A Lightweight Message Logging Scheme for Fault Tolerant MPI

Inseon Lee[1], Heon Y. Yeom[1], Taesoon Park[2], and Hyoungwoo Park[3]

[1] School of Computer Science and Engineering,
Seoul National University,
Seoul, 151-742, KOREA
{inseon,yeom}@dcslab.snu.ac.kr

[2] Department of Computer Engineering
Sejong University
Seoul, 143-747, KOREA
tspark@kunja.sejong.ac.kr

[3] Supercomputing Center, KISTI,
Taejon, 305-333, Korea
hwpark@hpcnet.ne.kr

Abstract. This paper presents a new lightweight logging scheme for MPI to provide fault tolerance. Checkpointing recovery is the most widely used fault tolerance scheme for the distributed systems. However, all the processes should be rolled back and restarted even for a single process failure to preserve consistency. Message logging can be used so that the other processes can proceed unaffected by the failure. However, logging all the messages tends to be prohivitively expensive. We note that the applications programmed using MPI follow certain rules and not all of the messages need to be logged. Our logging scheme is based on this observation and only the absolutely necessary information is logged or piggybacked. As a result, it is possible to greately reduce the logging overhead using our scheme and the experimental results matched well with the expectation.

1 Introduction

MPI is the defacto standard for writing parallel programs running on parallel computers, network of workstations(NOW), and computational GRID[1]. Although programming using MPI is more complicated than programming using distributed shared memory systems(DSM), it is more widely used since it is easy to support on various computing platforms and has far better performance. In today's large scale distributed systems, a node failure is not something that rarely occurs but a frequent event which needs to be carefully dealt with. For the distributed systems to be of any practical use, it is important for the system to be recoverable so that the processes do not have to restart from the beginning when a failure occurs. However, most MPI implementations do not address the fault tolerance issues.

There are quite a few approaches trying to provide fault tolerance to MPI. Cocheck [2] and Starfish [3] provide checkpointing based fault tolerance. These methods rely on the consistent set of checkpoints and all the processes have to be restarted in case of a failure. MPIFT[4] employs pessimistic message logging and MPI-FT[5] provides both pessimistic and optimistic message logging. Other approaches include FT-MPI[6] and MPICH-V[7]. Almost all of these approaches rely on some form of indirect communication to log messages as well as guarantee consistency. FT-MPI is the only exception and shows much better performance. However, application programmer should be aware of the checkpointing/logging activity and corrective operations need to be provided by the programmer. The main advantage of MPI over other message passing interfaces like PVM is the performance which results from direct communication without any mediator. If the message should be relayed in any reason, the performance suffers. We want to provide fault tolerance without any performance degradation.

In this paper, we present a causal logging implementation used in MPICH-GF[8], a fault tolerant MPI implementation based on the MPICH-G2, the Grid enabled MPI. MPICH-GF supports coordinated checkpointing as well as independent checkpointing with message logging. Both Pessimistic message logging and optimistic message logging are supported. Optimistic logging alone can lead to cascading rollback and our implementation is augmented with causal logging. No additional communication layer was added so that the performance advantage can be retained while providing fault tolerance. Our design is focused on providing Globus users a sure way to execute long-running applications without having to worry about failures. A typical Globus user can execute an existing MPI application without modifying the application program source.

The rest of this paper is organized as follows: In Section 2, we briefly discuss the related works. The system model and the definition of consistent logging for the correct recovery are presented in Section 3. The protocols for causal logging and recovery are presented in Section 4. The performance of the proposed protocol is discussed with the experimental results in Section 5 and Section 6 concludes the paper.

2 Related Work

There are basically two ways to program distributed systems. One is to use message passing environment such as MPI or PVM and the other is to use distributed shared memory systems. In order to provide fault tolerance, checkpointing and logging is used in both cases.

Causal logging is one logging approach which is gaining a lot of attention for the message-passing based distributed computing systems [9]. In the causal logging technique, the sender-based logging of data items is performed and the access information is logged at the volatile storage of the dependent processes. Since this scheme completely eliminates the needs for stable logging, logging overhead can significantly be reduced. Also, since the storage of the dependent processes are utilized, concurrent and multiple failures can be handled. However, in this scheme, the log of the access information has to be causally spread over the dependent processes, which may cause the non-negligible message overhead.

A causal logging scheme for the DSM system based on lazy release consistent(LRC) memory model[10] has been suggested in [11]. In this scheme, to reduce the message overhead, the data structures and operations supported by the LRC model, such as *diff*, write notices, and vector clocks, are utilized. The authors has proposed another efficient scheme in [12] which further reduces the amount information carried by each message. Instead of logging the vector clock for each synchronization operation, the sufficient and necessary information to recreate the corresponding vector clock is inserted into the existing write notice structures. Similar technique can be applied to the causal logging based on MPI communication. The idea of causal logging for the message passing system was first introduced in [13] where the authors identify the necessary information to replay the messages for debugging purposes. Our implementation is based on this idea and MPI specific information is utilized to reduce the amount of data to be logged as was done in case of recoverable DSM systems.

3 Background

3.1 System Model

We consider a Grid system consisting of a number of fail-stop nodes [14], connected through a communication network. Each node consists of a processor, a volatile main memory and a non-volatile secondary memory. The processors in the system do not share any physical memory and communicate by message passing. Globus[15] is used to provide communication and resource management. Failures considered in the system are transient and a number of concurrent node failures may happen in the system. Applications running on the Grid systems is programmed using MPI and each application is executed on fixed number of nodes communicating with one another. The computation of a process is assumed to be *piece-wise deterministic*; that is, the computational states of a process is fully determined by a sequence of data values provided for the sequence of receive operations.

3.2 Consistent Recovery

We define a *state interval*, denoted by $I(i, \alpha)$, as the computation sequence between the $(\alpha - 1)$-th and the α-th synchronization operations of a process p_i, where $\alpha > 1$ and the 0-th synchronization operation means the initial state of p_i. Then, in the Grid system where applications communicate using MPI, the computational dependency between the state intervals can be defined as follows:

Definition 1: A state interval $I(i, \alpha)$ is dependent on another state interval $I(j, \beta)$ if any one of the following conditions is satisfied:
(a) $i = j$ and $\alpha = \beta + 1$.
(b) $I(j, \beta)$ ends with a $send(i, x)$ and $I(i, \alpha)$ begins with an $receive(x)$.
(c)$I(i, \alpha)$ is dependent on $I(k, \gamma)$ and $I(k, \gamma)$ is dependent on $I(j, \beta)$.

Definition 1.(a) indicates the natural dependency within a process, Definition 1.(b) presents the inter-process dependency caused by message passing, and Definition 1.(c) states that the dependency relation is transitive.

Definition 2: A state interval $I(i, \alpha)$ is said to be an orphan, if for any interval $I(j, \beta)$, $I(i, \alpha)$ is dependent on $I(j, \beta)$ and $I(j, \beta)$ is discarded by a rollback.

Definition 3: A process is said to recover to a consistent recovery line, if any state interval of the system is not an orphan after the rollback-recovery.

4 Protocol Description

4.1 Overview

Independent checkpointing in conjunction with causal logging is one way to achieve the consistent recovery. Let $Log(e_k)$ be the information logged to regenerate the exactly same event e_k. Under the assumption of the *piece-wise deterministic* computation, if $Log(e_k)$ for every receive event, which may cause the potential orphan state, can be retrieved at the time of rollback-recovery, the consistent recovery can be guaranteed. For the correct regeneration of an event, $Log(e_k)$ must include the message which have been provided for the receive event e_k, the identifier of the sender which has sent the message and the message sequence number. The causal logging consists of two parts; one is the sender-based logging of the message itself, and the other is the causal logging of the message access information, such as the message identifiers, by the dependent processes.

To uniquely identify a message, each message is tagged with the following information: the sender's id, receiver's id, sender's message sequence number and the receiver's receive sequence number. The sender's message sequence number is assigned when the message is generated. However, the receive sequence number is assigned after the message is actually received by the receiver. Both the sender's message sequence number and the receiver's receive sequence number are kept by each process and incremented whenever there is a send/receive event. These are reset to the checkpointed value in case of failure. For logging of the messages, the MPI_send_buf structure maintained by each process can be utilized, since this structure can be regenerated from a correct recovery even after a system failure. As for the event identifiers to trace the data access, the receive sequence number can be used along with the message id(sender id and sender sequence number).

However, in MPI applications, most receive calls are deterministic. Upon re-execution, when presented with several conflicting messages, the receive call may be able to select the same message it has received before without any additional information. When writing MPI applications, programmers assume that the ordering is preserved between communicating nodes. For some MPI implementations using more than one communication channel, it might not be true. However, in that case, programmers have to use caution so that the order reversal would not affect the outcome of the execution. When the receive call is specified with the source rank and tag, it only receives the messages identified with the same source and tag.

The only problem is the receive calls with no source specification, which receives messages from any source. It is sometimes used when a process expects messages from several different processes in no particular order. It can be re-written using non blocking receives and MPI_wait. If that is not the case, these receives should be dealt with care so that the same message can be delivered when it is re-executed. The message id should be causally logged along with the receive sequence number.

4.2 Checkpointing and Message Logging

Each process in the system periodically takes a checkpoint to reduce the amount of recomputation in case of a system failure. A checkpoint includes the intermediate state of the process and the messages sent after the last checkpoint. Checkpointing activities among the related processes need not be performed in a coordinated way, however, if checkpointing is incorporated into the barrier operation or garbage collection, the overhead of checkpointing can be reduced.

4.3 Causal Logging

The only event that needs to be tracked is the receive operation without the source specification. We call this non-deterministic receive. When MPI_receive is called with non-deterministic receive, the message id and the receive sequence number should be recorded. If there is a checkpoint after the non-deterministic receive, the dependency information can be discarded. However, if a message should be sent out after the non-deterministic receive, the dependency information should be piggybacked to the message so that the dependent process can provide the recovery information in case of failure. It is also possible to save the information to the disk from time to time to reduce the information to be piggybacked.

4.4 Rollback-Recovery

For a process p_i to be recovered from a failure, a recovery process, say p_i', is first created and p_i' broadcasts the *recovery message* to all the other processes in the system. The *recovery message* should contain the process id and the vector clock of the checkpoint it is restoring. On the receipt of the recovery message, each process p_j first determines whether it is a dependent of p_i or not. Being a dependent of p_i means that p_j has received at least one message from p_i after it took the checkpoint which is being restored. If so, it replies with its causality notice structure, which includes p_i's receive sequence number and the corresponding message id. When p_i' collects the reply message from every process, it eliminates the duplicates and reconstructs its own message replay information. The recovery process p_i' then restores the latest checkpoint of p_i and the messages received from the logging server is enqueued to the POSTED queue. From the reconstructed state, p_i begins the recomputation as follows:

MPI_send(p_j,m): If the last message sequence number received from p_j is bigger than that of m, skip sending the message. Otherwise, the message is sent normally.

MPI_receive(p_j,m) : p_i searches the message from the POSTED queue and delivers the message. If the message is not found, it means that recovery is complete and should proceed normally.

MPI_receive(*,m) : p_i searches the message replay information and selects the matching message so that the same message that was delivered before the failure can be delivered . If the information is not found, it means that recovery is complete and should proceed normally.

Theorem 1: The rollback-recovery under the proposed protocol is consistent.

Proof Sketch: If for every send/receive event e_α, an event e_β dependent on e_α exists, $Log(e_\alpha)$ can be retrieved after a failure. As a result, the rollback-recovery of a process must be consistent.

5 Performance Study

To evaluate the performance of the proposed logging scheme, we have implemented the logging scheme on top of MPICH-GF. Our experimental environment consists of a cluster of 4 PCs running LINUX 2.4 and Globus 2.2. Our MPICH-GF is based on the MPICH v1.2.3. Each PC has a 1.4GHz Pentium4 processor and 256MB of main memory.

We have run two sets of application programs, the NAS Parallel Benchmarks [16] and the SPLASH [17] applications. The NPB consists of EP, IS, CG, MG, and LU and the application progrmas we used from the SPLASH-2 suite are FFT, BT/SP, Quick Sort, TSP, and Water. By looking at the application programs, we have classified them into three groups. The first group is those applications with deterministic message receive. In other words, all the receive operations in this group specify the source of the receiving message. For these applications, there is no need to record the causality information since all the receive operations are deterministic. All the applications in the NPB except LU belongs to this group. From Splash-2, FFT and BT/SP belongs to this group.

The second group is those applications where there are non-deterministic receive operations. However, even these receive operations are deterministic since each message is uniquely tagged by the sender and processed accordingly regardless of their receiving order. TSP and Water show these characteristics. It is possible to re-write these applications using non-blocking receive and wait operations to create the same effect.

The last group where the causal logging is really needed has only one application, Quick Sort. It is a typical master-slave type parallel program where all the slaves communicate with the master to get the task to execute and the master assigns remaining tasks to the slaves. Only the master executes the non-deterministic receives and about half of the messages the master received has been causally logged.

The performance of those applications is shown in figure 1. The execution time is normalized against the execution time with no logging. The overhead from message logging is quite high (close to 80 %) for applications exchanging lots of messages such as MG and Water. However, the overhead decreases quite a bit if we apply our protocol. Even for the applications with fewer messages, we can see that handling only *anysource* messages is beneficial.

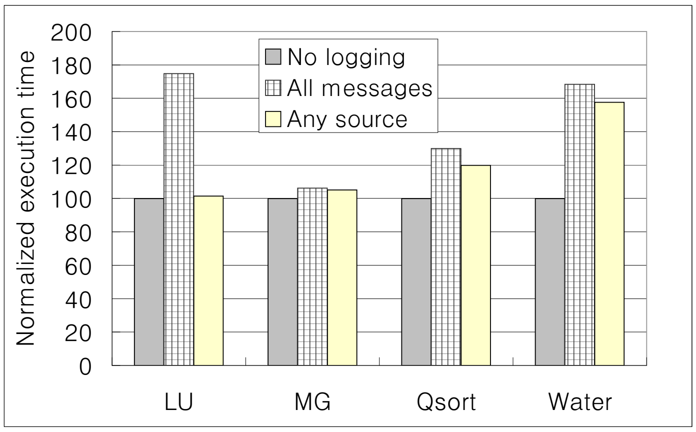

Fig. 1. The effect of Optimization

6 Conclusions

In this paper, we have proposed an efficient causal logging protocol for the Grid enabled MPICH implementation. The notable points of the proposed protocol is to reduce the amount of data to be logged using the MPI specific information. We note that only the messages received without the source specification should be logged and even that can be further reduced by closely looking at the application. As a result, causal logging can be achieved by piggybacking a small information to the outgoing message and the message overhead can be much smaller than the earlier logging schemes. To evaluate the performance of the proposed protocol, the logging protocol has been implemented on top of MPICH-GF, our fault-tolerant MPICH implementation for the Grid. The experimental results show that the proposed scheme can dramatically reduce the logs required for the causality tracking. MPICH-GF is available from http://dcslab.snu.ac.kr/projects/mpichgf/.

References

1. Foster, I., Kesselman, C. In: The Grid: Blueprint for a Future Computing Infrastructure. Morgan Faufmann Publishers (1999)
2. Stellner, G.: CoCheck: Checkpointing and process migration for MPI. In: Proceedings of the International Parallel Processing Symposium. (1996) 526–531
3. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In: Proceedings of IEEE Symposium on High Performance Distributed Computing. (1999)
4. Batchu, R., Skjellum, A., Cui, Z., Beddhu, M., Neelamegam, J.P., Dandass, Y., Apte, M.: MPI/FT:architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In: 1st International Symposium on Cluster Computing and the Grid. (2001)
5. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: Portable fault tolerance scheme for MPI. Parallel Processing Letters **10** (2000) 371–382
6. Fagg, G.E., Dongarra, J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: PVM/MPI 2000. (2000) 346–353
7. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Magniette, G.F., Néri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: SuperComputing 2002. (2002)
8. Woo, N., Yeom, H.Y., Park, T., Park, H.: MPICH-GF, transparent checkpointing and rollback-recovery for grid-enabled mpi processes. In: Proceedings of the 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing. (2003)
9. Alvisi, L., Hoppe, B., Marzullo, K.: Nonblocking and orphan-free message logging protocols. In: Symposium on Fault-Tolerant Computing. (1993) 145–154
10. Keleher, P.J., Cox, A.L., Zwaenepoel, W.: Lazy release consistency for software distributed shared memory. In: The 18th Annual International Symposium on Computer Architecture. (1992) 13–21
11. Yi, Y., Park, T., Yeom, H.Y.: A causal logging scheme for lazy release consistent distributed shared memory systems. In: Proceedings of the International Conference on Parallel and Distributed Systems. (1998) 139–146
12. Park, T., Lee, I., Yeom, H.Y.: An efficient causal logging scheme for recoverable distributed shared memory systems. Parallel Computing **28** (2002) 1549–1572
13. Netzer, R.H.B., Miller, B.P.: Optimal tracing and replay for debugging message-passing parallel programs. In: Proceedings of Supercomputing '92. (1992) 502–511
14. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Trans. on Computer Systems **1** (1983) 222–238
15. Foster, I., Kesselman, C.: The globus project: A status report. In: Proceedings of the Heterogeneous Computing Workshop. (1998) 4–18
16. NASA Ames Research Center: Nas parallel benchmarks. Technical report, http://science.nas.nasa.gov/Software/NPB/ (1997)
17. Woo, S., M. Ohara, E. Torrie, J.S., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd International Symposium on Computer Architectures. (1995) 24–36