A Message Logging Protocol Based on User Level Failure Mitigation

Xunyun Liu, Xinhai Xu, Xiaoguang Ren, Yuhua Tang, and Ziqing Dai

State Key Laboratory of High Performance Computing
National University of Defense Technology, Changsha, China
xuxinhai@nudt.edu.cn

Abstract. Fault-tolerance and its associated overheads are of great concern for current high performance computing systems and future exascale systems. In such systems, message logging is an important transparent rollback recovery technique considering its beneficial feature of avoiding global restoration process. Most previous work designed and implemented message logging at the library level or even lower software hierarchy. In this paper, we propose a new message logging protocol, which elevates payload copy, failure handling and recovery procedure to the user level to present a better handling of sender-based logging for collective operations and guarantee a certain level of portability. The proposed approach does not record collective communications as a set of point-to-point messages in MPI library; instead, we preserve application data related to the communications to ensure that there exists a process which can serve the original result in case of failure. We implement our protocol in Open MPI and evaluate it by NPB benchmarks on a subsystem of Tianhe-1A. Experimental results outline a improvement on failure free performance and recovery time reduction.

Keywords: Fault tolerance, Message logging, Checkpointing, User Level, Rollback-recovery.

1 Introduction

In a constant effort to deliver steady performance improvements, the size of High Performance Computing (HPC) systems, as observed by the Top 500 ranking, has grown tremendously over the last decade [1]. Unfortunately, the rise in size has been accompanied by an overall decrease in the mean time between failures (MTBF) [2]. In order to make large-scale parallel applications simultaneously survive crashes and mitigate the reliability-wall effects [3] in such systems, we need efficient and reliable fault tolerance mechanisms.

The Message Passing Interface (MPI) has become a de facto standard used to build high-performance applications [4], and fault tolerance for message passing applications is usually achieved by Checkpoint/Restart approach because of its simplicity of implementation and recovery [5]. However its recovery procedure is relatively time-consuming since the failure of one process makes all application

J. Kołodziej et al. (Eds.): ICA3PP 2013, Part I, LNCS 8285, pp. 312–323, 2013.

processes rollback to the last coordinated checkpoint. Message logging protocols present a promising alternative to Checkpoint/Restart, as they do not require coordinated checkpointing and globally rollback. Instead, only the crashed processor is brought back to the previous checkpoint, while the other processors may keep making progress or wait for the recovering processor in a low-power state [2].

To be more precise, message logging is a family of algorithms that attempt to provide a consistent recovery set from checkpoints taken at independent dates [6]. In message logging protocols, message exchanges between application processes are logged during failure free execution to be able to replay them in the same order after a failure, this is the so-called payload copy mechanism [7]. Also, the event logging mechanism is used to correct the inconsistencies induced by orphan messages and nondeterministic events, by adding the outcome of nondeterministic events to the recovery set, so it can be forced to a deterministic outcome (identical to the initial execution) during recovery.

Mainly due to the lack of support from the programming model, most of the previous implementations of message logging are located at the MPI library level, thus recent advances in message logging mostly focused on reducing the overhead of payload copy and event logging in the MPI library and had indeed achieved a reasonable fault tolerance cost [7–9]. But there are still few drawbacks in those researches: firstly, MPI itself has several different implementations (MPICH, Open MPI, etc.), thus it would take effort to transplant a message logging protocol designed for a specific MPI library to another environment. Secondly, fault tolerance ability for collective communications is provided by recording fine-grained point-to-point communications in the MPI library, which results in the inefficiency of the handling of collective operations in payload copy and recovery procedure.

In this paper, we adapt a message-logging protocol to run at the user level, rather than the MPI library level by building it on top of the User Level Failure Mitigation (ULFM) proposal [1]. Imposing a fault tolerance layer above ULFM certainly guarantees a level of portability, and recording the collective communication result into the sender's message logger as a whole alleviates the fault tolerant overhead for collective communications.

The rest of the paper is organized as follows. Section 2 introduces the basic idea behind the User Level Message Logging (ULML) by an example. Section 3 describes our fault-tolerance framework and the implementation. Section 4 discusses our evaluation methodology and demonstrates the superiority of our protocol over the classical method by benchmarking. Then section 5 reviews the related work, and finally, Section 6 concludes the paper.

2 Motivation and Basic Idea

This section starts by analyzing the drawbacks of classical library level message logging when handling collective operations. Afterwards, we introduce the motivation and basic idea behind the user level message logging (ULML).

As with most previous researches on message logging, we assume that the process execution is piecewise deterministic, and communications channels between processes are reliable and FIFO. Therefore, we will concentrate on the faults of computing processes with the assumption of fail-stop fault model.

2.1 Message Logging at Library Level

A parallel program with checkpointing is illustrated in the table below, its execution is constituted by 2 processes, denoted by P_0, P_1, which have been checkpointed to disk before executing any code.

```
1  --CKPT_HERE--
2  int a, int b;
3  MPI_Barrier(MPI_COMM_WORLD);
4  if(my_rank == 0)
5  {
6      a=4;
7    MPI_Send(&a,1,MPI_INT,1,0,MPI_COMM_WORLD);
8  }
9  if(my_rank == 1)
10 {
11   a=5;
12   MPI_Recv(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
13 }
14 MPI_Allreduce(&a,&b,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
```

The pessimistic message logging cited from [2] is chosen to illustrate the classical protocol at MPI library level. It is worth noting that even the recent advances have further refined the logging scheme to record only important events and messages at library level [8–10], they still confront with the same drawback as the original approach does when handling collective operations.

In this approach, a process, before sending a message, has to ask the receiver for a *ticket* (the Reception Sequence Number) to compose the determinant for that message. The determinant and the message are stored in the memory of the sender, and at that point the message can be actually sent to the receiver. Messages at the receiver are processed according to their assigned ticket number, and on recovery ticket numbers can be used to recreate the reception order of all messages. Furthermore, a collective communication should be divided into two point-to-point communications at library level, for the reason that the MPI library of a process must receive the ready signal of the opposing process to finish the implementation. Fig.1 demonstrates the communication procedure at the library level.

After executing this program, message m_1, m_2, m_3, m_4, m_5 and their assigned tickets will be recorded, which indicates that recording fine-grained point-to-point messages for one collective operation induces multiple payload overhead. If a process error occurs at this moment, the substitution needs to replay all the receptions in order of their tickets, thus the performance of recovery will be also remarkably slowed down when the scale of application rises.

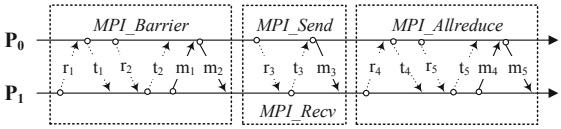

Fig. 1. The communication procedure of the example program with pessimistic message logging at library level, r_i represent requests, t_i represent tickets and m_i represent messages

2.2 Our Approach: Message Logging at User Level

Rabenseifner presented A five-year profiling study of applications running in production mode on the Cray T3E 900 at the University of Stuttgart, and it revealed that more than 40% of the time spent in MPI functions was spent in the two functions MPI_Allreduce and MPI_Reduce [11]. That implies collective communications account for a substantial percentage of total communication cost. Because of performing collectives frequently, scientific computing parallel programs magnify the drawbacks of library level message logging markedly, impelling us to explore an alternative solution at user level.

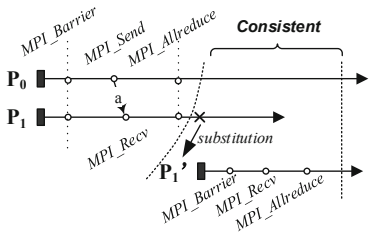

Fig. 2. The communication procedure of the example program with message logging at user level, P_1' is the substitution of P_1, and the system states are denoted by dashed lines

Our approach elevates the checkpointing/rollback mechanism, payload copy mechanism and recovery mechanism to user level, allowing us to record communication as a whole statement without splitting it into implementation details, so that we can re-transmit the result of collectives instead of individual point-to-point messages. For the same example program, Fig.2 presents all the communications at user level, and the payload copy mechanism is detailed below:

- `MPI_Barrier:` after finishing MPI_Barrier, each process increases a local variable representing the number of barrier operations executed by 1.
- `MPI_Send/MPI_Recv:` P_0 simply preserves the variable $\boldsymbol{a}$ into the message logger after sending the message to P_1.
- `MPI_Allreduce:` each process increases a local variable representing the number of all-reduce operations by 1, and then it logs the local variable $\boldsymbol{b}$ which is the result of the operation, into the message logger separately.

After executing this program, P_0 stores variables $\boldsymbol{a}$, $\boldsymbol{b}$ in the volatile memory as message logs, while P_1 only stores variable $\boldsymbol{b}$ into the message logger. Meanwhile, the statistics information of collective communications is updated in each process. If P_1 malfunctions at the end of the program, as shown in Fig.2, a new incarnation of the failed process denoted by P_1' is recovered from the checkpoint. By exchanging information between process P_0 and P_1', P_0 is informed that variable $\boldsymbol{a}$ needs to be resent and P_1' learns that there are a barrier operation and an all-reduce operation in the coming recovery procedure. So during recovery, P_1' skips the barrier operation, replays reception in MPI_Recv, and when executing MPI_Allreduce, P_1' does not replay the collective communication, instead, it receives the original operation result variable $\boldsymbol{b}$ from P_0 with MPI_Recv statement. Finally, the recovery system reaches a consistent global state after a failure.

2.3 Comparison of Overhead

The user level message logging significantly reduces the overhead of fault-tolerance for collective communications. taking MPI_Allreduce as an example: the all-reduce operation combines values from all processes and distributes the results to all processes, so it is often used when calculating and determining whether the computational accuracy meets the requirement or not at the end of the iteration in scientific computing programs. If we assume that (1)data are not compressed during the all-reduce operation and (2)source data items are independent of one another, table 1 shows the comparison of fault-tolerant overhead when we perform an all-reduce operation of X items of *itsize* bytes on P processes[12].

Table 1. Comparison of overhead induced by different message logging protocols

Message logging	At library level	At user level
Log number	$2 \times (P-1)$	P
Maximum size of each message log	$2 \times \frac{P-1}{P} \times X \times itsize$	$X \times itsize$
Fault free execution time-consumption	$\lceil \lg P \rceil (\alpha + n\beta + n\gamma + n\delta)$	$\lceil \lg P \rceil (\alpha + n\beta + n\gamma) + n\delta$
Minimum recovery time-consumption	$\alpha + n\beta$	$\alpha + n\beta$
Maximum recovery time-consumption	$\lceil \lg P \rceil (\alpha + n\beta)$	$\alpha + n\beta$

In the table above, a simple cost model is used to estimate the cost of the algorithm in terms of latency and bandwidth use. To be specific, α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte, and n is the number of bytes transferred. In the case of reduction operations, we assume that γ is the computation cost per byte for performing the reduction operation locally on any process, and δ is the preservation cost per byte for storing the message into memory.

3 Framework of the Protocol and Its Implementation

Our framework consists of three mechanisms: the **sender based payload copy mechanism** saves exchanged messages into volatile memory; the **communicator maintenance mechanism** is responsible for updating communicator when a process fails; once the improper communicator is updated, the **recovery mechanism** will resend logged messages to the substitution process and ensure the consistency of the system.

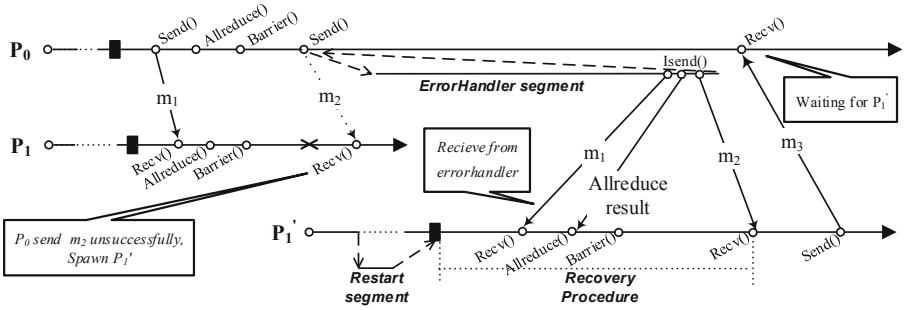

Fig. 3. Example execution of the user level message logging framework, the ErrorHandler segment and the Restart segment are highlighted in bold, and words in bubbles explain the actions of processes

The flow of the framework is illustrated in Fig.3 by an example. At first the application executes normally, P_0 sends a message m_1 to P_1 via MPI_Send and then records m_1 as a message log. When executing the all-reduce operation, both P_0 and P_1 preserve the operation result into memory. After finishing the barrier operation, P_1 fails unexpectedly. P_0 detects the communicator failure when it is trying to send m_2 to P_1, so it moves into the Error Handler segment automatically, then it spawns P_1' as a substitution process, re-transmits the logged message to help P_1' recover. After that, P_0 jumps out of the Error Handler, continues execution or waits for message m_3 from the opposite. On the other side, P_1' takes the place of P_1 in the communicator, jumps to the checkpoint address, reads the live variables and enters the recovery procedure by means of the execution of the Restart segment. During P_1''s recovery, all the external message it needs will be resent by P_0, and it will not replay any collective communication until the recovery completes successfully.

Next, we discuss in further detail about how these three mechanisms work.

3.1 Sender Based Payload Copy

Sender based payload copy rules can be classified into two categories: point-to-point payload copy and collective payload copy, which correspond to the two types of MPI communications.

Point-to-point payload copy for senders, messages sent to the different destinations are kept in different log queues, and those messages are sorted by the

assigned Send Sequence Number (SSN) according to the transmission sequence. Also, the length and tag of the message need be recorded alongside in order to rewrite the send statement on demand. In preparation for the transmission, each message will be packed into a flat format with the SSN appended at the end.

For receivers, they need to resolve the SSN after receiving a message, and preserve it into the Highest Sequence number Received (HSR) array representing the latest message received from the sending end. Any message that has a smaller SSN is supposed to have been handled correctly according to our assumption.

Collective payload copy every process counts various collective communications to form the statistics of collectives, and records the operation results on demand. Although every process can be considered as the sender of a collective communication, only when a process exists will its application data be updated by this collective operation, the operation results do require preservation by some particular processes.

3.2 Communicator Maintenance

Since the failure of the communicator will be exposed to the user code in our method, the communicator maintenance mechanism needs to be responsible for the detection of the failure and the restoration of the communicator with the help of the ULFM support.

The communicator could be modified for the purpose of fault tolerance in three cases: process initialization, communicator fault, and the substitution process restart. Thus the maintenance can be divided into three parts, and its work requires the mutual cooperation between processes, as shown in Fig.4.

Communicator maintenance in process initialization for all processes, process initialization is the procedure following the initialization of MPI environment. The maintenance duplicates the communicator from MPI_COMM_WORLD to a globally defined symbol, and attaches our Errorhandler function to the communicator as the default error handling procedure.

Communicator maintenance in Errorhandler the Errorhandler function will be automatically called whenever a process detects the failure on the communicator and returns the error code. In this function, surviving processes revoke the original communicator so that any subsequent operation on the original communicator will eventually fail. Afterwards, they create a new communicator from the revoked one by excluding its failed process. Then the failed processes is discovered by comparing the group of processes in the shrunken communicator with the group of processes in the original communicator. After that, a substitution will be spawned and the inter communicator generated will be merged into an intra-one. Finally the substitution will replace the failure process by reordering the ranks on the new communicator.

Communicator maintenance in Restart segment the Restart segment is a procedure where the substitution operates in collaboration with the surviving processes to merge and reorder the communicator after the MPI environment is initialized. Afterwards, it also attaches our Errorhandler function to the new communicator.

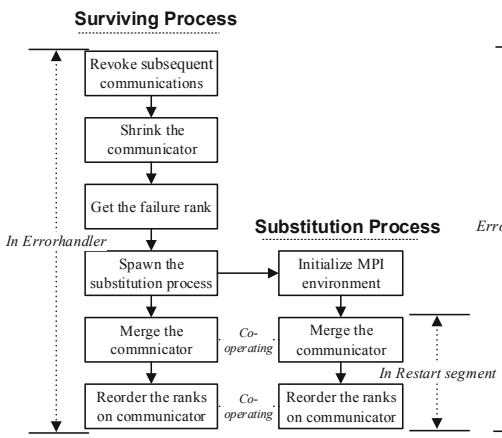

Fig. 4. The flow of the communicator maintenance once a failure occurs

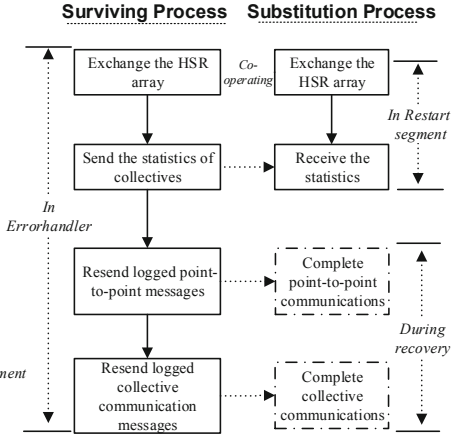

Fig. 5. The flow of the recovery, operations which are dashed may be performed multiple times

3.3 Recovery

Recovery mechanism ensures the consistency of the system by means of operations at user level. There are two key problems that need to be solved: (1) determine which messages need to be re-issued, and (2) guide the substitution process to jump over collective communications correctly during recovery. Therefore the recovery mechanism should exchange the records of reception between processes, and inform the substitution of the number of collective communications which have already been executed. The flow of the recovery mechanism is illustrated in Fig.5.

Recovery mechanism in Errorhandler firstly, each process obtains the Highest Sequence number Received (HSR) arrays which are kept in the other processes' memory, and forms the Highest Sequence number Delivered (HSD) array to determine whether a point-to-point message has been delivered successfully. After that, the logged messages whose SSN is bigger than HSD stored will be resent in order. For the logged collective messages, one of the surviving processes sends the statistics of collective communications to the substitution, Then all the logged collective operation results should be sent successively.

Recovery mechanism in the Restart segment the substitution gets the information it needs (HSR arrays and statistics of collective communication) by the execution of the Restart segment. When entering the recovery procedure, all the point-to-point receptions could be replayed by their original receive statements, and the collective communications will turn into receptions of the logged messages or even empty operations.

3.4 Implementation of User Level Message Logging

User level message logging (ULML) encapsulates the default MPI communication functions (including point-to-point and collective), thereby integrating message

logging fault tolerance capabilities in Open MPI. Each of the ULML MPI functions is an implementation of a particular fault tolerant algorithm, and its goal is to extend the communication with message logging features. ULML does not modify any core Open MPI component or the communication semantics, it calls the default MPI interface functions to perform the actual communications.

In order to implement the ULML in MPI programs, programmers need to follow these steps detailed below: 1. Analyze the communication features of the program, and insert user level checkpoints into the original programs with compiler directive $\#CKPT_i$. The method of choosing the positions of checkpoints has already been discussed in [4]. 2. Replace the original error handler function with our ULML error handler to bring in the communication maintenance and recovery mechanism. 3. Replace the original MPI communication functions with ULFM functions, in order to introduce the payload copy mechanism.

4 Experiments

4.1 Evaluation Methodology

Our computer platform is a subsystem of Tianhe-1A, located in Tianjin, China. Each node of the computer is equipped with two 2.93G Intel Xeon X5670 CPUs (12 cores per node) and 24 GB RAM. The interconnection is the same as described in [4], and the simplex point-to-point bandwidth is 80 Gb/s. All the experiments are executed in Redhat 5.5, and the results presented are mean values over 5 executions of each test.

To investigate application performance we use the NAS Parallel Benchmark suite. The CG benchmark presents heavy point-to-point latency driven communications, while the FT benchmark presents a collective communication pattern by performing all-to-all operations. Thus the class C problem of those benchmarks are tested in order to evaluate the performance of point-to-point payload copy and collective payload copy respectively.

Moreover, we choose the naive pessimistic message logging approach from [2] and the active optimistic message logging protocol (O2P) from [10] as comparative methods at library level.

4.2 Fault Free Performance Evaluation on NAS Benchmarks

In Fig.6, we plot the normalized execution time of CG according to a growing number of processors to evaluate the comparative scalability, the standard execution time of coordinated application-level checkpointing/restart equals 1. Notice that only the performance penalty associated with message logging is presented since no checkpoints and faults are imposed. Fig.6 shows that the performance of the ULML and O2P is comparable, the executions of the two protocols are very similar and exhibit the same scalability with the overhead stays under 5%. Conversely pessimistic approach experiences a severe performance degradation topping at 17% increase in execution time, the increasing point-to-point communication rate (19988 times at 64 cores to 25992 times at 128 cores for example)

greatly affects the overhead induced by pessimistic message logging. But our ULML is immune to this defect by avoiding any bandwidth consumption, except for appending the SSN which has a negligible influence on the message size. Overall, considering its simplicity of implementation, the ULML presents a salutary alternative to refined message logging protocols at library level.

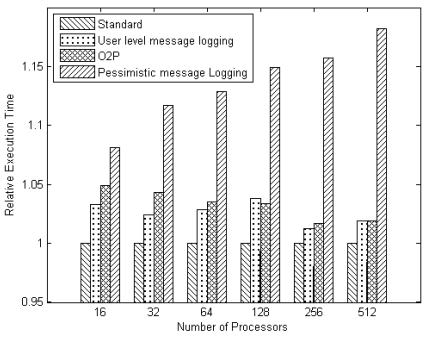

Fig. 6. Scalability of CG Class C

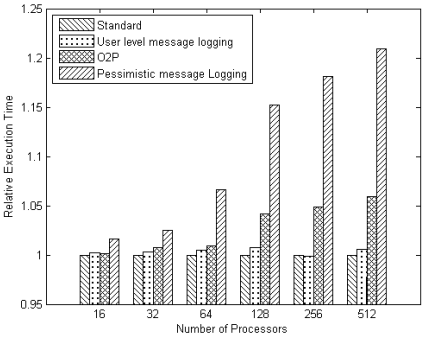

Fig. 7. Scalability of FT Class C

Fig.7 presents the execution time of FT with processor numbers ranging from 16 to 512, normalized to each benchmark with standard execution. For up to 512 cores, the scalability of the proposed message logging approach is excellent since the overhead is solely due to sender-based payload logging. Also, O2P works quite well up to 64 processes, but when the amount of data piggybacked continues to increase because the event logger is overloaded and does not manage to log the determinants in time, O2P eventually suffers from at most a 6% slowdown in our test case. The performance superiority is mainly imputed to the better handling of collectives in the ULML, and the overhead induced by it is very close to the error margin of measurements.

4.3 Recovery Performance Evaluation

We simulate a fault on a processor by sending SIGKILL to a process, CG Class C running on 8 processes is chosen as our test case. First we checkpoint the system at iteration 10, then introduce a failure to process 3 at iteration 70. Table 2 presents the elapsed wall clock time and CPU time consumed to recover the system. We find that the ULML reduces the wall clock time by 26.9% and saving CPU time by 22.1%. Also, different phases of the recovery procedure are timed to measure the factors limiting the speed of our restart protocol. Take process 1 as an example: it spends 0.1253 seconds on communicator maintenance in Errorhandler, and 0.002924 seconds on recovery mechanism in Errorhandler to resend 4741 messages (these two time statistics are not stable, but will not exceed 0.2 seconds). After finishing the Errorhandler segment, process 1 enters a suspended state for 59.761 seconds. Therefore, we believe that our communicator

Table 2. Comparison of recovery time-consumption (Seconds)

	Checkpoint/Restart		User level message logging	
	Failure free	Failure occurred	Failure free	Failure occurred
Wall time	2.78	5.24	2.83	3.83
CPU time	21.33	38.54	22.12	30.01

maintenance and recovery mechanism are lightweight, and the bottleneck is the re-execution of the substitution process.

5 Related Work

Research on message logging has a long history. The seminal paper by Strom and Yemini presented a recovery protocol for distributed systems which permits communication, computation and checkpoint to proceed asynchronously, thus introducing the concept of message logging and causality tracking [13]. Sender-based message logging was introduced by Johson and Zwaenepoel [14], by describing how to secure the correctness of the protocol with the Send Sequence Number and the messages logged in sender's volatile memory. Alvisi and Marzullo presented a classification of the different message logging schemes into three families: pessimistic, optimistic, and causal, according to the different methods of logging reception orders [6]. Recently, Bouteiller used determinism in MPI applications to reduce the number of determinants to log [8]. Guermouche proposed an uncoordinated checkpointing protocol for send-deterministic MPI applications and achieved a satisfying overhead [9]. But all the researches above rely on modifying the MPI library, thus they will face portability issues and induce multiple overhead for collectives in all cases.

The User-Level Failure Mitigation proposal was put forward to improve the resilience of application from programming model in 2012. This proposal allows libraries and applications to increase the fault tolerance capabilities by supporting additional types of failures, and build other desired strategies and consistency models to tolerate faults. The ULFM proposal makes it possible to elevate the message logging layer and guarantee the portability.

6 Conclusion

In this article, we introduce the user level message logging protocol, a new kind of portable fault tolerance method for MPI programs. The new methodology proposed is simple yet effective, particularly suited for collective communication intensive programs. Overall, our work facilitates the adoption of message logging in large-scale scientific computing programs.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No.61303071 and 61120106005, and funds (No.124200011) from Guangzhou Science and Information Technology Bureau.

References

1. Bland, W.: User level failure mitigation in mpi. In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 499–504. Springer, Heidelberg (2013)
2. Meneses, E., Bronevetsky, G., Kale, L.V.: Evaluation of simple causal message logging for large-scale fault tolerant hpc systems. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW 2011, pp. 1533–1540. IEEE Computer Society (2011)
3. Yang, X., Wang, Z., Xue, J., Zhou, Y.: The reliability wall for exascale supercomputing. IEEE Transactions on Computers 61, 767–779 (2012)
4. Xu, X., Yang, X., Lin, Y.: Wbc-alc: A weak blocking coordinated application-level checkpointing for mpi programs. IEICE Transactions, 786–796 (2012)
5. Chakravorty, S., Kale, L.: A fault tolerance protocol with fast fault recovery. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–10 (2007)
6. Alvisi, L., Marzullo, K.: Message logging: Pessimistic, optimistic, causal, and optimal. IEEE Trans. Softw. Eng. 24, 149–159 (1998)
7. Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 51–64. Springer, Heidelberg (2011)
8. Bouteiller, A., Bosilca, G., Dongarra, J.: Redesigning the message logging model for high performance. Concurr. Comput.: Pract. Exper. 22, 2196–2211 (2010)
9. Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F.: Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In: 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS), pp. 989–1000 (2011)
10. Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., Dongarra, J.: Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure recovery. In: IEEE International Conference on Cluster Computing (Cluster 2009), New Orleans, États-Unis, pp. 1–9 (2009)
11. Rabenseifner, R.: Automatic mpi counter profiling of all users: First results on a cray t3e 900-512. In: Proceedings of the Message Passing Interface Developer's and User's Conference(MPIDC 1999), pp. 77–85 (1999)
12. Patarasuk, P., Yuan, X.: Bandwidth efficient allreduce operation on tree topologies. In: IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments, pp. 1–8 (2007)
13. Strom, R., Yemini, S.: Optimistic recovery in distributed systems. ACM Trans. Comput. Syst. 3, 204–226 (1985)
14. Zwaenepoel, W., Johnson, D.: Sender-Based Message Logging. In: Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing, pp. 49–66 (1987)