EXECUTION REPLAY FOR AN MPI-BASED MULTI-THREADED RUNTIME SYSTEM

M. RONSSE AND K. DE BOSSCHERE
RUG-ELIS, St.-Pietersnieuwstraat 41, B9000 Gent, Belgium

J. CHASSIN DE KERGOMMEAUX
LMC-IMAG, B.P 53, F38041 Grenoble Cedex 9, France

In this paper we present an execution replay system for Athapascan, an MPI-based multi-threaded runtime system. The main challenge of this work was to deal with nondeterministic features of MPI - promiscuous communications and varying number of test functions - without compromising the efficiency of an existing solution for execution replay of shared memory thread based programs. Novel solutions were designed and implemented in an efficient execution replay system for Athapascan programs.

1 Introduction

This paper describes an execution replay system for Athapascan [1], a runtime system for parallel systems composed of shared memory multiprocessor nodes connected by a network. Athapascan exploits both inter node and inner node parallelism. The nodes communicate using a communication library such as MPI and each node consists of different POSIX threads communicating through shared memory.

Athapascan is built on top of a POSIX thread library and an MPI communication library (Figure 1). The thread library offers mutexes and condition variables for synchronization purposes, while the communication library takes care of message passing. The Athapascan kernel extends the POSIX & MPI layer in two ways: (i) semaphore functions are added and (ii) mutually exclusive versions of the MPI functions are provided. The latter is necessary as most MPI implementations are not thread-safe (the functions are non-reentrant) and most thread-safe MPI implementations are not thread-aware: each time a thread calls a blocking communication primitive, it also blocks the other threads of the same node (Unix process). Therefore Athapascan uses a daemon thread that transforms all communications into non blocking communications and that regularly checks for incoming messages.

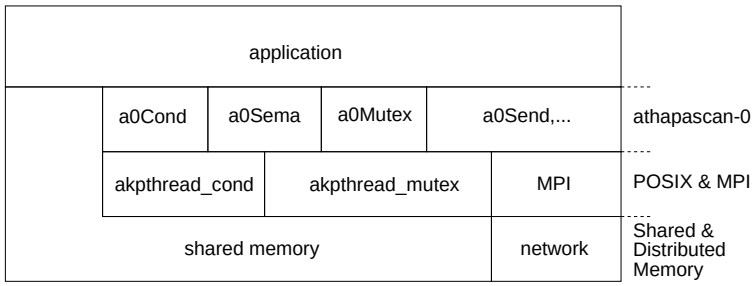

Figure 1. Athapascan is built using a layered approach, extending the functionality offered by POSIX threads and MPI messages.

2 Cyclic Debugging

Developing applications for such a complex system is difficult and error prone but it is possible to alleviate the debugging task by providing adequate debugging support. The basic support that comes to mind is cyclic debugging (watchpoints, breakpoints, stepping through the application, ...). Unfortunately, cyclic debugging assumes that a program execution can be faithfully re-executed any number of times, which is not the case for the class of nondeterministic parallel and distributed programs we envision. Even if two executions produce the same output, they are not guaranteed to have the same internal program flow which is essential if one wants to debug the application.

Known sources of nondeterminism are: certain system calls (such as `random()` and `date()`, ...), signals, unsynchronized accesses to shared memory (for parallel programs) and message exchanges (for distributed programs). There exists effective and fairly efficient ways to remove the sources of nondeterminism that are not caused by the parallel nature of the program (e.g., nondeterministic input can easily be recorded). This paper will only address nondeterminism caused by the parallel and distributed execution, i.e., race conditions between threads and processes. The standard technique to guarantee reproducibility of an execution is to implement an execution replay system [2]. Such a system makes a program execution reproducible by tracing a first execution (*record phase*) and by using the traced information to enforce equivalent re-executions (*replay phase*). The main challenge is to maximally limit the overhead during the record phase to limit as much as possible the probe effect.

3 Execution Replay for Athapascan

The abstraction level of recording is the POSIX and MPI layer. The reason is that the application programming interface of Athapascan includes many nondeterministic functions while its nondeterminism arises from the use of a very small number of POSIX or MPI functions. Moreover, this allows us to implement the record/replay method as a layer between the Athapascan layer and the shared memory and message passing layer and it allows the Athapascan layer[a] to be changed without worrying about the record/replay layer. The drawback of this choice is that more low-level function calls have to be considered than if recording were performed at the Athapascan level of abstraction. However, the amount of tracing could remain very low by combining the most efficient recording techniques for shared memory with novel solutions designed for the combination of nonblocking communication and test functions.

3.1 Shared memory

Replaying shared memory programs is possible by logging the order of the memory operations, and by imposing the same ordering during replay. It is obvious that this is very intrusive. Therefore, we only log the ordering of a subset of all memory operations: the synchronization operations. Forcing the same order during replay will guarantee a correct re-execution, provided the program contains no data races. The latter requirement can be tested using data race detection tools. These tools are very intrusive, but they can be used during replay as replay guarantees a correct execution up to the first data race [3].

The replay system used for Athapascan is based on ROLT (Reconstruction of Lamport Timestamps) [4] an execution replay system that only traces the partial order of synchronization operations by attaching a scalar logical timestamp to these operations. To get a faithful replay, it is sufficient to stall each synchronization operation until all synchronization operations with a smaller timestamp have been executed.

3.2 Message passing

As Athapascan uses LAM[5], an MPI implementation that is not thread-safe, for sending and receiving messages, Athapascan grabs a mutex while executing an MPI function. This causes these functions to be executed in a serialized

[a] Still in development.

way. Since the order of the mutex operations is traced during the record phase, the same serial ordering will be imposed during the replay phase. However, this is not sufficient for a deterministic replay: promiscuous receive operations –operations that do not specify the sender– and nonblocking communication operations require special attention.

Sending Messages MPI uses FIFO channels for its communication. This means that if a single thread sends two messages to a node, the messages will always be received in the same order. Even if the two messages are sent by two different threads on the same node, they will still arrive in the same order during replay because the MPI send operations are serialized per node by the Athapascan kernel. 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.

There is however a problem with messages sent by different nodes to a single node. In that case, the events are not serialized by the MPI library, and hence the order of events is not recorded, and might be replayed incorrectly. Fortunately, this problem can be solved at the receiver side.

Receiving Messages On the receiving side, events are also serialized. This time however, they are not serialized by the mutex guarding the MPI functions, but by the mutex of the Athapascan kernel daemon thread. As such, the order of the receive operations is automatically recorded and can be replayed. However, problems arise if the receive operation is promiscuous. A receive operation is promiscuous if a message can be received from any possible source (e.g. `MPI_Recv(...,MPI_ANY_SOURCE,...)`). These operations 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 2). In a sense, they play the role of 'racing' memory operations in nondeterministic programs on multiprocessors.

These messages should be received by replayed executions in the same order as during the recorded ones. We solved this problem by additionally recording the sender of a message for a promiscuous receive and by replacing a promiscuous receive by a regular receive operation (from the sending node) during replay.

Nonblocking operations A last source of nondeterminism is introduced by the nonblocking receive and send functions. This class of functions has been overlooked in previous papers on record/replay for message passing libraries [6]. Nonblocking test operations are however intensively used

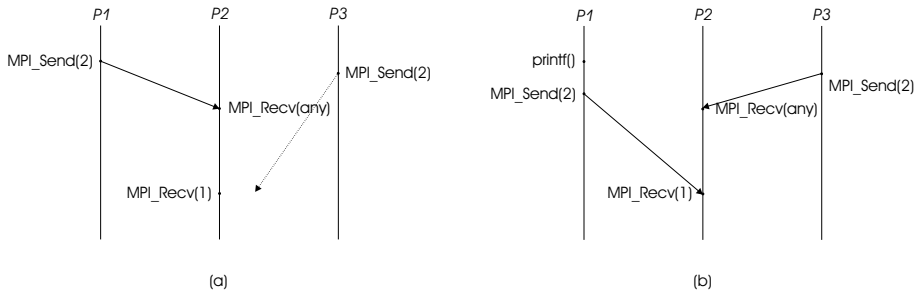

Figure 2. The result of a promiscuous receive operation can depend on small timing variations, e.g. caused by adding a `printf()` statement

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 nondeterministic. Although many programs will not base their operations on the number of failed tests, some could do so (e.g., to implement some kind of time-out), and hence cannot be correctly replayed when not recorded (see Figure 3).

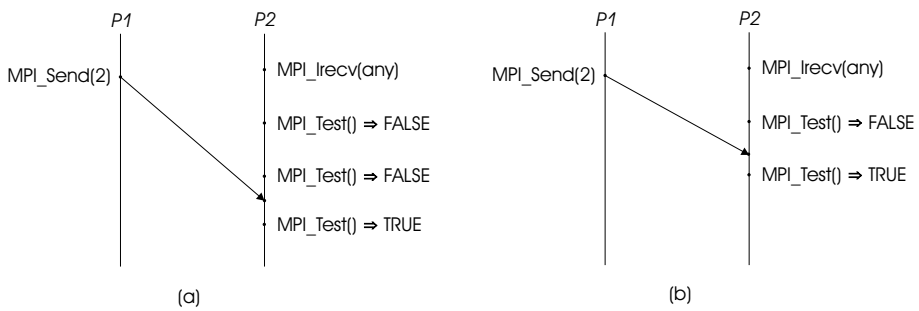

Figure 3. The number of nonblocking test operations can depend on small timing variations.

For the nonblocking test operations, ordering 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 ordering 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 ordering 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 (n). This greatly reduces the amount of information that has to be stored in the log files. It also has a positive effect on the replay speed: since all these test operations are now collapsed into one single execution of the operations, their replay can be many times faster than in the original execution. During this phase, the first $n - 1$ tests simply return FALSE and the last test returns TRUE and forces 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. A similar solution was adopted to record the number of unsuccessful calls to `MPI_IProbe`. Note that this approach is possible because the test functions have no side-effects: replaying less test functions than during the original execution introduces no problems.

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

4 Evaluation

An execution system has now completely been implemented in Athapascan and has already proven helpful in locating bugs in Athapascan itself. The execution replay system was tested on several 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 system itself.

Preliminary test results show that the time overhead 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 test 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 Mbps Ethernet connection.

program	execution time (s)			trace
name	normal	record	replay	size (b)
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

5 Related work

One of the first implementations of a replay mechanism for MPI was proposed in [7]. The tool uses a technique that is comparable to ours in order to deal with nonblocking operations, and uses the technique of Netzer and Miller to detect racing messages [8]. As such, a vector clock is attached to each message. The authors don't discuss the implications of this approach: what if messages become too big? The techniques used in [6] is also comparable to ours, but test functions are not dealt with. As such the number of test operations can vary between different replayed executions.

6 Conclusion

We have implemented execution replay for Athapascan, a runtime system using messages and shared memory for communication. We solved the problem of reconstructing the order of messages received from different (unsynchronized) nodes, and of efficiently recording and replaying nonblocking operations. We solved the first problem by not only recording the order of receive operations, but also the identification of the sending node per successful receive. We solved the latter problem by not recording individual nonblocking operations, but by counting the number of failed tests.

References

1. 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, August 1997. Springer Verlag.
2. 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.
3. Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.
4. Luk J. Levrouw and Koenraad M. Audenaert. An efficient record-replay mechanism for shared memory programs. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 169–176. IEEE Computer Society Press, January 1993.
5. Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*. Toronto, Canada, June 1994.
6. Dieter Kranzlmueller and Jens Volkert. Debugging Point-To-Point Communication in MPI and PVM. In *Proceedings of EuroPVM/MPI 98*, volume 1497 of *LNCS*, pages 265–272, September 1998.
7. Christian Clemencon, Josef Fritscher, Michael Meehan, and Roland Ruhl. An Implementation of Race Detection and Deterministic Replay with MPI. Technical Report CSCS TR-94-01, Swiss Scientific Computing Center, January 1995.
8. R.H.B. Netzer and B.P.Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, pages 502–511, November 1992.