PARALLEL PROGRAM DEBUGGING BASED ON DATA-REPLAY

Masao Maruyama[1,2], Tomoaki Tsumura[2] and Hiroshi Nakashima[2]

[1] Department of Information and Computer Engineering
Kisarazu National College of Technology
2-11-1 Kiyomidai-Higashi, Kisarazu,
Chiba 292-0041, Japan

[2] Department of Information and Computer Sciences
Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku,
Toyohashi, Aichi 441-8580, Japan

{maruyama, tsumura, nakasima}@para.tutics.tut.ac.jp

Abstract

Nondeterministic nature of parallel programs is the major difficulty in debugging. *Order-replay*, a technique to solve this problem, is widely used because of its small overhead. It has, however, several serious drawbacks: all processes of the parallel program have to participate in replay even when some of them are clearly not involved with the bug; and the programmer cannot stop the process being debugged at an arbitrary point. We adopt another method for deterministic replay, *Data-replay*, which logs contents of the events rather than their order, and makes it possible to run and stop each process independently. Data-replay is well able to cooperate with reverse execution mechanisms. We applied the Data-replay mechanism to MPI based parallel programs. The result of our experiment with NAS Parallel Benchmarks shows that our mechanism works at a practical cost. Logging communicated data incurs only 24 % overhead while it accelerates replayed execution by 38 %, both in average.

Key Words:parallel debugger, Data-replay, reverse execution, checkpointing

1 Introduction

Parallel programs are more difficult to debug than sequential programs due to their nondeterministic behavior.

Order-replay is a widely used solution for this problem, in which the order of nondeterministic events such as wildcard receive events are saved in logging execution and reproduced in subsequent replay executions. The technique has advantages of small logging overhead both in data size and time.

On the other hand, it has several serious drawbacks. First, the user cannot stop parallel processes being debugged in an arbitrary manner. For example, assume that there are two communicating processes; the process P_a sends a message M at the event S_a and P_b receives it at R_b. Also assume that we found an erroneous behavior of P_b and tried to detect its cause by inserting breakpoints in P_b. Then we examined the behavior of P_b in detail to find that the real cause is in the message M and try to know why and how P_a generated the erroneous message. At this point, however, we will realize that P_a has already completed its erroneous procedure to generate M and has proceeded too ahead to examine the cause of the error, because Order-replay execution must obey the causality of the events S_a and R_b. That is, it is impossible to stop P_a and P_b as we wish, at a point before S_a and a point after S_b. It is possible to rerun the program by Order-reply mechanism inserting a breakpoint at a point before S_a, but such a *overrun-and-rerun* debugging is seriously inefficient.

Another defect is that all processes of the parallel program have to participate in replay execution even when some of them are clearly not involved with the bug. The processes irrelevant to the bug are not only obstructive to debugging but also wasteful of expensive computing resource. Since a modern parallel computer may consist of hundreds or thousands of processors, this problem becomes more serious.

Thus we propose a debugging system based on *Data-replay*, another technique for deterministic replay. Data-replay is similar to Order-replay in logging/replay execution scheme, while it logs the contents of events such as received messages instead of the event order, and replay is performed on single process basis.

Data-replay allows the user to run an arbitrary subset of the processes, and to stop any process at any point independently of the others. Moreover, it is well able to cooperate with reverse execution mechanisms which provide another powerful means for parallel program debugging.

The concept of Data-replay itself is not very novel but had been considered as a poor predecessor (e.g. [1]) of Order-replay which LeBlanc first proposed as Instant replay[2]. Then Data-replay has been almost ignored by parallel debugging community because it has been believed that data logging incurs an impractically large overhead. Our primary contribution is to disabuse this impracticality superstition with a real implementation for MPI programs and its performance evaluation as discussed in this paper. Another important contribution is to exhibit that Data-reply is not only practical but also more efficient than Order-replay especially when it is combined with reverse execution.

The rest of this paper is organized as follows. The next section describes the concept of our Data-replay technique, and Section 3 presents its design and implementation. Section 4 shows the result of experiments to evaluate the system. After a brief discussion of related work in Section 5, we conclude the paper in Section 6 showing our future work.

2 Data-replay

As described above, Data-replay is a technique for deterministic replay. Its important characteristics compared to Order-replay are as follows.

Single process replay Replay is performed on single process basis, because all the data of interprocess events necessary for the execution of a process has been logged. Therefore not all processes of the parallel program being debugged have to be executed.

Once a data-log is collected on a full-scale parallel computer which is still expensive, the user may perform the rest of debugging process with fewer processors by only replaying processes being suspected or of interest. Moreover, since the computer for logging and replay are not necessarily same, the user may work the debugging process on a machine, for example a PC of the user's own, rather than keeping the full-scale computer occupied. In contrast, Order-replay requires all processes to run throughout the debugging session and often on the full-scale or a sufficiently large scale computer for reasonably quick execution of the debugged program.

No restriction on setting breakpoints Since each process is replayed independently, no restriction on setting breakpoints is imposed by interprocess causal relationship. The user may freely advance each process until it encounters an arbitrarily placed breakpoint. For example, we may run the process P_a discussed in the previous section setting a breakpoint at somewhere preceding the event S_a, *after* we found the problem in the execution of P_b with a breakpoint following R_b.

Well able to cooperate with reverse execution Reverse execution is another powerful means for debugging because it allows us to rollback a debugging execution to a point which we have already passed. However, the cost for reverse execution of a parallel program is often inacceptable because it usually requires occasional checkpointing which must be consistent throughout the whole parallel system. When used with Data-replay, on the other hand, we may use a simpler and faster checkpointing method because we need it only for a single process being debugged.

Moreover, an additional processor, e.g. one of dual-processor PC, will effectively hide the cost of reverse execution by running another replaying execution with checkpointing on it. When we are debugging a process occasionally stopping its execution for examination, a copy of the process runs on a *shadow* processor leaving checkpoints in an appropriate frequency. Then, if we wish to go back to a point past, the checkpoint most recent to the point should be available by the shadow execution. Note that checkpoints in *future* could also be available so that we may skip a part of the debugging execution quickly. Also note that this shadow checkpointing is impractical for Order-replay because it requires another large scale environment equivalent to that for foreground debugging.

Larger but reasonable overhead in logging execution Data-replay takes larger cost than Order-replay due to its larger logging data size. However, our claim is that the overhead involved with data logging mainly for received messages is not so large as to make Date-replay impractical.

Since communications between processes greatly affect the performance of a program, its programmer should try to reduce the size and frequency of the messages. At the same time, the cost of communication via network is comparable to that of write access to disks. Therefore, a program with a reasonably small communication overhead should be also reasonable to be executed with message data logging, as discussed in Section 4 with supportive experiment results.

3 Design and implementation

In this section, we describe the implementation of our Data-replay and reverse execution mechanism, which work in cooperation with each other. We applied the technique to MPI, a widely used message passing library. The current system works on MPICH-1.2.5.2 under Linux 2.4.7. Since it is implemented exploiting the MPI profiler interface and requires no modification of MPI's internal code, it is easily applicable to other MPI implementations.

3.1 Design overview

The system consists of three execution phases as shown in Fig. 1. In the first *logging* execution, the log of each event is recorded together with corresponding data (data log) so that subsequent reply phases reproduce same computation results without inter-process communication. The second *replay* execution, which is optional and may be *shadowed*, reproduces the result of one (or a few) of parallel processes referring the data log, and takes checkpoints in an appropriate frequency. Note that we separated execution phases for logging and checkpointing, unlike conventional systems such as Igor[3] and Recap[4] which perform both in one execution. This separation makes it possible to hide checkpointing overhead by shadowing, and to reduce the possibility to distort the program behavior in logging execution by lessening its overhead. The third and subsequent executions are for main work of debugging of one or a few processes. In this phase, the execution is not only replayed but also may be rolled back and forward to one of checkpoints.

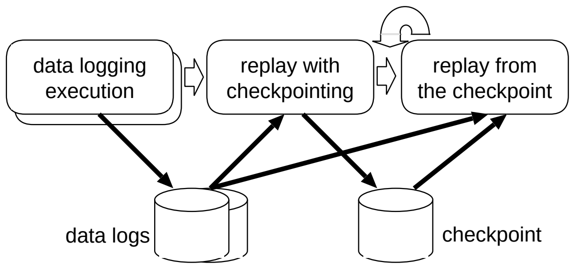

Figure 1. The execution phases.

3.2 Data-replay mechanism

In Data-replay, all irreproducible information must be recorded within logging execution in advance. In general, a process by itself can not reproduce the results of external events. Irreproducible information varies with where replay mechanism is instrumented. For instance, if it is under MPI layer, only the events making communication with other processes are irreproducible. Such an implementation records relatively small amount of log, but will depend on MPI library and/or underlying operating system.

We place our Data-replay layer between applications and MPI library, as shown in Fig. 2, in order to keep it independent of any particular MPI implementation and operating system.

In logging executions, our Data-replay library catches every MPI function call, then calls it and saves returned value from MPI and memory changes to log file before returning to the caller.

No actual MPI call is made in replay, but instead Data-replay library returns the data recorded at the corresponding event in the logging execution. In other words, it simulates MPI's behavior.

The system is implemented only using MPI profiling interface, and does not requires modification of either the application or MPI library.

3.3 Logged data

The system records information for analysis of event relation including message communicators, message tags and data types in addition to minimum data needed to Data-replay.

For example, following data are recorded for the function MPI_Recv.

- return value from the function
- MPI status
- contents of receive buffer
- source of the message
- message tag
- communicator

Only the first three are necessary for Data-replay and

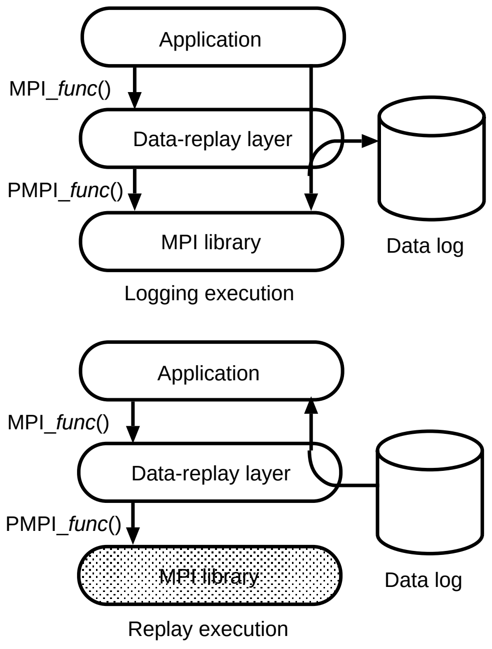

Figure 2. Data logging/replay mechanism.

the others only for the analysis of event relations. For MPI_Send,

- return value from the function
- destination of the message
- message tag
- communicator

are recorded. Note that message body is not logged here.

Some MPI functions require special handling as follows.

Nonblocking receive MPI_Wait does not return the address of the corresponding receive buffer, which is specified indirectly in its argument named *request-handle*. In order to locate the buffer, our system maintains a table to map a request-handle to the receive buffer corresponding to it. When a request-handle is created for a receive, it is registered in the table with its buffer address, and is referred when the completion of the receive is confirmed by MPI_wait.

Communicator creation Although information on the set of processes belonging to a communicator is unnecessary for Data-replay mechanism, it is helpful to the debugger users. On the invocation of functions creating a new communicator such as MPI_Comm_create and MPI_Comm_split, our system records the handle and the IDs of the processes (ranks in MPI_COMM_WORLD) belonging to it.

Table 1. Specification of the computer for the experiments.

Node processor	PentiumIII 866MHz
Memory	512MB
Network	1000Base-T
Number of Nodes	16(NPB)/43(OG)
OS	Linux 2.4.7

Since the additional information includes Order-replay log, a debugger based on our system may provide functions for analysis of event relationship and for execution control in a manner equivalent to or more flexible than Order-replay based systems. For example, the debugger can detect the point where a process is blocked waiting for a message in original execution, and stop the process until the sender process reaches the corresponding transmission if its user wishes an *in-step* or *synchronous* debugging. Such a debugging involving two or more processes is not inconsistent with our single process basis replay mechanism. Rather, our mechanism allows a debugger and its user to choose the synchronous debugging of bugs incurred by a miscoordination of processes, or the asynchronous one to concentrate one suspected process.

3.4 Reverse execution

Our system provide checkpointing and rollback facility for single process. In cooperation with Data-replay, it allows individual process to restart from an arbitrary point by rolling its execution back (or forward) to the checkpoint most recent to the point and then replaying the execution to the point.

We adopted "incremental checkpointing" in order to reduce the overhead. For data segment and heap, each checkpoint only includes the pages modified since the preceding checkpoint, while the stack is always fully saved.

The checkpoint interval is specified at run-time. The system polls the system timer whenever an MPI function is called, and takes the checkpoint if time is up. Asynchronous checkpointing with timer interrupt is also possible, but our claim is that checkpoints synchronous to MPI events are more useful as rollback points because they are easily located in the event graph of the program being debugged.

In order to minimize the cost of checkpoint restoring, we perform the restore operation of incrementally saved checkpoints in reverse order. In other words, we first restore the pages saved in the target checkpoint, then those in the preceding checkpoint which are not yet restored, and repeat this procedure until the first checkpoint is examined. Thus the cost of restoring is almost proportional to the number of pages modified until the target checkpoint, rather than those saved in the target and predecessor checkpoints. Finally, the stack is restored from the target checkpoint.

4 Experiments

In this section we describe the experiments and performance study of our system. We implemented our Data-reply system and also an Order-replay for comparison. We measured the performance of both systems with seven programs from the NAS parallel benchmarks (NPB v2.3)[3], BT, CG, EP, IS, LU, MG and SP, and an Othello (aka. Reversi) playing program (OG) written by ourselves as an example of nondeterministic programs. All the programs are parallelized with MPI library, which is MPICH-1.2.5.2 in our experiments. As described in Section 3, each MPI function call is trapped by our mechanism through the MPI profiler interface in the logging and replay executions.

All experiments were performed on a cluster shown in Table 1. For NPB programs, problem class is B and the number of processes is 16. OG consists of 43 processes for parallel α-β search of the game tree of 6-degree and 10-level deep, whose topmost three-level nodes corresponds to parallel processes. All the performance numbers shown in this section are the averages of ten measurement runs.

4.1 Execution time and log amount

Table 2 shows the time of the logging and replay executions of two replay techniques, together with that of the base execution without logging nor replaying. The numbers shown in the table are for rank-0 processes of NPB programs, while rank-7, the leftmost leaf node process, is chosen for OG. These processes perform computation as much as or more than the other processes. The amount of logged data, which is recorded into local storages, is given in Table 3.

Execution time Execution time of Data-logging relative to the base execution is in the range of 1.00 (OG) to 1.68 (IS), and the average is 1.24, while the average overhead of Order-logging is 1 %. Although Data-replay technique is inferior to Order-replay for logging overhead as anticipated, we evaluated that 20 to 30 % overhead is still practical and acceptable in most cases.

Even if the overhead is unacceptably large, for example, to run programs *always* with logging or to avoid the *probe effect*, our Data-replay is still usable by combining with Order-replay. That is, we may run a program with Order-logging in daily work and, when we find a buggy behavior, run the program with Data-logging using Order-replay mechanism. This additional logging execution needs some cost of course, but more payout will be earned by the subsequent debugging run as discussed below.

Replay time of Data-replay is in the range of 0.30 (IS) to 0.96 (EP), and 0.62 in average. Order replay, on the other hand, takes almost same time as the base execution. The reason of the speedup in Data-replay is that the process does not need to be blocked in the communication and synchronization events such as MPI_Recv and MPI_Barrier.

Table 2. Execution time of benchmark programs.

	Base	Data log	Data replay	Order log	Order replay
BT	459.6(1.00)	487.4(1.06)	398.1(0.87)	457.3(0.99)	457.3(0.99)
CG	211.2(1.00)	337.2(1.60)	66.1(0.31)	228.0(1.08)	230.6(1.09)
EP	70.4(1.00)	71.4(1.01)	67.6(0.96)	70.1(1.00)	70.1(1.00)
IS	20.4(1.00)	34.3(1.68)	6.1(0.30)	20.5(1.00)	20.4(1.00)
LU	240.1(1.00)	264.0(1.10)	182.3(0.76)	240.5(1.00)	241.0(1.00)
MG	23.4(1.00)	32.5(1.39)	10.2(0.44)	23.8(1.02)	24.2(1.03)
SP	414.2(1.00)	456.7(1.10)	282.9(0.68)	416.3(1.01)	416.5(1.01)
OG	290.7(1.00)	291.3(1.00)	190.9(0.66)	291.8(1.00)	291.8(1.00)
Average	(1.00)	(1.24)	(0.62)	(1.01)	(1.02)

Table 3. Amount of logs of benchmark programs.

	# of events	Data log[MB]	Order log[MB]
BT	36436	506.6	0.46
CG	69928	850.4	0.96
EP	11	0.0	0.00
IS	42	88.9	0.00
LU	102561	153.4	1.37
MG	13627	45.7	0.19
SP	65418	907.2	0.84
OG	2823	0.2	0.20

Table 4. Amount of logs per second.

BT	CG	EP	IS	LU	MG	SP	OG
1.10	4.03	0.00	4.36	0.64	1.95	2.19	0.00

Our claim is that the replay cost is more significant than that of logging, because locating the cause of a bug usually requires to replay the execution repeatedly. For example, when a user needs to repeat halfway replay executions n times to fix a bug, the total execution time of Data-replay $T_d(n)$ and Order-replay $T_o(n)$ in average case of Table 2 are $T_d(n) = 1.24{+}0.62n/2, T_o(n) = 1.01{+}1.02n/2$ respectively. That is, the Data-logging overhead is almost recovered by the first replay, and subsequent replays achieve a significantly large time-cost reduction steadily.

Log amount Data-replay produced up to 907MB (SP) of log, while Order-replay log is 1.4MB (LU) or less. Although the Data-log size is much larger than the Order-log, as anticipated again, the amount less than 1GB causes no problem in modern computing systems. Another justification of the log size is given from the growth rate of Data-log shown in Table 4. This result exhibits that one-hour run of a program of the highest logging rate (IS) only consumes 15GB disk space approximately, which is about 20 % or less of the local disk space of modern PC clusters.

Table 5. Execution results of checkpointing.

	CP size[MB]	# of CPs	time[s](normalized)
BT	753.3	11.5	501.5(1.09)
CG	43.5	10.9	72.6(0.34)
EP	1.2	1.0	67.6(0.96)
IS	68.5	3.0	7.8(0.38)
LU	114.1	10.0	191.2(0.80)
MG	222.5	10.0	34.0(1.45)
SP	233.4	10.5	313.5(0.76)
OG	0.4	9.0	191.0(0.66)

4.2 Checkpointing and restoring

Next we measured the time and data size of checkpointing. In this experiment, the process of rank-0 (for NPBs) or rank-7 (OG) was replayed taking checkpoints. Checkpointing interval was set to 1/10 of the total execution time of each program. Table 5 shows the amount of checkpoint data, execution time and the number of checkpoints actually taken [1].

An important observation obtained from the result is that the total size of checkpoints is comparable with that of Data-log and thus it is feasible too. More importantly, the execution time is still shorter, much shorter in CG and IS, than the base execution except for BT and MG. Therefore, we may conclude that the checkpointing for reverse (and forward-skipping) execution becomes a practical debugging aid when it is combined with Data-replay technique.

Finally, we measured the performance of restoring. After taking 11 checkpoints for each process by adjusting intervals, we restored the execution states targeting the first, sixth and eleventh checkpoints. As shown in Table 6, later targets consume longer restoring time because they need more checkpoint traversals. The growth rate, however, is quite small because the restoring of each page is taken only

[1] As stated in Section 3.3, our checkpointing synchronous to MPI events will not produce checkpoints in a given interval if a program has a few events as in EP and IS. Also note that the fractions in the number of checkpoints are the results of timing variance of events in ten measurement runs.

Table 6. Execution time for rollback(in seconds).

	1st CP	6th CP	11th CP
BT	5.42	5.86	6.03
CG	5.78	5.90	6.00
LU	5.45	5.77	5.86
MG	2.40	2.55	2.71
SP	5.41	5.72	5.86
OG	0.10	0.10	0.12

once regardless of the number of checkpoints as discussed in Section 3.4. For example, the restoring operation for the eleventh checkpoint of BT takes 6.03 seconds which is only about 10 % longer than that for the first one. This result, together with the fact that the absolute cost of restoring is less than 10 %, strongly supports that a roll-back or roll-forward operation in parallel program debugging is a means not only powerful but also efficient.

5 Related work

BugNet[1] is an early debugging system for message passing parallel programs based on checkpointing and Data-replay technique.

Igor[4] and Recap[5] are also examples of the debugger using checkpointing. Igor uses incremental checkpointing technique, while Recap has a mechanism with suspended processes for checkpointing. Unfortunately, computers of those days did not have enough capability for logging the entire message contents and/or process states. For example, Recap produces 1MB/s of log data on VAX11/780 whose disk space was indicated in MB. Therefore, it was the consensus that content-based replay is only used for tracing I/O and for tracing the result of certain system calls such as gettimeofday() due to its cost, as Ronsse stated in [6].

Order-replay was first proposed by LeBlanc as Instant Replay[2]. There are many parallel debugging systems based on similar techniques. Most of them make it a point to solve irreproducibility but deprecate efficiency in debugging session.

Netzer proposed an uncoordinated checkpointing technique[7] in which content of a message is logged only when it may cause “Domino effect” (according to their experiments, 1-10 % of messages). Although it reduces the logging cost, however, it still has serious drawbacks that (1) almost all processes participate in replay, (2) it is slower in replay than our method, and (3) its implementation strongly depends on the communication layer.

6 Conclusion

In this paper we proposed a debugging technique based on Data-replay and checkpointing. Since the Data-replay does not require all the parallel processes to participate in the replay execution, a program may be debugged on a smaller scale system in more flexible manner than debuggers based on Order-replay.

We implemented it for MPI and evaluated its performance comparing with Order-replay. The experiment results exhibit not only the practicality of the Data-replay with respect to its temporal and spatial cost but also the performance superiority to Order-replay in the replay phase which is usually repeated in debugging. We also exhibit the efficiency of a checkpoint-based rollback mechanism which provides another powerful means of debugging.

We are currently developing a parallel debugger with our Data-replay as the basis of a flexible execution control of processes. In order to provide the users with means of handling a huge events and processes, the debugger is planned to include a event/process manipulation language which is our another important work for parallel debugging.

Acknowledgments

This research work is partly supported by Japan Science and Technology Agency as a CREST research program entitled “Mega-Scale Computing Based on Low-Power Technology and Workload Modeling.”

References

[1] R. Curtis and L. Wittie. Bugnet: A debugging system for parallel programming environments. In *Proc. 3rd Intl. Conf. Distributed Computing Systems*, pages 394–399, 1982.

[2] T. J. LeBlanc and J. M. Mellor-Crummey. Debuggin parallel programs with instant replay. *IEEE Trans. Comp*, C–36(4):471–482, 1987.

[3] D. H. Bailey et al. The nas parallel benchmarks. *Intl. J. Supercomputer Applications*, 5(3):63–73, 1991.

[4] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices*, 24(1):112–123, 1989.

[5] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *ACM SIGPLAN Notices*, 24(1):124–129, 1989.

[6] M. Ronsse et al. Execution replay and debugging. In *proc. 4th Intl. Workshop on Automated Debugging(AADEBUG 2000)*, pages 5–18, 2000.

[7] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental replay of message-passing programs. In *Proc. the 1993 ACM/IEEE conference on Supercomputing*, pages 840–849, 1993.