likely change the order in which processes interact, and can eventually change the complete execution path. Hereby, symptoms of programming errors might suddenly appear, disappear or change.

Hence, unless we succeed in making nondeterministic program executions deterministic, cyclic debugging techniques cannot be used with this class of programs. A well-known technique to make a nondeterministic program deterministic, is to observe an execution (e.g., by recording the outcome of all nondeterministic choices made by the program), and to impose this trace on subsequent executions (with the same input). The replayed execution is now completely deterministic, and can be used any number of times to debug the program.

Debugging an erroneous program then amounts to recording an erroneous execution and to applying cyclic debugging techniques during subsequent replayed (erroneous) executions. In order for this technique to be effective (i) the perturbation resulting from the recording operation ought to be kept sufficiently low so that errors appearing during non-recorded executions do not vanish in recorded ones and vice-versa, and (ii) the overhead during replay should be acceptable.

The record/replay technique can be implemented in two different ways: either by tracing the message data, and by imposing these data to the receive and test operations during replay (*contents driven replay*), or by tracing the order of racing events, and by imposing that order during replay (*control driven replay*). Instant Replay [10] is an example of control driven replay for shared memory programs. It is obvious that contents based replay requires trace files that are at least an order of magnitude bigger than required for control based replay. Therefore, control driven replay is a better candidate for an efficient implementation of the record phase.

On the other hand, control driven replay only works in the assumption that the replayed executions depart from the same initial state and that the next state in each process p_i can be derived from the previous state, and a possible input (either user input, or an incoming message). Furthermore, all outgoing messages must be derivable from the program state that produces it. In other words, the program must comply with the rules of causality, and given identical user input, the contents of the messages must be guaranteed to be identical if the same control is imposed on the execution. For message passing systems compliant with these assumptions, a trace file with just a description of the nondeterministic choices made by the program is sufficient to guarantee a correct replay.

Most execution replay systems are based on the Instant Replay mechanism which was initially designed for shared memory parallel programming models [10], but it was later adapted to message passing models [11], to an asynchronous distributed programming model [7], and to a fairly complex object-oriented distributed programming model [8].

The only system that comes close to our work is the NOPE system [9] that was designed to deterministically replay MPI programs. In contrast to our work, the NOPE system considers promiscuous receive operations as the only source of nondeterminism. It does not deal with the nondeterminism caused by nonblocking test operations. This is however important: any program that depends on the number of test-operations performed (e.g., for time-outs) cannot be correctly replayed.

In the literature on record/replay, a great deal of attention is given to the reduction of the trace size because the this might be the factor that makes record/replay not feasible for long-running programs. A substantial reduction of the trace size could be obtained by using vector clocks [14] (for message passing systems), Lamport clocks [15] (for shared memory systems), and the properties of the programming model [5]. Our work goes beyond all the attempts for the compression of the trace data that originates from the test-functions. We successfully aggregate a succession of failed test-events into a counter and a final

test-event. To the best of our knowledge, aggregating events has never been used as a technique for reducing trace sizes.

In this paper we start with an overview of the nondeterministic features of message passing libraries, followed by a description of the solutions we propose to efficiently deal with them during the record phase and the replay phase. We then continue with an effective trace compression scheme based on the aggregation of nonblocking test-functions. This scheme not only reduces the size of the traces, but also speeds up the replay considerably.

2 Nondeterministic features of message passing libraries

We assume that messages in a point-to-point communication comply with the non-overtaking property which means that successive messages sent between two nodes are always received in the order in which they were sent (this is the case for MPI and PVM). Hence, since messages are assumed to be produced deterministically, receive operations in a private point-to-point communication that specify the sender are also deterministic.

The non-overtaking property does however not hold for messages that originate from different senders. Therefore, the so-called promiscuous receive operations (e.g., *MPI_Recv(...,MPI_ANY_SOURCE,...)*) which can receive a message from any other node are nondeterministic which means that the order in which communication events are taking place can differ between two executions, leading to different execution paths (see figure 1). In a sense, they play the role of 'racing' store operations in nondeterministic programs on multiprocessors.

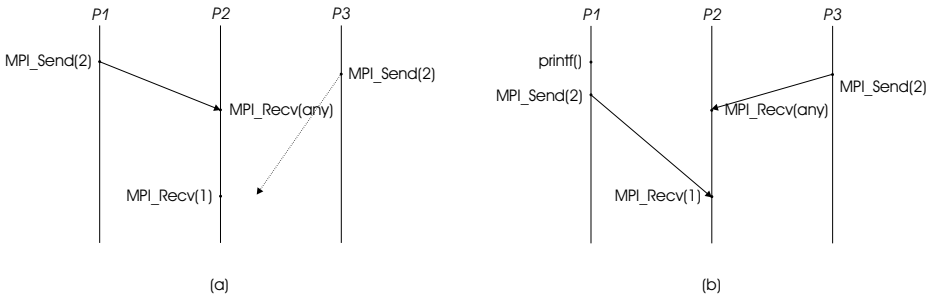

Fig. 1. The result of a promiscuous receive operation can depend on small timing variations.

Besides the promiscuous receive operations, there is another class of instructions that can cause nondeterminism in a message passing program:[1] nonblocking

[1] This paper only deals with nondeterminism due to the parallel nature of a program. Sources of nondeterminism that are also present in sequential programs such as `random()` or external input are not treated in this article.

test functions. This class of functions has been overlooked in previous papers on record/replay for message passing libraries [9].

Nonblocking test operations are however intensively used in message passing programs, e.g., to maximally overlap communication with computation: a nonblocking receive operation returns a request object as soon as the communication is initiated, without waiting for its completion. The request objects can be used to check the completion of the nonblocking operations by means of test-operations, which can in turn be blocking (*Wait*), or nonblocking (*Test*).

By the very fact that the test operations are nonblocking, they can be used in polling loops. The actual number of calls will depend on timing variations of parallel program, and is thus non-deterministic. Although many programs will not base their operations on the number of failed tests, some could do so (e.g., to implement a kind of time-out), and hence cannot be correctly replayed when not recorded (see Figure 2).

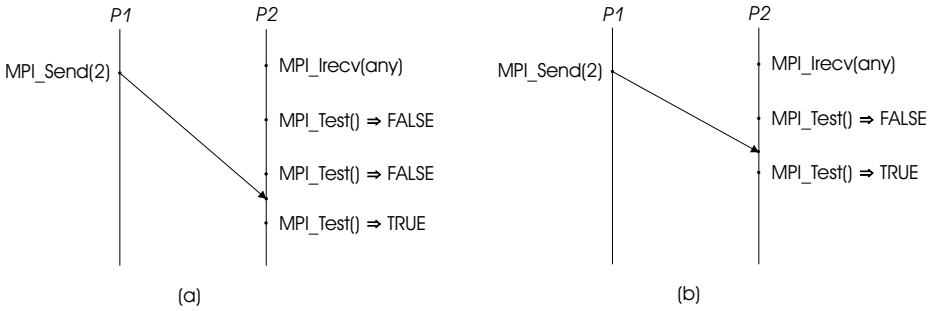

Fig. 2. The number of nonblocking test operations can depend on small timing variations.

A similar situation may occur for series of calls to functions testing for the arrival of a message (`MPI_IProbe`).

3 Execution replay for promiscuous receive and nonblocking test operations

As mentioned above, it is possible to use control driven replay for replaying message passing programs: it suffices to make sure that all nondeterministic choices are made in exactly the same way. In this paper, we focus on promiscuous receive operations and nonblocking test operations.

For the promiscuous receive operations, it suffices to record the actual sender of the messages received. This information can then be used during replay to force the promiscuous receive operations to wait on a messages originating from a particular sender process. Hence, a promiscuous receive operation can be made deterministic during replay by replacing it on-the-fly by a point-to-point receive

operation. Notice that this does not mean that the promiscuous receive operation is statically replaced by a point-to-point receive operation. Instead, every individual call to the promiscuous receive is dynamically replaced by the corresponding point-to-point receive.

For the nonblocking test operations, control driven replay implies that the setting of the condition that is tested for (e.g., the arrival of a message), must be postponed until the required number of test-operations is carried out. Only then, the operation that sets the condition can be allowed to resume. Replaying these test functions is no problem for a pure control driven replay system, where the order of *all* message operations is logged.

This approach has however a serious drawback. Nonblocking test operations are typically used in polling loops where they can in theory run for a very long time. Every iteration of the polling loop will add a bit of information to the trace file, although the polling loop is actually used to wait on a particular event. The fact that the trace file grows while the process is just waiting is not acceptable. It turns out that contents driven replay is actually more efficient for test functions than control driven replay. This is because a test function normally reports a series of failures, followed by a success unless the program finished before the test succeeds, when the request is cancelled (`MPI_Cancel`) or when the application stops polling and uses an `MPI_Wait` to wait for the completion of the request. Since all test functions (for one request) are identical we can count them and log this single number. During the replay phase, this number is decremented for each executed test function and as soon as the number becomes zero, we return TRUE and force an `MPI_Wait`. For an unsuccessful series of test functions, we log a number that is bigger than the number of executed test functions. This will force all test functions to fail during the replay phase as the counter never becomes zero. A similar solution was adopted to record the number of unsuccessful calls to `MPI_IProbe`.

Special care needs to be taken so that the number of unsuccessful tests is read from the traces as soon as the first `MPI_Test` of a series is performed. Similarly, in case of a non-blocking promiscuous receive operation`MPI_IRecv`, followed by a series of tests, the identification of the sender node needs to be read from the traces at the time the receive operation is called. These two problems were addressed by assigning request numbers to the first `MPI_Test` of a series as well as to each promiscuous `MPI_IRecv` call. These request numbers are stored in the trace files with the number of unsuccessful tests (resp. sender node identification) and the trace files are sorted before the first replayed execution. More technical details on the solution may be found in [4].

This approach dramatically reduces the trace size of test operations, especially in applications that use the nonblocking operations intensively.

Using contents based replay for the nonblocking test operations has yet another unexpected advantage. Since we do know the result of the failing test operations, there is actually no more need to call the actual message passing library routines anymore. We can now on-the-fly replace the failing test-operations by a call to a stub that just reports the failure, and replace the successful test

operation by the corresponding blocking test (*Wait*) operation. This only works if we can guarantee that the failing nonblocking test operations do not cause side effects, which is the case for MPI and PVM.

Replacing the message passing library routines by stubs has a dramatic effect on the replay speed of the nonblocking test-functions. Programs that do a lot polling on request objects might execute quite a bit faster during replay because the message passing library is not called anymore (see figure 3).

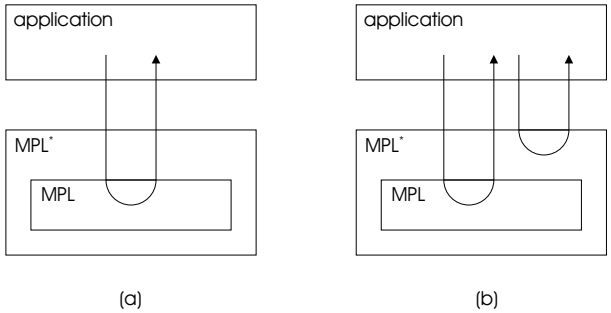

Fig. 3. MPL^* is a layer around the normal MPL. During the record phase MPL^* forwards all MPL-calls to the MPL (part (a)). During the replay phase, MPL^* does forward some calls to the MPL, while others call (e.g. the failed `MPI_Test` calls) simply return FALSE (part (b)).

4 Evaluation

The solutions that were proposed in this paper have been implemented in Athapascan-0 [4,1,2]. Athapascan-0 is a multi-threaded, portable, runtime system for parallel programs based on threads and message passing. The aims of Athapascan-0 and similar programming models are to ease the parallel programming of irregular applications, to mask communication and I/O latencies without elaborate efforts from the programmer, to offer a unified framework for exploiting both shared-memory parallelism and message passing [6].

The Athapascan-0 system is targeted towards hardware systems composed of a network of shared memory multiprocessor nodes. It offers two levels of parallelism: *inter* node and *inner* node. Inter node parallelism is based on a dynamically evolving network of threads, communicating by means of MPI [12]. The inner node parallelism is based on a fixed number of POSIX [13] compliant threads communicating using shared memory. Each Athapascan-0 node is implemented by a separate MPI process.

Implementing record/replay for Athapascan-0 requires record/replay for the nondeterministic features of both the thread-library, and the MPI-library. An implementation of record/replay for a *POSIX* compliant thread library has been

described extensively elsewhere [15]. Here, we only focus on the record/replay of the MPI-library. We have inserted our instrumented MPI-libary between the ATHAPASCAN-0-library, and the MPI-library. This allowed us to immediately replay ATHAPASCAN-0 program without additional coding.

The execution replay system was tested on several toy ATHAPASCAN-0 programs featuring all possible cases of nondeterminism of the programming model. It was also tested on the available programming examples of the ATHAPASCAN-0 distribution, and it was used to debug the ATHAPASCAN-1 system [3], which represents a large ATHAPASCAN-0 program.

Preliminary test results show that the time overheads during both the record and replay executions remain acceptable while the sizes of the trace files are fairly small. The table below shows execution times (wall time in seconds) for three toy test (not particularly efficient) programs, measured with a confidence range of 95 %. Mandelbrot computes a Mandelbrot set in parallel; queens(12) computes all solutions to a size 12 queens problem while scalprod computes the scalar product of two vectors of 100,000 floating point numbers each. The experiments were performed on two PCs running the Linux operating system and connected by a 100 MBit Ethernet connection.

program	normal	record	replay	logsize (bytes)
mandelbrot	1.858+-0.003	1.914+-0.037	1.864+-0.002	3912
queens(12)	6.70 +-0.15	6.58 +-0.05	6.86 +-0.17	1710
scalprod	4.38 +-0.04	4.44 +-0.07	4.66 +-0.03	874

Surprisingly, the slowdown of the replayed executions remains extremely low. Such a phenomenon stems probably from the behavior of the daemon thread used by ATHAPASCAN-0. When activated, a large part of its activity is probing communications and testing for the completion of non-blocking MPI requests. During replayed executions, most of the calls to *MPI_test* or *MPI_probe* are replaced by simple integer comparisons in the stubs.

5 Conclusion

This article describes a technique to efficiently record and replay the nondeterministic features of message passing libraries, in particular the promiscuous receive and the nonblocking test operations. We have tested out library on the ATHAPASCAN-0 parallel programming systems that is built on top of MPI. These tests have shown that our solution is highly efficient both in time overhead and size of the trace files.

Acknowledgment

This work was partially sponsored by the "Programme d'Actions Intégrés franco-belge Tournesol No. 98114". Michiel Ronsse is supported by the GOA-project 12050895 of the Universiteit Gent. Koen De Bosschere is research associate with the Fund for Scientific Reseach — Flanders.

References

1. J. Briat, I. Ginzburg, and M. Pasin. *Athapascan-0 Reference and User Manuals*. LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, March 1998. http://www-apache.imag.fr/software/ath0/.
2. J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference*, pages 590–599, Passau, Germany, Aug 1997. Springer Verlag.
3. Gerson G. H. Cavalheiro, François Galilée, and Jean-Louis Roch. Athapascan-1: Parallel Programming with Asynchronous Tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, USA, june 1998. http://www-apache.imag.fr/ gersonc/publications/yale98.ps.gz.
4. J. Chassin de Kergommeaux, M. Ronsse, and K. De Bosschere. Efficient execution replay for athapascan-0 parallel programs. Research Report 3635, INRIA, March 1999. http://www.inria.fr/RRRT/publications-fra.html.
5. A. Fagot and J. Chassin de Kergommeaux. Formal and experimental validation of a low-overhead execution replay mechanism. In *Proceedings of Euro-Par'95*, Stockholm, Sweden, August 1995. Springer-Verlag, LNCS.
6. I Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.
7. M. Hurfin, N. Plouzeau, and M. Raynal. EREBUS A debugger for asynchronous distributed computing systems. In *Proceedings of the 3rd IEEE Workshop on Future Trends in Distributed Computing Systems*, Taiwan, April 1992.
8. H. Jamrozik. *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants*. PhD thesis, Université Joseph Fourier, Grenoble, May 1993.
9. D. Kranzlmüller and J. Volkert. Debugging point-to-point communication in mpi and pvm. In *Proc. EUROPVM/MPI 98 Intl. Conference*, pages 265–272, September 1998.
10. Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.
11. E. Leu, A. Schiper, and A. Zramdini. Execution Replay on Distributed Memory Architectures. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 106–112, Dallas, USA, December 1990.
12. Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Standard*, May 1994.
13. Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proc. of the Winter USENIX Conference*, pages 29–41, San Diego, CA, January 1993.
14. R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing '92*, Minneapolis, Minnesota, November 1992. Institute of Electrical Engineers Computer Society Press.
15. M. Ronsse and L. Levrouw. An experimental evaluation of a replay method for shared memory programs. In E. D'Hollander, G.R. Joubert, F.J. Peters, D. Trystram, K. De Bosschere, and J. Van Campenhout, editors, *Parallel Computing: State-of-the-Art and Perspectives*, pages 399–406. North-Holland, Gent, 1996.