An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications

Franco Zambonelli
Dipartimento di Scienze dell'Ingegneria – Università di Modena e Reggio Emilia
Via Campi 213-b – 41100 Modena, Italy
franco.zambonelli@unimo.it

Robert H.B. Netzer
Department of Computer Science
Brown University, Providence (RI)
rn@cs.brown.edu

Abstract

To support incremental replay of message-passing applications, processes must periodically checkpoint and the content of some messages must be logged, to break dependencies of the current state of the execution on past events. The paper presents a new adaptive logging algorithm that dynamically decides whether to log a message based on dependencies the incoming message introduces on past events of the execution. The paper discusses the implementation issues of the algorithm and evaluates its performances on several applications, showing how it improves previously known schemes.

1. Introduction

Debugging[1] long-running parallel/distributed programs requires the capability of incremental replay, i.e., of replaying selected intervals of an execution. Because programs that last hours or days are common, one should not be forced to replay the whole execution from the beginning to isolate a bug that manifested itself in a well defined section of the program.

To permit incremental replay, each process must periodically checkpoint, i.e., save its computational state to a stable storage, to allow its execution to be restarted from any one of its checkpoints and not from the beginning. Though coordinated checkpointing strategies are possible and widely explored [1], the paper addresses independent checkpointing: application processes can checkpoint independently of each other and without any coordination [7, 8].

In the message-passing model, several problems arise in replaying the execution of a process from one of its checkpoints. On the one hand, the order of message delivery must be traced and preserved during the replay, to grant deterministic re-execution. On the other hand, all the messages received by a process after the checkpoint of interest and until the next one need to be reproduced. The paper assumes tools are available for preserving the delivery order – widely studied in past works [4, 5] – and focuses on the latter problem. Two main techniques are possible: *(i)* all the messages received by a process are logged and restored during the replay [4]; *(ii)* the intervals of those processes from which messages have been received during the replay interval are re-executed too, in order to re-compute the messages and send them again [9]. The former technique is ineffective, because logging a message has high costs in both execution time and storage space. The latter technique, even if it were to introduce no overhead in the execution, does not grant any bound on the amount of computation that must be re-executed to replay a given interval: in the worst case, the whole execution may need to be replayed to re-compute the needed messages.

An alternative technique, known as *adaptive message logging*, can significantly reduce the logging effort while limiting the amount of computation needed to replay. This is done by introducing on-line algorithms to dynamically detect whether a message needs to be logged, on the basis of the dependencies on past events of the execution that it introduces on the receiver process [7]. Dependency information can be made dynamically available to processes by piggybacking it into the computation messages, thus avoiding the overhead of additional control messages.

This paper presents a new adaptive logging algorithm that grants a bound on the amount of computation needed to replay any given interval of the execution. The presented algorithm improves previously known logging algorithms in several ways: *(i)* it logs, in the average, a lower percentage of messages, as confirmed by tests on a set of message-passing applications; *(ii)* its behavior can be tuned to meet user needs; *(iii)* it prevents deadlocks during replay, a problem with some past schemes.

2. The model

A parallel/distributed program based on message-passing can be modeled as a set of processes $P_1, P_2, \ldots, P_n$ that execute either internal or communication events. Internal events of a process can include local checkpoint events. The paper indicates as $C_{i,x}$ the x^{th} the checkpoint taken by a process P_i and as $I_{i,x}$ its x^{th} checkpoint interval, i.e., the set of all the events included between $C_{i,x}$ and $C_{i,x+1}$. Communication events include message sending and receiving. Message delivery is not required to be FIFO.

Given a program, different executions are possible, depending on both the order in which messages are delivered and the time at which processes take local checkpoints.

2.1. The replay dependence relation

Given an execution of a checkpointed message-passing program, the replay dependence relation ($\xrightarrow{RD}$) shows how events would depend on one another during a replay [7]. An event b is said to be *replay dependent* on an event a ($a \xrightarrow{RD} b$) if and only if a must be re-executed before b can be re-executed, either because a precedes b in the same process and no checkpointing occurs between a and b or because a sequence of *unlogged* messages was sent from a (or a following event) to b (or a preceding event). The relation is transitive.

Let us consider the execution in figure 1, where horizontal arrows represent the execution of each process, black dots represent local checkpoints, and inter-process arrows represent messages exchanged between processes (solid arrows for unlogged messages, dashed arrows for logged ones). In this execution, the receipt of $m2$ makes the successive events of P_2 (until its next checkpoint $C_{2,2}$) replay dependent on the events of P_3 preceding the sending of $m2$. The receipt of $m4$ from P_2 makes P_1 replay dependent on P_2 and, transitively, on P_3.

The $\xrightarrow{RD}$ relation is included ($\subseteq$) but not equivalent to the *happened before* ($\xrightarrow{HB}$) relation [1]. In fact: *(i)* the first event after a checkpoint is not replay dependent on the last event before that checkpoint; *(ii)* a logged message does not introduce any replay dependence relation. With reference to figure 1, events of P_2 past $C_{2,1}$ are not replay dependent on any event of P_2 preceding $C_{2,1}$. Events of P_3 are not dependent on events of P_2 past $C_{2,2}$ because $m7$ is logged.

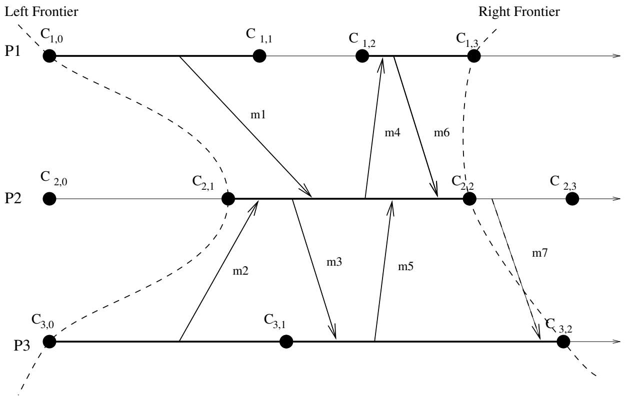

Figure 1. the replay set of $I_{2,1}$

2.2. The replay set

The replay dependence relation can be extended to checkpoint intervals. The notation $I_{i,x} \xrightarrow{RD} I_{j,y}$ indicates that there are events in $I_{j,y}$ that are replay dependent on $I_{i,x}$. Then, $I_{i,x}$ must be replayed in order to replay $I_{j,y}$. Note that the relation $I_{i,x} \xrightarrow{RD} I_{j,y}$ does not imply that all the events of $I_{i,x}$ introduce dependencies on $I_{j,y}$. In figure 1, $I_{2,1}$ is replay dependent on $I_{3,0}$ and, then, $I_{3,0}$ must be re-executed to replay $I_{2,1}$. However, events of $I_{3,0}$ past the sending of $m2$ do not introduce dependencies on $I_{2,1}$.

Given one interval I of an execution, there exists a set of checkpoint intervals that introduce replay dependence on it and need to be re-executed when replaying I. We call this the *replay set* associated with the interval. Formally:

Definition: For a given checkpoint interval I of an execution, the *replay set* $RS[I]$ is the set of intervals $RS[I] \equiv \{I_{i,x} : I_{i,x} \xrightarrow{RD} I\}$

In addition, the two sets of the earliest and of the latest checkpoints that delimitate the extension of the replay set are defined as its *left* and *right frontiers*, respectively. For example (figure 1), the replay set of the interval $I_{2,1}$, (emphasized by width lines) includes the intervals $I_{1,0}, I_{1,2}, I_{2,1}, I_{3,0}, I_{3,1}$; its left frontier is composed of $C_{1,0}, C_{2,1}, C_{3,0}$; its right frontier is composed of $C_{1,3}, C_{2,2}, C_{3,2}$.

3. The algorithm

The proposed adaptive logging algorithm is fully distributed: a process locally decides whether to log a message or not at the moment the message is received. The decision is based on information, locally stored by each process, about the replay set of its current checkpoint interval and on information, piggybacked with each message, about the replay set of the sender.

The basic idea of the algorithm is simple: because a message transitively induces on the receiver the replay dependence relations of the sender, a receiver will log a message if it would make the size of its replay set – i.e., the number of checkpoint intervals it is composed of – increase over a tolerated bound. Note that the replay set relation is included in the happened before one and, then, is detectable on-line [1]. Because the proposed algorithm on-line detects the exact shape of the replay set, it can be defined as *full informed*.

To keep track of its replay set, each process stores, for all the processes it is replay dependent on, the indexes of the associated checkpoint intervals that introduce replay dependence on it.

At any new checkpoint, all replay dependence relations are voided, i.e., for an interval I, $RS[I] = I$, because an interval is always replay dependent on itself. At any send event, the information about the local replay set is piggybacked with the message. At any received and not logged message m, the new replay set of the receiver becomes the union of its current replay set and of the one of the sender (piggybacked with m and denoted as $m.RS$), i.e., for an interval I, $RS[I] \leftarrow RS[I] \bigcup m.RS$. The intervals that are present in both replay sets are counted once in the union. If the message is logged, it does not introduce any new dependency and does not require the update of the replay set. This scheme can be summarized as follows:

```
m=receive();
if(size_of_union_of_RSs(RS, m.RS)>Bound)
    log(m);
else
    RS=compute_union_of_RSs(RS, m.RS));
fi
```

In general, the replay set of a process is likely to grow in size as the execution proceeds in a checkpoint interval and new messages are received. However, the proposed algorithm bounds the size of the replay set and, consequently, bounds the amount of information needed to keep track of it. The maximum amount of information one process will ever store and piggyback is $N + Bound$ integers, where N is the number of application processes and $Bound$ is the tolerated size of the replay set. For example, in an application composed of 16 processes, a bound of 16 for the size of the replay set requires to store and piggyback at most 32 integers with each message.

Table 1. description of the test programs

Program	Execution Time (sec)	Exchanged Data (Mbytes)	Avg. Message Size (bytes)
matrix determinant	48.3	43.1	3183
fast fourier transform	417.0	243.2	23233
finite differences	199.4	19.0	1241
circuit test generator	144.0	35.2	1641
VLSI channel router	1358.0	67.9	182

4. Evaluation

To evaluate the effectiveness of the presented algorithm, we adopted five message-passing programs as testbeds and simulated the execution of the algorithm from message traces of (non-checkpointed) executions of each program. The test programs, developed for a 16-node $iPSC860$ hypercube, include programs to compute the determinant of a matrix, the fast fourier transform and finite differences over a grid; a circuit test generator and a $VLSI$ channel router. Table 1 reports the basic characteristics of the test programs, which are heterogeneous in both execution times and amount of data exchanged, as well as in expressed communication patterns. Then, though the programs are not very long running, they can be considered representative for the evaluation of the replay algorithms, whose behaviour is determined by the communication patterns of an application rather than by the global execution time.

Checkpoint events have been artificially inserted in the message traces with different time periods: the interval between two checkpoints in a process has been varied from 1% to 50% of the application execution time. This range covers all practical cases and more: from a checkpoint every few seconds (see table 1) up to just one or two checkpoints in the whole execution. Checkpoints have been inserted in the traces with a random skew from their basic checkpoint period, to simulate the likely behavior of uncoordinated checkpointing.

Three indicators are significant towards the evaluation of the algorithm: *(i)* the percentage of messages logged during an execution measures the on-line replay cost, i.e., the logging overhead; *(ii)* the average and *(iii)* the maximum number of replay intervals per process required to replay the intervals of an execution (i.e., by considering all the intervals, the average size of the associated replay sets and the size of the largest one, divided by the total number of processes) measure the average and the worst case off-line replay costs, respectively.

One could criticize different metrics need to be introduced toward the effectiveness of the replay, such as the length of the longest sequential path needed to replay [6]. However, most of todays parallel and distributed architectures are not widely available at a cheap cost, and the replay activity cannot assume the availability of parallel executing resources. That makes it preferable to limit the total amount of computation rather than the parallel execution time, the former measure being independent on the amount of available computing resources.

4.1. Evaluation of the full-informed algorithm

The five test programs exhibit similar behaviors w.r.t. the appliance of the full informed algorithm. For this reason, the data relative to the different programs have been aggregated and averaged, to alleviate the presentation without significant loss of information.

Figure 2 shows the average and the maximum number of replay intervals per process, applying the full informed algorithm with different bounds on the size of the replay set. These figures also report the number of intervals required to replay in absence of any logging algorithm (no logging case). Figure 3 plots the corresponding percentage of logged messages.

As a first consideration, one can see from figure 2 that the algorithm generally achieves a significant reduction – w.r.t. the no logging case – in the amount of checkpoint intervals needed to replay, both in the average and in the worst case. However, the larger the application checkpoint period (and the length of checkpoint intervals) the less the relative reduction achieved by the algorithm – w.r.t. the no logging case – in the amount of checkpoint intervals needed to replay. In the case of very large checkpoint periods, the on-line logging efforts (mostly independent on the checkpoint period, as from figure 3), are not counterbalanced by a comparable reduction of the off-line replay costs. This identifies a general requirement for effective incremental replay rather than a peculiar limit of the proposed algorithm: an application must checkpoint frequently enough to make checkpoint intervals significantly shorter than the global execution time.

Apart from the above extreme situation, the behavior of the algorithm depends on the imposed bound on the size of the replay set. A too strict bound forces the algorithm to log a high amount of messages: for example, a bound of 16 intervals for the size of the replay set causes logging about 25-35% of the messages (figure 3). In this case, however, both the average and the maximum number of replay intervals per process are kept low, granting fast and low-cost incremental replay (figure 2). Larger bounds reduce the amount of logged messages and permit limiting the computation required for replay. A bound of 32 intervals on the size of the replay set limits the average number of replay intervals (figure 2-up) to about 1 per process and reduces the percentage of logged messages to 10-15% (figure 3). The worst case (figure 2-down) is bounded by 2.

The possibility of tuning the internal parameters of the algorithm permits users to select the preferred trade-off between on-line (logging) and off-line (amount of replay intervals) replay costs by selecting the most appropriate bound on the size of the replay set. If a user wants to minimize the on-line logging overhead, (s)he can choose a large bound for the replay set, tolerating a slower and more expensive replay activity. Conversely, if a user is in need of fast and cheap replay, (s)he can impose a very strict bound on the size of the replay set, paying the price of a higher on-line overhead.

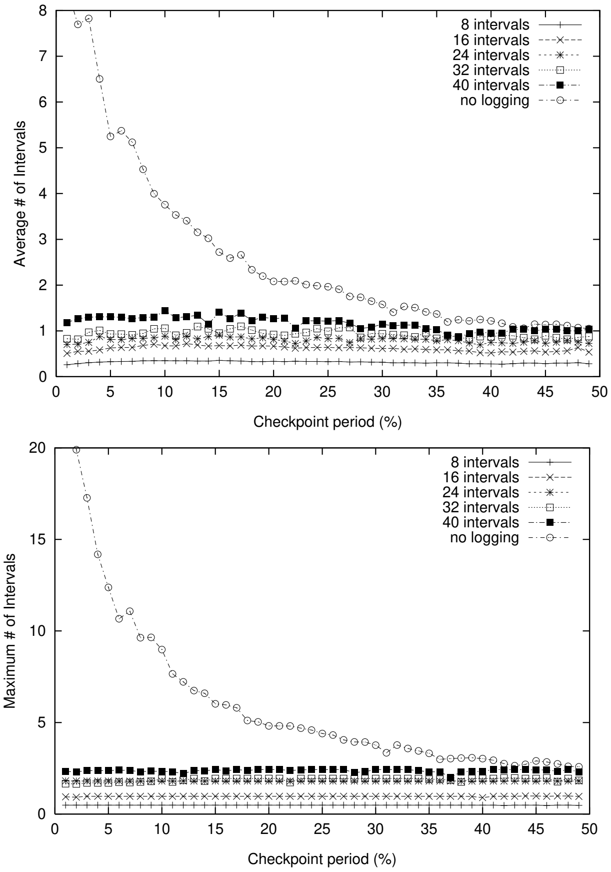

Figure 2. full informed algorithm: average (up) and maximum (down) number of replay intervals per process depending on the tolerated size of the replay set

4.2. Comparison with the domino algorithm

The *domino* algorithm for adaptive logging, proposed in [7], does not aim to exactly compute the replay set but only its left frontier. A vector of checkpoint indexes is locally stored by each process and piggybacked with messages, to track the earliest checkpoint interval of each process, if any, on which the current interval is replay dependent. A process logs a message if it introduces replay dependencies on past intervals of the process itself, i.e., domino dependencies.

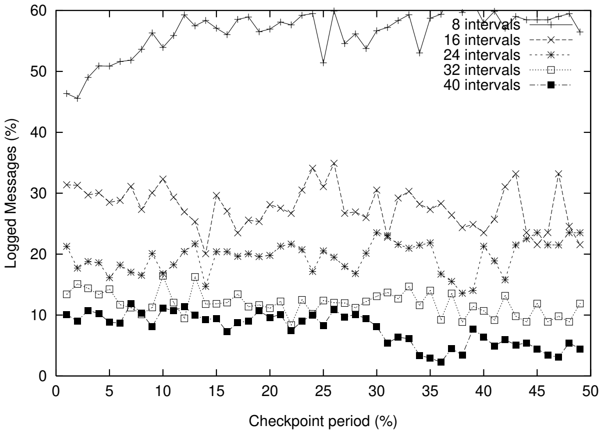

Figure 3. full informed algorithm: percentage of logged messages depending on the tolerated size of the replay set

As a first consideration, the domino algorithm is less flexible than the full informed one, because the logging function cannot be parameterized, thus precluding tuning the algorithm behavior to user needs. In addition, evaluated with the same test programs, it exhibits worse performance and can also deadlock the replay, as discusses later.

By setting the bound of the full informed algorithm so that it logs about the same percentage of messages as the domino algorithm, the two algorithms behave comparably w.r.t. the average number of replay intervals per process (figure 4-up). Instead, the maximum number of replay intervals per process is lower for the full informed algorithm (figure 4-down). In other words, with the same on-line costs, the full informed algorithm grants a lower worst case for the off-line replay costs.

By setting the bound in the full informed algorithm so that the maximum number of replay intervals per process is about the same in the two algorithms (40 intervals, 2,5 per process), the full informed algorithm logs a lower percentage of messages (figure 5). In other words, the full informed algorithm induces lower on-line replay costs to grant the same worst case for the off-line replay costs.

One could criticize that the performance improvement of the full informed algorithm over the domino one is not significant enough to justify its adoption, especially considering that the domino algorithm needs a reduced and fixed amount of information to be piggybacked with messages (a vector of checkpoint indexes). In our opinion, instead, reducing the amount of logging is more important than reducing by a few bytes the length of application messages. With reference to table 1, one can see that a reduction of 10% in logged messages saves several Mbytes of disk storage (and the cost of accessing it). On the other hand, increasing by a few bytes the amount of piggybacked information induces a limited additional overhead on applications, especially in the presence of efficient interconnection networks.

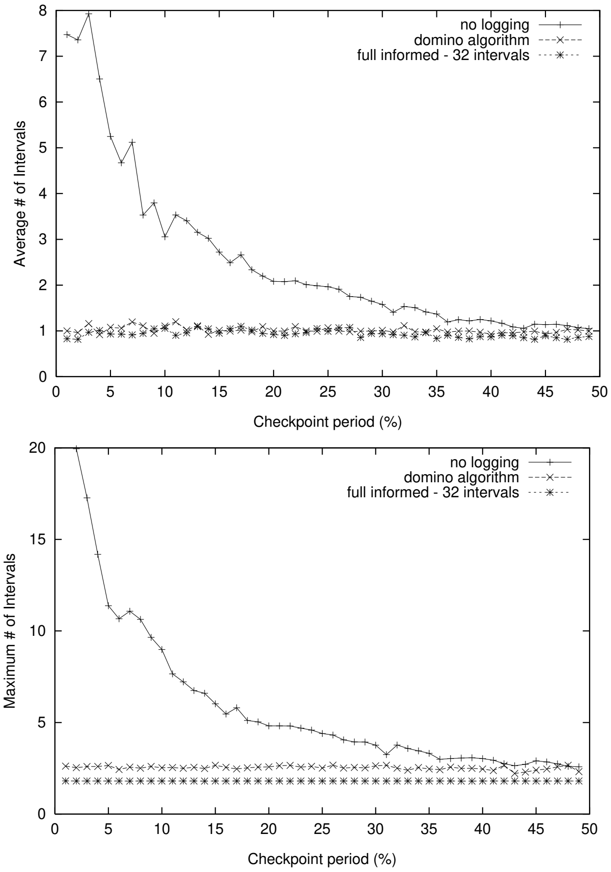

Figure 4. comparison of the algorithms: average (up) and maximum (down) number of replay intervals per process when logging about 15% of messages

5. Replay schemes

After an execution, to replay a given interval, its execution must be resumed from the corresponding checkpoint, together with the execution of all the intervals that belong to its replay set. Depending on the logging algorithm adopted, the post-mortem detection of these intervals and their re-execution introduce different problems.

Replay Scheme 1: In the full informed algorithm, each process has available on-line exact information about its current replay set. This information, if stored at each checkpoint, trivially grants the post-mortem detection of the replay set of each interval. Then, to execute a replay requires only to resume the execution of each interval of the replay set from the corresponding checkpoint. With reference to

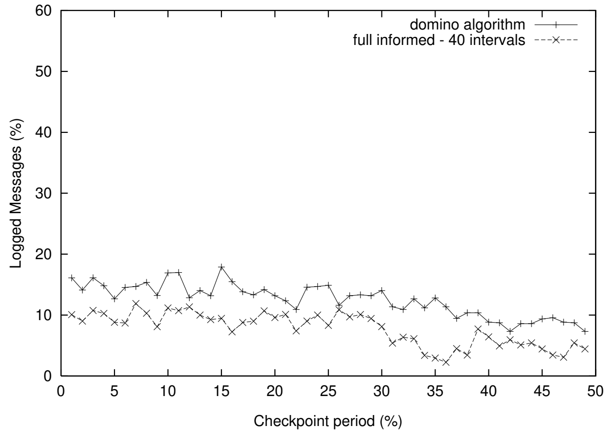

Figure 5. comparison of the algorithms: percentage of logged messages when bounding the maximum number of replay intervals per process to circa 2.5

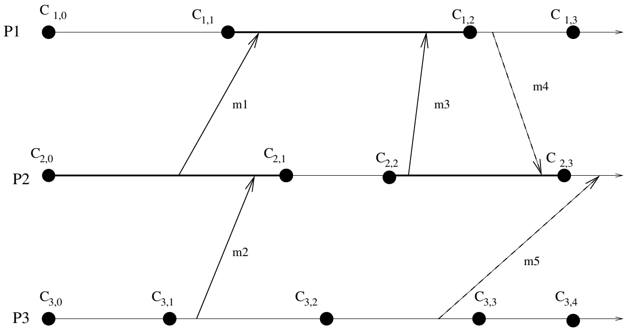

Figure 6. A deadlock in the replay of $I_{1,1}$

the execution of figure 6, to replay the interval $I_{1,1}$ one has to resume separately the execution of $I_{2,0}$ and $I_{2,2}$, together with $I_{1,1}$ itself, from the corresponding checkpoints.

In the domino algorithm, each process has available online only the information about the left frontier of the replay set. It has neither the exact shape of the replay set nor its right frontier. Then, one cannot execute a replay by resuming the execution of all the intervals of the replay set, because one cannot know which intervals are included in it.

Replay Scheme 2: The alternative scheme is to resume execution of the processes from their checkpoints on the left frontier of the replay set and let them proceed until the replay of the interval of interest completes, without resuming separately the execution of each interval. With reference to figure 6 and the replay of $I_{1,1}$, one can resume (in addition to P_1 from $C_{1,1}$) P_2 from $C_{2,0}$ and let its execution proceed until needed, i.e., over $C_{2,1}$ and $C_{2,2}$, until the replay of $I_{1,1}$ completes. This scheme, which is the one proposed in [7] and in [6], can also be applied in the case of the full informed algorithm.

Though simple and elegant, Replay Scheme 2 tends to waste execution resources. Firstly, the execution of some process could also proceed over the right frontier of the replay set. In addition, the replay cannot skip the execution of those intervals that are not included in the replay set, though included between its left and right frontiers. For example (figure 6), $I_{2,1}$ not belonging to the replay set of $I_{1,1}$.

Apart from wasting resources, Replay Scheme 2 is not generally applicable because of possible deadlocks. In figure 6, the replay of $I_{1,1}$ requires the replay of $I_{2,0}$, because of $m1$ and of $I_{2,2}$ because of $m3$. However, after having sent $m1$, the execution of P_2 blocks because the message $m2$ from P_3 will never arrive. In fact, no interval of P_3 is included in the replay set of $I_{1,1}$ and, then, P_3 is not replayed at all. P_2 will not proceed with its execution through $I_{2,1}$ and $I_{2,2}$, and will never be able to send $m3$, deadlocking the replay.

In general terms, the deadlock problem from which the Replay Scheme 2 suffers – not identified by previous works in the area – can be stated as follows:

Theorem 1. given the replay set $RS[I]$ of one interval I, if this replay set includes several intervals of the same process P_j, $I_{j,x}, \ldots, I_{j,x+n}$, and one of these intervals but the latest one, i.e., $I_{j,x+i}, i{\neq}n$, has a replay set $RS[I_{j,x+i}]$ not included in $RS[I]$, i.e., $RS[I_{j,x+i}] \not\subseteq RS[I]$, the replay deadlocks.

Proof. To replay the interval I, one has to replay $I_{j,x}, \ldots, I_{j,x+i}, \ldots, I_{j,x+n}$. Suppose that $I_{j,x+i}$ has a replay set that includes the interval $I_{k,y}$. Then, to complete the re-execution of $I_{j,x+i}$, let it proceed over $C_{j,x+i}$ and arrive until $I_{j,x+n}$, one has to re-execute $I_{k,y}$. Otherwise, the execution of process P_j blocks, transitively blocking the replay of I, because the intervals of P_j in its replay set included between $I_{j,x+2}$ and $I_{j,x+n}$ will never be re-executed. However, if $I_{k,y}$ is not included also in the replay set of I it will not replayed and the replay deadlocks. $\square$

To solve this problem and prevent deadlock in replay, different variations on Replay Scheme 2 can be devised.

Replay Scheme 2a: One can compute the transitive union of the left frontiers of all the intervals identified by the left frontier of the interval of interest, and re-execute any process from the left frontier obtained in that way. The problem of this solution is that it does not grant any bound on the amount of computation required: the replay can roll back in the past, unless the left frontier of any replay set is a *strongly* consistent global checkpoint [9, 3].

Replay Scheme 2b: Another possible solution is to evaluate, at replay time and for any receive event, whether the message is received from an interval belonging to the replay set. If it is, the message must be waited for and the execution can proceed afterwards. If it is not, the execution of the process can be stopped and resumed from its next checkpoint, avoiding the possibility of deadlock. This solution has the drawback of requiring the trace of every message received by a process. Though tracing messages is necessary to grant deterministic re-execution, adaptive tracing algorithms exist that trace only a fraction of the messages [5]. If these algorithms are used, it may be impossible to know, during the replay, from where a message must arrive.

It appears that the only general solution to prevent deadlocks is to adopt Replay Scheme 1, i.e., re-execute all the intervals of the replay set separately, from the corresponding checkpoints. This scheme, as already stated, is possible in the case of the full informed algorithm but not in the case of the domino one, because of the lack of the necessary information. Of course, one could apply the domino algorithm by making available to it the additional information about the replay set. In this case, however, there would be no reason not to exploit the additional information available to implement the full informed logging algorithm that, as shown in the previous section, achieves better performance.

6. Related work

In the past, several works in the area of parallel/distributed debugging have paid attention to message tracing techniques for deterministic re-execution [5], without addressing incremental replay.

Simple approaches to incremental replay propose to log the content of all messages [4]. However, the run-time cost of logging could be extremely high and could even exceed storage capability.

More recent papers propose adaptive logging algorithms for incremental replay. The domino algorithm was discussed earlier [7]. A logging algorithm to bound the critical-path, i.e., the length of the maximum sequential path needed to replay a given interval, is introduced in [6]: a message is logged only if its delivery would create a sequential execution path exceeding a specified bound. The algorithm is simple but does not grant a bound on the *global* amount of computation required for replay. It can be effective only if parallel execution resources are fully available for replay. Unfortunately, as we have already stated, this is not always the case. In addition, the algorithm exhibits the same deadlock problem as the domino one.

Other proposals aim to avoid logging while still granting fast re-execution. The approach proposed in [2] couples any execution of a parallel program with a twin execution, to be charged with all debugging activities, i.e., tracing and logging, making the original execution free of any on-line overhead. Though original and potentially effective, the approach is not easily applicable because it requires a large amount of resources. A formal analysis of the properties of a checkpointed execution is presented in [9]. This leads to the definition of an algorithm for the post-mortem detection of the intervals of an execution that can be replayed without message logging. The algorithm can be useful either to optimize message logging on subsequent executions or to couple the logging activity with a checkpointing activity. However, it does not allow the user to specify exactly where a re-execution can begin.

7. Conclusions and future work

The paper focuses on incremental replay of parallel/distributed programs and describes a new adaptive logging algorithm that improves previously known schemes by: *(i)* logging a lower percentage of messages; *(ii)* permitting the algorithm behavior to be tuned to user needs; *(iii)* preventing deadlocks in replay.

Future work will deal with testing the presented algorithm with a larger class of applications, possibly very long running. In addition, we are currently studying the relationship between adaptive logging and consistent checkpointing [3, 10].

Acknowledgements: Work partially supported by the Italian MURST within the Project "MOSAICO – Design Methodologies and Tools of High Performance Systems for Distributed Applications", and by NSF grant CCR-9702188.

References

[1] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems. In S. J. Mullender, editor, *Distributed Systems*, pages 55–96. ACM press, 1993.

[2] O. Gerstel, M. Hurfin, N. Plouzeau, M. Raynal, and S. Zaks. On-the-fly replay: a practical paradigm and its implementation for distributed debugging. In *6th IEEE Symposium on Parallel and Distributed Processing*, pages 266–272, October 1994. Dallas, TX.

[3] J. Helary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *16th IEEE Symposium of Reliable Distributed Systems*, October 1997.

[4] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(7):471–482, April 1987.

[5] R. Netzer and B. Miller. Optimal tracing and replay for debugging message passing programs. *The Journal of Supercomputing*, 8:371–388, 1995.

[6] R. Netzer, S. Subramanian, and J. Xu. Critical-path-based message logging for incremental replay of message passing programs. In *International Conference on Distributed Computing Systems*, pages 404–413, June 1994.

[7] R. Netzer and J. Xu. Adaptive message logging for incremental program replay. *IEEE Parallel and Distributed Technology*, November 1993.

[8] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[9] R. Netzer and Y. Xu. Replaying distributed programs without message logging. In *6th IEEE Symposium on High-Performance Distributed Computing*, August 1997.

[10] F. Zambonelli. On the effectiveness of distributed checkpoint algorithms for domino-free recovery. In *7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.