Active Optimistic Message Logging for Reliable Execution of MPI Applications

Thomas Ropars[1] and Christine Morin[2]

[1] Université de Rennes 1, IRISA, Rennes, France
Thomas.Ropars@irisa.fr
[2] INRIA Centre Rennes - Bretagne Atlantique, Rennes, France
Christine.Morin@inria.fr

Abstract. To execute MPI applications reliably, fault tolerance mechanisms are needed. Message logging is a well known solution to provide fault tolerance for MPI applications. It as been proved that it can tolerate higher failure rate than coordinated checkpointing. However pessimistic and causal message logging can induce high overhead on failure free execution. In this paper, we present O2P, a new optimistic message logging protocol, based on active optimistic message logging. Contrary to existing optimistic message logging protocols that saves dependency information on reliable storage periodically, O2P logs dependency information as soon as possible to reduce the amount of data piggybacked on application messages. Thus it reduces the overhead of the protocol on failure free execution, making it more scalable and simplifying recovery. O2P is implemented as a module of the Open MPI library. Experiments show that active message logging is promising to improve scalability and performance of optimistic message logging.

1 Introduction

The Mean Time Between Failures of High Performance Computing (HPC) systems is continuously decreasing as the scale of such systems keeps on growing. Efficient fault tolerance mechanisms are needed to enable applications to finish their execution despite frequent failures.

MPI is a widely used paradigm for HPC applications. Transparent fault tolerance solutions are attractive because all the mechanisms needed to handle failures are provided by the library. Thus the application programmer can focus on implementing her application without wasting time with the complex task of failure management. Rollback-recovery techniques are well known techniques to provide transparent fault tolerance for MPI applications [8].

Coordinated checkpointing is the most widely used rollback-recovery technique. Its main drawback is that the failure of one process implies a rollback of all the application processes. Furthermore with coordinated checkpointing all processes are checkpointed and restarted almost at the same time. When stable storage is implemented as a central server, issues due to concurrent access may occur. Message logging has the advantage over coordinated checkpointing protocols to minimize the impact of a failure since they do not require all processes to rollback in the event of one failure. It can be combined with uncoordinated checkpointing without the risk of domino effect. Additionally message logging is more suitable for applications communicating with the outside world [7]. Previous evaluations of rollback-recovery techniques in an MPI library [14, 3] have shown that message logging protocols, and especially causal message logging protocols, are more scalable than coordinated checkpointing. Furthermore it has been demonstrated that reducing the size of piggybacked data on application messages in causal message logging protocols has a major impact on performance [4]. Optimistic message logging protocols require less information to be piggybacked on messages than causal message logging protocols. Hence optimistic message logging can perform better than causal message logging.

In this paper we present O2P, an optimistic message logging protocol targeting performance and scalability. O2P is a sender-based message logging protocol tolerating multiple concurrent failures. It is based on an innovative active message logging strategy to keep the size of piggybacked data minimal. Implemented as a module of Open MPI [11], it is to our knowledge the first optimistic message logging protocol implemented in a widely used MPI library. Evaluations show that active optimistic message logging is a promising technique to provide an efficient and scalable fault tolerance solution for MPI applications.

The paper is organized as follows. Section 2 presents the related work and motivates this work. The O2P protocol based on active optimistic message logging is described in Section 3. We describe our prototype in Section 4. Experimental results are provided and discussed in this section. Finally, conclusions and future works are detailed in Section 5.

2 Background

2.1 Message Logging Principles

Message logging protocols assume that process execution is piecewise deterministic [21], i.e. the execution of a process is a sequence of deterministic intervals started by a non-deterministic event, a message receipt. This means that starting from the same initial state and delivering the same sequence of messages, two processes inevitably reach the same state.

Determinants [1] describe messages. A determinant is composed of the message payload and a tag. To identify messages, each process numbers the messages it sends with a sender sequence number (ssn) and the messages it receives with a receiver sequence number (rsn). A message tag is composed of the sender identifier, the ssn, the receiver identifier and the rsn. Message logging protocols save determinants into stable storage to be able to replay the sequence of messages received by a process in the event of failures. A message is stable when its receiver has saved the corresponding determinant in stable storage. Sender-based message logging [12] is usually used to avoid saving the message payload with the determinant, i.e. the payload is saved in the message sender volatile memory.

The deterministic intervals composing the process execution are called state intervals. A state interval is identified by an index corresponding to the receiver sequence number of the message starting the interval. Message exchanges create causal dependencies between the state intervals of the application processes. The state intervals are partially ordered by the Lamport's happen-before relation.If some determinants are lost in a process failure, some state intervals of the failed process cannot be recovered. Non-failed processes depending on those lost state intervals become orphan and have to be rolled-back to reach a consistent global state, i.e. a state that could have been seen during failure free execution. Message logging is generally combined with uncoordinated checkpointing. In that case, checkpointing can be seen as a solution to reduce the size of the logs. When a process is checkpointed, the determinants of all the messages it delivered previously can be deleted as soon as it is sure the process will never rollback the checkpointed state. We do not consider checkpoints in the rest of the paper.

The three classes of message logging protocols are: optimistic, pessimistic, and causal. A formal definition of these three classes can be found in [1]. They differ in the way they ensure a consistent global state is reachable after a failure. Pessimistic message logging never creates any orphan processes. Determinants are logged synchronously on stable storage: a process has to wait for the determinants of all the messages it has delivered to be stable before sending a message. Thus in the event of failures, only the failed processes have to restart. Causal message logging also ensures the no-orphan-process condition by piggybacking on messages the dependency information needed to be able to replay them. On message delivery, the receiving process saves locally those dependency data. Thus when a process fails the information needed to replay the messages is provided by the non-failed processes. Finally, in optimistic message logging determinants are saved asynchronously on stable storage. Application message sending is never delayed but orphan processes might be created. To be able to detect orphan processes, some dependency information is also piggybacked on messages. When a failure occurs all the orphan processes are detected due to the dependency information maintained by each process and are rolled-back to a consistent global state.

2.2 Message Logging Protocols Evaluation

During the last decade, the respective performance of existing rollback-recovery techniques has been compared through experimentation. The MPICH-V project that aims at providing a fault tolerant MPI library provided several results. In [14], they show that message logging tolerates a higher fault frequency than coordinated checkpointing. However pessimistic message logging induces large overhead on failure free execution due to synchronous logging [3]. That is why they implemented a causal message logging protocol [14] that provides better failure free performance since message sending is never delayed. However the causal protocol still induces some overhead because of the size of the dependency information piggybacked on application messages and because of the time needed to compute them before message sending and after message delivery.

Since the size of the piggybacked data is one of the key point in causal message logging performance, they promoted the use of an event logger to reduce this amount of data [4]. An event logger is an interface to a reliable storage where determinants are saved. The application processes send their determinants to the event logger that sends back an acknowledgement when a determinant is saved. The saved determinants do not need to be piggybacked on the messages anymore and thus the size of the piggybacked data is reduced. The evaluations, made with three causal protocols [9, 13, 14] showed that the use of an event logger has a larger impact on causal message logging performance than any optimization in the computation of the process dependencies. However for high message frequency, causal message logging combined with an event logger still induces a large overhead on failure free execution. Comparing the three classes of message logging protocols on recovery demonstrated that their performance on recovery is very similar [15].

Optimistic message logging requires less information to be piggybacked on messages than causal message logging, as explained in Section 3.1. Hence we have investigated optimistic message logging. Our goal is to take into account the use of an event logger in the design of an optimistic message logging protocol to make it more efficient and scalable than existing optimistic protocols

3 O2P, an Active Optimistic Message Logging Protocol

O2P is a sender-based optimistic message logging protocol based on active message logging to take advantage of the use of an event logger. In this section, we first briefly explain why optimistic message logging can perform better than causal message logging. Then we describe the system model we consider. Finally, we detail O2P and how it takes advantage of the use of an event logger.

3.1 Optimistic versus Causal Message Logging

Causal message logging combined with an event logger is very similar to optimistic message logging. However optimistic message logging can perform better on failure free execution because it requires less information to be piggybacked on application messages. This is illustrated by the simple scenario described in Figure 1. In this example, messages m_1 and m_2 have not been logged when m_3 is sent. A causal message logging protocol needs to piggyback on m_3 all the information needed to be able to replay this message in the event of a failure, i.e. determinants of m_1 and m_2 must be piggybacked on m_3. With an optimistic message logging protocol, only the data needed to detect orphan processes are piggybacked, i.e. the dependency to the last non stable state interval. So with optimistic message logging only m_2 determinant needs to piggybacked on m_3.

3.2 System Model

We consider a distributed application composed of n processes communicating only through messages. Our assumptions about communication channels are the one described in the MPI Standard [10]. Communication channels are FIFO and reliable but there is no bound on message transmission delay. Each process has

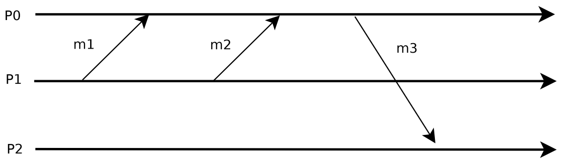

Fig. 1. A simple communication pattern

access to a stable storage server. An event logger act as the interface between the application processes and the stable storage. Information saved in volatile memory is lost in a process crash and only the information saved on stable storage remains accessible. We assume a fail-stop failure model for processes.

3.3 Protocol Description

Optimistic message logging protocols need to track dependencies between processes during failure free execution to be able to detect orphan processes in the event of a failure. Several optimistic message logging protocols are described in the literature varying mainly in the way they track dependencies. To be able to tolerate multiple failures, protocols use either dependency vectors [21, 19] or fault tolerant vector clocks [20, 6]. Damani et al. [5] have demonstrated that dependency vectors have properties similar to Mattern's vector clocks.

O2P uses dependency vectors to track transitive dependencies between processes. A dependency vector is a n entry vector, n being the number of processes in the application. Each entry is composed of an incarnation number and a state interval index. The incarnation number of a process is incremented each time it restarts or rolls-back. Incarnation numbers are used to discard orphan messages coming from old incarnations.

Existing optimistic message logging protocols implement the determinant's logging on stable storage as a periodical task. Determinants of the messages delivered by a process are buffered in the process memory and are periodically saved on stable storage. They don't focus on minimizing piggybacked data size to improve failure free performance. Vectors of size n are piggybacked on application messages to trace dependencies between processes state intervals. This solution is not scalable since the amount of piggybacked data grows with the application size.

However it has been proved that tracking dependency to non-stable state intervals is enough to be able to detect orphan processes after a failure [5]. A process state interval is stable when the determinants of all messages delivered by that process before this state interval have been saved on stable storage. Furthermore we proved in [16] that with tracking dependencies only to non-stable state intervals, it was possible to recover the maximum consistent global state of the application after a failure. O2P takes advantage of these properties to reduce the size of dependency vectors it piggybacks on application messages.

```
Sending a message $msg$ by process $p_i$ to process $p_j$
    Piggyback $DependencyVector_i$ on $msg$    // $DependencyVector_i$ is the dependency vector of process $p_i$
    Send $msg$ to $p_j$
    Save $msg$ in volatile memory

Delivering a message ($msg$,$DependencyVector_{msg}$)
    Get ($msg$,$DependencyVector_{msg}$) from the incoming queue
    $si_i \leftarrow si_i + 1$    // Incrementing state interval index
    Deliver $msg$ to the application
    Update $DependencyVector_i$ with $det_{msg}$    // $det_{msg}$ is the determinant of $msg$
    Update $DependencyVector_i$ with $DependencyVector_{msg}$
    Send $det_{msg}$ to the event logger

Upon reception Ack($StableVector_{msg}$) on process $p_i$
    for all $0 \leq k < n$ such that $StableVector_{msg}[k] \geq DependencyVector_i[k]$ do
        $DependencyVector_i[k] = \perp$
```

Fig. 2. O2P failure free protocol

Failure Free Protocol. O2P logs determinants in an active way. Since O2P is a sender based protocol, message payload is not included in the determinants saved in stable storage. A simplified version of the failure free protocol is described in Figure 2. As soon as a message is delivered, its determinant is sent to the event logger to be logged. Process p_i saves in entry j of its dependency vector the last non-stable state interval of p_j its current state interval depends on. This entry is set to $\perp$ if p_i does not depend on any non stable state interval of p_j.

O2P is based on two optimistic assumptions. The first one is the traditional assumption used in optimistic message logging protocols: logging a determinant is fast enough so that the risk of experiencing a failure between message delivery and the log of the corresponding determinant is small. Thus the risk of orphan process creation is low.

The second optimistic assumption we use is that logging a determinant is fast enough so that the probability of having it saved before the next message sending is high. If the second assumption is always valid, all the entries in the dependency vector of a process sending a message are empty.

To take advantage of this, dependency vectors are implemented as described in [18]. Instead of piggybacking the complete dependency vector on every message, only the entries which have changed since the last send to the same process and which are not empty, are sent as piggybacked data. Thus if the second optimistic assumption is always valid, no data has to be piggybacked on the application messages.

Event Logger. The event logger is the interface between the processes and the reliable storage. When it receives a determinant from a process, it saves this determinant and sends back the corresponding acknowledgement. In order to make a process aware of the new determinants saved by other processes, the event logger maintains a n entry vector called $StableVector$. Entry k of the $StableVector$, updated each time the event logger saves a new determinant from process p_k, is the last stable state interval of p_k. This vector is included in the acknowledgements sent by the event logger. When a process delivers an acknowledgement, it updates its dependency vector according to the $StableVector$. This mechanism contributes to reduce the size of the piggybacked data by avoiding piggybacking information on already stable state intervals.

Recovery Protocol. After a failure, a failed process is restarted from its last checkpoint if any or from the beginning. It gets from the event logger the list of determinants logged before the failure. Then it informs the other processes of its restart and gives its maximum recoverable state interval. When a non-failed process receives a failure notification, it can determine if it is orphan according to its dependency vector. If its dependency vector is empty, it cannot be orphan. The non-orphan processes can continue their execution while the failed and orphan processes interact to find the maximum consistent global state as described in [16].

4 Performance Evaluation

4.1 Implementation Details

We have implemented O2P as a component of the Open MPI library. Open MPI provides a *vampire* Point-to-point Management Layer (PML) that enables to overload the regular PML to execute additional code on every communication request made by the application processes. Thus it enables to implement fault tolerance protocols. O2P is a component of the *vampire* PML framework.

The event logger is a mono-threaded MPI process that can handle asynchronous communications. This process is started separetely from the application processes. Application processes connect to the event logger when they start logging determinants.

Different solutions can be used to implement piggyback mechanisms. On each message sending, a new data-type can be created using absolute addresses to attach piggyback data to the application payload. Another solution is to send an additional message with the piggybacked data after the application message. Performance depends on the application and on the MPI implementation [17]. The evaluations we made showed that sending additional messages was more efficient in our case. So it is the solution we adopted for our prototype.

4.2 Experimental Settings

Our experiments are run on a 60-node cluster of Dell PowerEdge 1950 servers. Each node is equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 4 GB of memory and a 160 GB SATA hard drive. An additional Dell PowerEdge 1950 server equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 8 GB of memory and two 300 GB Raid0/SATA hard drives, is used to run the event logger. All nodes, equipped with a Gigabit Ethernet network interface, are linked by a single Cisco 6509 switch and run Linux 2.6.18.

Table 1. Communication rate of the Class A NAS Parallel Benchmarks for 16 processes

	BT	CG	FT	LU	MG	SP
Execution time (in s.)	23.32	0.72	1.54	15.03	0.62	14.73
Messages/second per process	13	256	6	198	108	41

4.3 Results

To evaluate O2P, we used 6 class A applications from the NAS Parallel Benchmark [2], developed by the NASA NAS research center. Table 1 shows the communication frequency per process of these applications. In the experiments, application processes are uniformly distributed over the 60 nodes of the cluster. The measurements we provide are mean values over 5 executions of each test. For the experiments, we compare the active optimistic message logging protocol used by O2P with a *standard* optimistic message logging protocol. This *standard* protocol is not the implementation of any existing optimistic message logging protocol but is a modified version of O2P that doesn't take into account acknowledgments sent by the event logger to update dependency vectors. So processes don't waste time updating their dependency vectors on acknowledgment delivery. But the amount of piggybacked data is also not optimized. We consider that the *standard* protocol behaves like the optimistic protocols described in the literature regarding piggybacked data management.

Size of the Piggybacked Data. To evaluate the impact of active message logging, we measure the amount of data piggybacked on each application message. We run the 6 benchmarks with 4 to 128 processes. Figure 3 presents the mean number of timestamps, i.e. the number of entries in the dependency vector, piggybacked on each message during the execution of the application. We first evaluate this amount of piggybacked data for the *standard* optimistic message logging protocol. This result is the number of timestamps a traditional optimistic message logging protocol would piggyback on application messages. This amount grows linearly with the size of the application, underlining the scalability limit of existing optimistic message logging protocols. The results for the active message logging strategy used by O2P are represented by the curve named *Active*. For applications with low communication rates, i.e. BT and FT, active message logging works perfectly up to 64 processes. Determinants are logged with a very short delay by the event logger. Most of the time, the processes do not depend on any non stable state intervals when they send a message. So they do not have any information to piggyback. Here active optimistic message logging efficiently reduces the amount of piggybacked data. For 128 processes, the amount of data piggybacked is almost the same as for standard optimistic message logging because the event logger is overloaded and does not manage to log the determinants in time. For the benchmarks with a higher communication frequency, this limit is reached earlier.

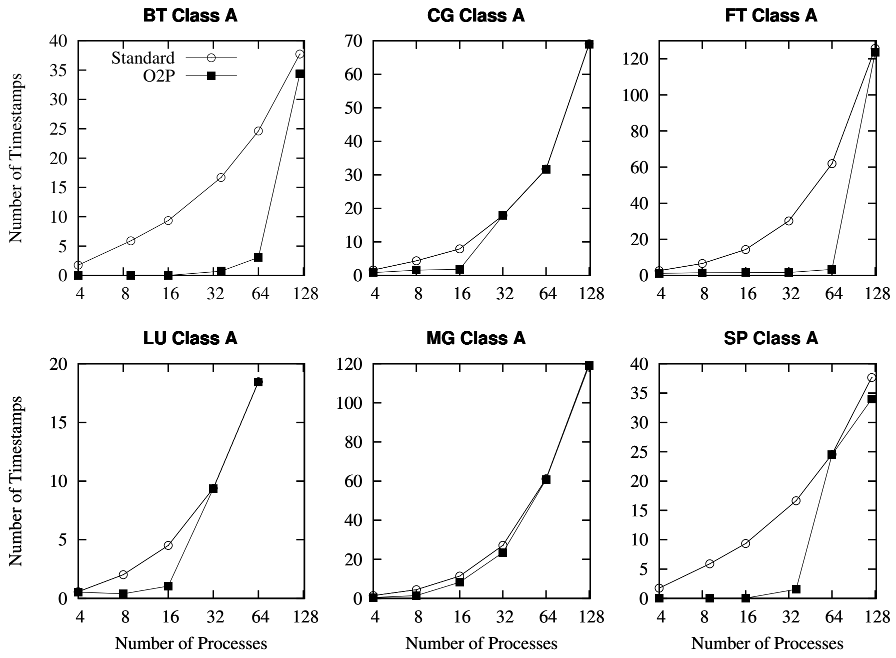

Fig. 3. Amount of data piggybacked on application messages

Failure Free Performance. The overhead induced by O2P on failure free execution for 4 representative benchmarks is summarized in Table 2. Evaluations are made with 16, 32, and 64 processes. Application BT requires a square number of processes. For this application, 36 processes are used instead of 32.

The overhead induced by the *standard* optimistic message logging protocol is compared with O2P. O2P almost always provides better performance than the standard protocol. For the applications with low communication rate, i.e. BT and FT, the overhead induced by O2P is very small. For CG, which has a very high communication rate, the overhead induced by optimistic message logging is significant even if active optimistic message logging contributes to reduce it.

4.4 Discussion

Results show that active optimistic message logging can reduce efficiently the amount of data piggybacked on messages during failure free execution. Thus the performance and scalability of optimistic protocol is improved. Furthermore when active optimistic message logging works well, the mean number of piggybacked timestamps is close to zero. It means that most of the time when a process sends a message it does not depend on any non stable state interval, i.e. the message will never be rolled-back. In this way, active optimistic message logging also reduces the cost of the recovery protocol because it limits the risk of orphan process creation. Furthermore the non-failed processes that do

Table 2. Overhead on failure free execution

N	Protocol	BT	CG	FT	MG
16	Standard	1.14%	12.85%	0.26%	6.80%
	O2P	0.98%	12.01%	0.13%	6.80%
32	Standard	2.49%	21.34%	0.95%	8.57%
	Active	2.22%	21.81%	0.72%	8.57%
64	Standard	3.71%	41.32%	9.23%	15.00%
	Active	3.13%	32.34%	0.00%	15.00%

not depend on any non stable state intervals when a failure occurs, can ignore the failure notification and continue their execution. Future works include the evaluation of the benefits of active optimistic message logging on recovery.

Experiments with large number of processes and with high communication rate applications indicate that the event logger is the bottleneck of our system. Implemented as a mono-thread MPI process, several solutions can be considered to improve it. The event logger could be implemented as a multi-thread MPI process to take advantage of actual multi-core processors. Active replication techniques could also be used to improve both performance and reliability of the event logger.Stable storage can be implemented in a distributed way using the volatile memory of the computation nodes. To make a data stable despite r concurrent failures, it has to be replicated in $r+1$ nodes. Hence we are investigating the implementation of an event logger based on a fault tolerant distributed shared memory such as Juxmem.

5 Conclusion

Message logging is an attractive solution to provide transparent fault tolerance for MPI applications since it can tolerate higher failure rate than coordinated checkpointing [14] and is more suitable for applications communicating with the outside world. It has been proved that making use of a stable storage to reduce the amount of data piggybacked by causal message logging on application messages makes it more efficient and scalable [4]. However causal message logging still induces a large overhead on failure free execution. Optimistic message logging requires less information to be piggybacked on messages than causal message logging. However, due to the risk of orphan process creation, the recovery protocol of optimistic message logging protocols is complex and can be costly. This is why existing optimistic message logging solutions are not widely used.

In this paper, we present O2P, a new optimistic protocol based on active optimistic message logging. Active optimistic message logging aims at overcoming the limits of existing optimistic message logging solutions. Contrary to existing optimistic protocols that saves determinants periodically, O2P saves determinants as soon as possible to reduce the amount of data piggybacked on messages and reduce the risk of orphan process creation. Thus scalability and performance of optimistic message logging are improved.

O2P is implemented as a module of Open MPI. It is to our knowledge the first optimistic message logging protocol implemented in a widely used MPI library. Experimental results show that active message logging improves the performance and scalability of optimistic message logging. Furthermore it simplifies recovery by reducing the risk orphan process creation. However our centralized implementation of the event logger, in charge of logging the determinants, is the actual bottleneck of the system. To overcome this limitation, we are investigating a solution based on a distributed shared memory to implement determinant logging. Future works also include the comparison with other message logging protocols implemented in Open MPI.

References

[1] Alvisi, L., Marzullo, K.: Message Logging: Pessimistic, Optimistic, Causal, and Optimal. IEEE Transactions on Software Engineering 24(2), 149–159 (1998)

[2] Bailey, D., Harris, T., Saphir, W., van der Wilngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report Report NAS-95-020, NASA Ames Research Center (1995)

[3] Bouteiller, A., Cappello, F., Herault, T., Krawezik, K., Lemarinier, P., Magniette, F.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 25. IEEE Computer Society Press, Los Alamitos (2003)

[4] Bouteiller, A., Collin, B., Herault, T., Lemarinier, P., Cappello, F.: Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In: Proceedings of the 19th IEEE InternationalParallel and Distributed Processing Symposium (IPDPS 2005), April 2005, vol. 1, p. 97. IEEE Computer Society Press, Los Alamitos (2005)

[5] Damani, O.P., Wang, Y.-M., Garg, V.K.: Distributed Recovery with K-optimistic Logging. Journal of Parallel and Distributed Computing 63, 1193–1218 (2003)

[6] Damani, O.P., Garg, V.K.: How to Recover Efficiently and Asynchronously when Optimism Fails. In: International Conference on Distributed Computing systems, pp. 108–115. IEEE Computer Society Press, Los Alamitos (1996)

[7] Elnozahy, E.N., Zwaenepoel, W.: On The Use And Implementation Of Message Logging. In: 24th International Symposium On Fault-Tolerant Computing (FTCS-24), pp. 298–307. IEEE Computer Society Press, Los Alamitos (1994)

[8] Elnozahy, E.N(M.), Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys 34(3), 375–408 (2002)

[9] Elnozahy, E.N., Zwaenepoel, W.: Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. IEEE Transactions on Computers 41(5), 526–531 (1992)

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, `www.mpi-forum.org`

[11] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004, pp. 97–104 (2004)

[12] Johnson, D.B., Zwaenepoel, W.: Sender-Based Message Logging. In: Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing, pp. 14–19 (1987)

[13] Lee, B., Park, T., Yeom, H.Y., Cho, Y.: An Efficient Algorithm for Causal Message Logging. In: IEEE Symposium on Reliable Distributed Systems, pp. 19–25. IEEE Computer Society Press, Los Alamitos (1998)

[14] Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., Cappello, F.: Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In: CLUSTER 2004: Proceedings of the 2004 IEEE International Conference on Cluster Computing, Washington, DC, USA, pp. 115–124. IEEE Computer Society Press, Los Alamitos (2004)

[15] Rao, S., Alvisi, L., Vin, H.M.: The Cost of Recovery in Message Logging Protocols. In: Symposium on Reliable Distributed Systems, pp. 10–18 (1998)

[16] Ropars, T., Morin, C.: O2P: An Extremely Optimistic Message Logging Protocol. INRIA Research Report 6357 (November 2007)

[17] Schulz, M., Bronevetsky, G., de Supinski, B.R.: On the performance of transparent MPI piggyback messages. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 194–201. Springer, Heidelberg (2008)

[18] Singhal, M., Kshemkalyani, A.: An efficient implementation of vector clocks. Information Processing Letters 43(1), 47–52 (1992)

[19] Sistla, A.P., Welch, J.L.: Efficient Distributed Recovery Using Message Logging. In: PODC 1989: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing, pp. 223–238. ACM Press, New York (1989)

[20] Smith, S.W., Johnson, D.B., Tygar, J.D.: Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In: FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers, Pasadena, California, pp. 361–371 (1995)

[21] Strom, R., Yemini, S.: Optimistic Recovery in Distributed Systems. ACM Transactions on Computing Systems 3(3), 204–226 (1985)