CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:2196–2211
Published online 27 June 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1589

Redesigning the message logging model for high performance[‡]

Aurelien Bouteiller[*, †], George Bosilca and Jack Dongarra

ICL, University of Tennessee Knoxville, Claxton 1122 Volunteer Boulevard,
Knoxville, TN 37996, U.S.A.

SUMMARY

Over the past decade the number of processors used in high performance computing has increased to hundreds of thousands. As a direct consequence, and while the computational power follows the trend, the mean time between failures (MTBF) has suffered and is now being counted in hours. In order to circumvent this limitation, a number of fault-tolerant algorithms as well as execution environments have been developed using the message passing paradigm. Among them, message logging has been proved to achieve a better overall performance when the MTBF is low, mainly due to a faster failure recovery. However, message logging suffers from a high overhead when no failure occurs. Therefore, in this paper we discuss a refinement of the message logging model intended to improve the failure-free message logging performance. The proposed approach simultaneously removes useless memory copies and reduces the number of logged events. We present the implementation of a pessimistic message logging protocol in Open MPI and compare it with the previous reference implementation MPICH-V2. The results outline a several order of magnitude improvement on the performance and a zero overhead for most messages. Published in 2010 by John Wiley & Sons, Ltd.

Received 8 January 2010; Revised 21 January 2010; Accepted 15 February 2010

KEY WORDS: high performance computing; fault tolerance; message logging; uncoordinated checkpoint

1. INTRODUCTION

The Top500 list, the list of the 500 most powerful supercomputers in the world, highlights a constant increase in the number of processors. An attentive reading shows that the top 10 of these supercomputers contain several thousands of processors each. Despite a more careful design of the components in these supercomputers, the increase in the number of components directly affects the reliability of these systems. Moreover, the remaining systems in the Top500 list are built using

[*] Correspondence to: Aurelien Bouteiller, ICL, University of Tennessee Knoxville, Claxton 1122 Volunteer Boulevard, Knoxville, TN 37996, U.S.A.
[†] E-mail: bouteill@eecs.utk.edu
[‡] This article is a U.S. Government work and is in the public domain in the U.S.A.

commodity components, which greatly affects their reliability. While the Mean Time Between Failures (MTBF) on the BlueGene/L [1] is counted in days, the commodity clusters exhibit a usual MTBF of tens of hours. With a further expected increase in the number of processors in the next generation supercomputers, one might deduce that the probability of failures will continue to increase, leading to a drastic decrease in reliability.

Fault-tolerant algorithms have a long history of research. Only recently, since the practical issue has been raised, High Performance Computing (HPC) software has been adapted to deal with failures. As most HPC applications are using the Message Passing Interface (MPI) [2] to manage data transfers, introducing failure recovery features inside the MPI library automatically benefits a large range of applications. One of the most popular automatic fault-tolerant technique, coordinating checkpoint, builds a consistent recovery set [3,4]. As today's HPC users are facing occasional failures, they have not suffered from the slow recovery procedure, involving restarting all the computing nodes even when only one has failed. Considering that future systems will endure higher fault frequencies, the recovery time could become another gap between the peak performance of the architecture and the effective performance that users can actually harvest from the system.

Because message logging does not rely on such coordination, it is able to recover faster from failures. From previous results [3], it is expected that a typical application makespan will be better than coordinated checkpoint when the MTBF is less than 9 h whereas coordinated checkpoint will not be able to progress anymore for an MTBF less than 3 h. Yet, message logging suffers from a high overhead on the communication performance. Moreover, the better the latency and bandwidth offered by newer high performance networks, the higher the relative overhead. These drawbacks need to be addressed to provide a resilient and fast fault-tolerant MPI library to the HPC community. In this paper we propose a refinement of the classical model of message logging, closer to the reality of high performance network interface cards, where message receptions are decomposed in multiple dependent events. We better categorize message events allowing (i) the suppression of intermediate message copies on high performance networks and (ii) the identification of deterministic and non-deterministic events, thus reducing the overall number of messages requiring latency disturbing management. We demonstrate how this refinement can be used to reduce the fault-free overhead of message logging protocols by implementing it in Open MPI [5]. Its performance is compared with the previous reference implementation of message logging MPICH-V2. The results outline a several orders of magnitude improvement of the fault-free performance of pessimistic message logging and a drastic reduction in the overall number of logged events.

The remainder of this paper is organized as follows: in the following section we recall classical message logging and then depict the modifications we introduce to better fit HPC networks. In the third section we depict the implementation issues of the prototype in Open MPI. The fourth section presents experimental evaluation, followed by related work and the conclusion.

2. MESSAGE LOGGING

2.1. Classical message logging

Message logging is usually depicted using the more general model of message passing distributed systems. Communications between processes are considered explicit: processes explicitly request sending and receiving messages, and a message is considered as delivered only when the receive operation associated with the data movement completes. Additionally, from the perspective of the application each communication channel is FIFO, but there is no particular order on messages traveling along different channels. The execution model is pseudo-synchronous: there is no global shared clock among processes but there is some (potentially unknown) maximum propagation delay of messages in the network. An intuitive interpretation is to say that the system is asynchronous and there is some *eventually reliable* failure detector.

Failures can affect both the processes and the network. Usually, network failures are managed by some CRC and message reemission provided by the hardware or low-level software stack and do not need to be considered in the model. We consider that processes endure definitive crash failures, where a failed process stops sending any message.

Events. Each computational or communication step of a process is an event. An execution is an alternate sequence of events and process states, with the effect of an event on the preceding state leading the process to the new state. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes. However, Lamport defines a causal partial ordering between events with the *happened before* relationship [6]. It is noted that $e \prec f$ when event f is causally influenced by e.

These events can be classified into two categories: deterministic and non-deterministic. An event is deterministic when, from the current state, there is only one outcome state for this event. On the contrary, if an event can result in several different states depending on its outcome, then it is non-deterministic. Examples of deterministic events are internal computations and message emissions, which follow the code-flow. Examples of non-deterministic events are message receptions, which depend on time constraints of message deliveries.

Checkpoints and inconsistent states. Checkpoints (i.e. complete images of the process memory space) are used to recover from failures. The recovery line is the configuration of the application after some processes have been reloaded from checkpoints. Unfortunately, checkpointing a distributed application is not as simple as storing each single process image without any coordination, as illustrated by the example execution of Figure 1. When process P_1 fails, it rolls back to

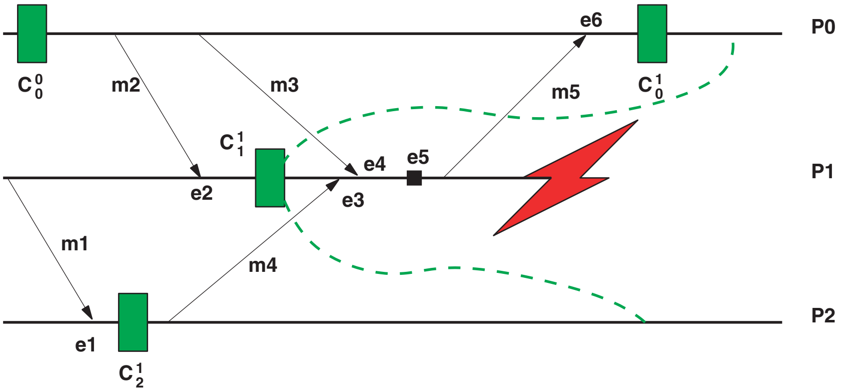

Figure 1. Example execution of a distributed system with checkpoints and inconsistent recovery line.

DOI: 10.1002/cpe

checkpoint C_1^1. Messages from the past crossing the recovery line (m_3, m_4) are *in-transit* messages: the restarted process will request their reception whereas the source process never sends them again, thus it is needed to save these messages. Messages from the future crossing the recovery line (m_5) are *orphan*: following the Lamport relationship, current state of P_0 depends on reception of m_5 and by transitivity on any event that occurred on P_1 since C_1^1 (e_3, e_4, e_5). As the channels between P_0 and P_1 and between P_2 and P_1 are asynchronous, the reception of m_3 and m_4 could occur in a different order during re-execution, leading to a recovered state of $P1$ that diverges from the initial execution. As the current state of P_0 depends on states that P_1 could never reach anymore, the overall state of the parallel application after the recovery could be inconsistent. Checkpoints leading to an inconsistent state are useless and must be discarded; in the worst case, all checkpoints are useless and the computation may have to be restarted from the beginning.

Recovery. In event logging, processes are considered as *Piecewise deterministic*: only sparse non-deterministic events occur, separating large parts of deterministic computation. Considering that non-deterministic events are committed during the initial execution into some *safe* repository, a recovering process is able to replay exactly the same order for all non-deterministic events. Therefore, it is able to reach exactly the same state as before the failure. Furthermore, message logging considers the network as the only source of non-determinism and only logs the relative ordering of messages from different senders (e_3, e_4 in Figure 1). The sufficient condition ensuring a successful recovery requires that a process must never depend on an unlogged event from another process. As the only way to create a dependency between processes is to send a message, all non-deterministic events occurring between two consecutive sends can be merged and committed together.

Event logging only saves events in the remote repository, without storing the message payload. However, when a process is recovering, it needs to replay any reception that happened between the last checkpoint and the failure and therefore requires the payload of those messages (m_3, m_4 in Figure 1). During normal operation, every outgoing message is saved in the sender's volatile memory: a mechanism called sender-based message logging. This allows for the surviving processes to serve past messages to recovering processes on demand, without rolling back. Unlike events, sender-based data do not require stable or synchronous storage. Should a process holding useful sender-based data crash, the recovery procedure of this process replays every outgoing send and thus rebuilds the missing messages.

2.2. Non-blocking communications

To reach top performance, the MPI standard defines a more sophisticated set of communication routines than simple blocking send and receive. One of the most important optimizations for a high throughput communication library is *zero copy*, the ability to send and receive directly in the application's user-space buffer without intermediary memory copies. Figure 2 shows the basic steps of non-blocking zero-copy communications in MPI. First the application *requests* a message to be received, specifying the message source, tag, and reception buffer. When a message arrives from the network, the source and the tag are compared to the pending requests. If the message does not match any pending request it is copied in a temporary buffer (m_1 is tagged $tagB$ and all pending requests wants $tagA$) until a matching request is posted by the application. If the message matches a pending request, such as m_3 and m_4, it is directly written in the receive buffer without intermediate copy. Because requests can be ANY_SOURCE the result of the matching may depend on the order of

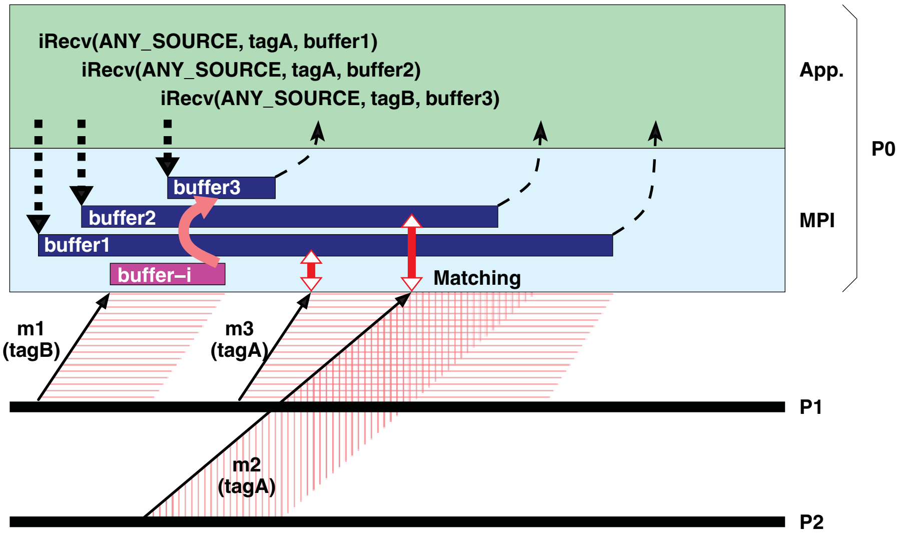
iRecv(ANY_SOURCE, tagA, buffer1)
iRecv(ANY_SOURCE, tagA, buffer2)
iRecv(ANY_SOURCE, tagB, buffer3)
App.
P0
MPI
buffer3
buffer2
buffer1
buffer–i
Matching
m1
(tagB)
m3
(tagA)
P1
m2
(tagA)
P2

Figure 2. Steps in a zero-copy MPI receive operation.

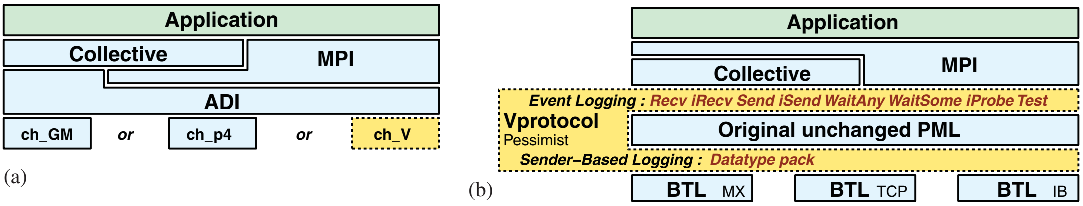

Figure 3. Comparison between the MPICH and the Open MPI architecture and the interposition level of fault tolerance (fault-tolerant components are dashed): (a) MPICH-1.2/MPICH-V and (b) Open MPI/Open MPI-V.

reception of the first fragment of the message (if m_2 had arrived earlier it would have been delivered in $buffer_1$). The next step for the application might be to probe for message delivery readiness. The result of those probe functions may depend on the message transfer termination time, but is not related to the matching order (m_3 is matched first but lasts longer than m_2).

Because the classical message logging model assumes the message reception is a single atomic event, it cannot catch the complexity of zero-copy MPI communications involving distinct matching and delivery events. As an example in MPICH, only the lowest blocking point-to-point transport layer called the *device* matches the classical model, explaining why previous state-of-the-art message logging implementations, such as MPICH-V, replace the low-level device with the ch_v fault-tolerant one (see Figure 3(a)). This device has adequate properties regarding the hypothesis of message logging: (i) messages are delivered in one single atomic step to the application (though message interleave is allowed inside the ch_v device), (ii) intermediate copies are made for every message to fulfill this atomic delivery requirement, and the matching needs to be done at delivery

DOI: 10.1002/cpe

time, (iii) as the message logging mechanism replaces the regular low-level device, it cannot easily benefit from zero copy and OS bypass features of modern network cards, and (4) because it is not possible to distinguish the deterministic events at this software level, every message generates an event (which is obviously useless for deterministic receptions). We will show in the performance analysis section how these strong model requirements lead to dramatical performance overhead in an MPI implementation when considering high performance interconnects.

2.3. Refinements for zero-copy messages

By relaxing the strong model described previously, it is possible to interpose the event logging mechanism inside the MPI library. Then it is only necessary to log the communication events at the library level and the expensive events generated by the lower network layer can be completely ignored. This requires consideration of the particularity of the internal MPI library events, but allows to use the optimized network layers provided by the implementation. The remainder of this section describes this improved model.

Network events. From the lower layer comes the packet-related events: let m denote a message transferred in $length(m)$ network packets. We note that r_m^i equals the ith packet of message m, where $1 \leq i \leq length(m)$. Because the network is considered reliable and FIFO, we have $\forall 1 \leq i \leq length(m)-1$, $r_m^i \prec r_m^{i+1}$. We denote $tag(m)$ the tag of message m and $src(m)$ its emitter. Packets are received atomically from the network layer.

Application events. From the upper layer comes the application-related events. We note that $Post(tag, source)$ is a reception post, $Probe(tag, source)$ is the event of checking the presence of a message, and $Wait(n, \{R\})$ is the event of waiting n completions of the request identifier set $\{R\}$. Because the application is considered piecewise deterministic, we can assign a totally ordered sequence of identifiers to upper layer events. Let r_0 be a request identifier obtained by the $Post_0(tag_0, source_0)$ event. As posting is the only way to obtain a request identifier, if $r_0 \in \{R\}$, $Post_0(tag_0, source_0) \prec Wait_0(n, \{R\})$. There is at most one event Post per message and at least one Wait event per message. If $r_{m_0}^1 \prec Probe_0(tag_0, source_0) \prec Post_0(tag_0, source_0)$, then $Probe_0(tag_0, source_0)$ must return true. Otherwise, it must return false. The main difference between Probe and Post is that in case $r_{m_0}^1$ precedes one of these events, $Probe_0(tag_0, source_0)$ will not discard $r_{m_0}^1$, whereas $Post_0(tag_0, source_0)$ will always do so.

Library events. The library events are the result of the combination of a network-layer event and an application-layer event. There are two categories of library events: (i) Matching (denoted by M) and (2) Completing (denoted by C). Matching binds a network communication with a message reception request; Completing checks the internal state of the communication library to determine the state of a message (completed or not).

(1) To build a Matching event from a reception event and a Post event, we define a reception-matching pair of events: r_m^1 and $Post_0(tag_0, source_0)$ match for reception if and only if $(source_0 = src(m) \vee source_0 = ANY) \wedge (tag_0 = tag(m) \vee tag_0 = ANY)$. The Matching event built from the reception-matching events is causally dependent on the two elements of the matching pair: $Post_0(tag_0, source_0) \prec M_0$ and $r_m^1 \prec M_0$. The reception-matching pair is determinist if and only if $source_0 \neq ANY$. Additionally, based on the same rules, we can build a Matching from a Probe event and a reception event. In this case, the result of the Matching M_0

is successful if and only if $r_m^1 \prec Probe_0(tag_0, source_0)$. Otherwise, the Matching event takes a special value (undefined source). Because the order between r_m^1 and $Probe_0(tag_0, source_0)$ is non-deterministic, all probe-matching pair events are non-deterministic.

(2) Similarly, to build a Completing event from a reception event and a Wait event, we define a completion-matching pair of events: $r_m^{length(m)}$ and $Wait(n, \{R\})$ match for completion if and only if there is a matching event M_0 built from r_m^1 containing the request identifier r_0 and $r_0 \in \{R\}$. The Completing event built from the completion-matching events is causally dependent on the two elements of the matching pair: $Wait(n, \{R\}) \prec C_0$ and $r_m^{length(m)} \prec C_0$. All the r_m^i events are non-deterministic per definition. Thus, every $Waitn, \{R\}$ event is non-deterministic, because the result of these events depends upon the internal state of the library, which depends upon the $r_m^{length(m)}$ events. However, according to the matching and completion rules, if $r_m^{length(m)}$ and $Wait(n, \{R\})$ is a completion-matching pair, the Completing event built is deterministic if and only if $n_0 = |R_0|$ (case of Wait, WaitAll, Recv).

Although the refinement introduces many new events, most of them are not necessarily logged, since they are deterministic. Only non-deterministic events (non-deterministic Matching due to ANY sources; non-deterministic Matching due to probe-matching events; non-deterministic completion due to WaitSome, WaitAny, TestAll, Test, TestAny, and TestSome) are logged and introduce a synchronization with the event logger.

3. IMPLEMENTATION IN OPEN MPI

3.1. Generic fault-tolerant layer

The Open MPI architecture is a typical example of the new generation MPI implementations. Figure 3(b) summarizes the Open MPI software stack dedicated to MPI communications. Regular components are summarized with plain lines, whereas the new fault-tolerant components are dashed. At the lowest level, the BTL exposes a set of communication primitives appropriate for both send/receive and RDMA interfaces. A BTL is MPI semantics agnostic; it simply moves a sequence of bytes (potentially non-contiguous) across the underlying transport. Multiple BTLs might be in use at the same time to strip data across multiple networks. The PML implements all logic for point-to-point MPI semantics, including standard, buffered, ready, and synchronous communication modes. MPI message transfers are scheduled by the PML based on a specific policy according to short and long protocols, as well as using control messages (ACK/NACK/MATCH). Additionally, the PML is in charge of providing the MPI matching logic as well as reordering the out-of-order fragments. All remaining MPI functions, including some collective communications, are built on top of the PML interface. While in the current implementation of the fault-tolerant components only point-to-point based collectives are supported, we plan to support, in the near future, other forms of collective communication implementations (such as hardware-based collectives).

In order to integrate the fault-tolerance capabilities in Open MPI, we added one new class of components, the `Vprotocol` (dashed in Figure 3(b)). A `Vprotocol` component is a parasite enveloping the default PML. Each is an implementation of a particular fault-tolerant algorithm; its goal is not to manage actual communications but to extend the PML with message logging features. As all of the Open MPI components, the `Vprotocol` module is loaded at runtime on user's request and replaces some of the interface functions of the PML with its own. Once it has logged or modified the communication requests according to the needs of the fault-tolerant algorithm, it calls the real PML to perform the actual communications. This modular design has several advantages compared to the MPICH-V architecture: (i) it does not modify any core Open MPI component, regular PML message scheduling and device optimized BTL can be used, (ii) expressing a particular fault-tolerant protocol is easy, it is only focused on reacting to some events, not handling communications, and (iii) the best suited fault-tolerant component can be selected at run time.

3.2. Pessimistic message logging implementation

The `Vprotocol` pessimist is the implementation based on our refined model. It provides four main functionalities: sender-based message logging, remote event storage, any source reception event logging, and non-deterministic delivery event logging. Each process has a local Lamport clock, used to mark events; during Send, iSend, Recv, iRecv, and Start, every request receives the clock stamp as a unique identifier.

Sender-based logging. The improvements we propose to the original model still rely on a sender-based message payload logging mechanism. We integrated the sender-based logging to the data-type engine of Open MPI. The data-type engine is in charge of packing (maybe non-contiguous) data into a flat format suitable for the receiver's architecture. Each time a fragment of the message is packed, we copy the resulting data in a *mmaped* memory segment. Because the sender-based copy progresses at the same speed as the network, it benefits from cache reuse and releases the send buffer at the same date. Data are then asynchronously written from memory to disk in background to decrease the memory footprint.

Event logger commits. Non-deterministic events are sent to event loggers processes (EL). An EL is a special process added to the application outside the MPI_COMM_WORLD; several might be used simultaneously to improve scalability. Events are transmitted using non-blocking MPI communications over an inter-communicator between the application process and the event logger. Although asynchronous, there is a transactional acknowledgement protocol to ensure that every event is safely logged before any MPI send can progress.

Any source receptions. Any source logging is managed in the iRecv, Recv and Start functions. Each time any source receive is posted, the completion function of the corresponding request is modified. When the request is completed, the completion callback logs the event containing the request identifier and the matched source. During recovery, the first step is to retrieve the events related to the MPI process from the event logger. Then every promiscuous source is replaced by the well-specified source of the event corresponding to the request identifier. Because channels are FIFO, enforcing the source is enough to replay the original matching order.

Non-deterministic deliveries. Non-deterministic deliveries (NDD) are the iProbe, WaitSome, WaitAny, Test, TestAll, TestSome, and TestAny functions. The Lamport clock is used to assign a unique identifier to every NDD operation. When an NDD ends, a new delivery event is created, containing the clock and the list of all the completed request identifiers. During replay, when the NDD clock is equal to the clock of the first event, the corresponding requests are completed by waiting for each of them.

It sometimes happens that no request is completed during an NDD. To avoid creating a large number of events for consecutive unsuccessful NDD, we use lazy logging; only one event is created for all the consecutively failed NDD. If a send operation occurs, any pending NDD have to be flushed to the EL. If an NDD succeeds, any pending lazy NDD is discarded. During recovery, the NDD whose clock is lower than the first NDD event in the log has to return no request completed.

4. PERFORMANCES

4.1. Experimental testbed

The experimental testbed includes two clusters. In the first cluster, each node is a dual Opteron 246 (2 GHz) with 2 GB DDR400 memory and a Myrinet 2000 PCI-E interconnect with a 16 ports Myrinet switch; this machine is used only for Myrinet 2000 experiments. In the second cluster, each node is a dual Xeon Woodcrest with 4 GB DDR5300 memory. Two different networks are available on this machine, a Gigabit Ethernet and a Myrinet 10 G interconnect with a 128 ports Myrinet switch. The software setup is Linux 2.6.18 using MX 1.2.0j. Benchmarks are compiled using gcc and gfortran 4.2.1 with the -O3 flag. We used NetPIPE [7] to perform ping-pong tests, whereas the NAS Parallel Benchmarks 3.2.1 (NPB) and High Performance Linpack (HPL) are used to investigate the application behavior. Applications are deployed with one process per node; HPL uses threaded GotoBLAS to make full use of the available cores. Because the proposed approach does not change the recovery strategy used in previous works, we only focus on the failure-free performance.

4.2. Benefits from event distinction

One of the main differences of the refined model is the split of message receptions into two distinct events. In the worst case, this might lead to logging twice as many events compared to the model used in other message logging implementations. However, better fitness between model and MPI internals allows for detecting (and discarding) deterministic events. Table I characterizes the amount of non-deterministic (actually logged in Open MPI-V) events compared to the overall number of exchanged messages. Although we investigated all the NPB kernels (BT, MG, SP, LU, CG, FT) to cover the widest spectrum of application patterns, we detected non-deterministic events in LU and MG only. In all other benchmarks, Open MPI-V does not log any event, thanks to the detection

Table I. Percentage of non-deterministic events to the total number of exchanged messages on the NAS Parallel Benchmarks (Class B).

	BT	SP	LU					
#Processors	All		4	32	64	256	512	1024
%Non-deterministic	0	0	1.13	0.66	0.80	0.80	0.75	0.57
	FT	CG	MG					
#Processors	All		4	32	64	256	512	1024
%Non-deterministic	0	0	40.33	29.35	27.10	22.23	20.67	19.99

of deterministic messages. On both MG and LU, the only non-deterministic events are any-source messages; there is no non-deterministic deliveries or probes. In MG, two-thirds of the messages are deterministic, whereas in LU less than 1% use the any-source flag, outlining how the better fitting model drastically decreases the overall number of logged events in the most usual application patterns. As a comparison, MPICH-V2 logs at least one event for each message (and two for rendezvous messages). According to our experiments, the same results hold for classes A, C, and D of the NAS. The ratio of logged events does not correlate with the number of computing processes in LU and decreases when more processes are used in MG, meaning that the fault-tolerant version of the application is at least as scalable as the original one.

Avoiding logging of some events is expected to lower the latency cost of a pessimistic protocol. Figure 4 presents the overhead on Myrinet round trip time of enabling the pessimistic fault-tolerant algorithm. We normalize Open MPI-V pessimist (labeled Open MPI-B with sender-based in the figure) according to a similar non-fault-tolerant version of Open MPI, whereas we normalize the reference message logging implementation MPICH-V2 according to a similar version of MPICH-MX; in other words, 100% is the performance of the respective non-fault-tolerant MPI library. We deem this as reasonable as (i) the bare performance of Open MPI and MPICH-MX are close enough that using a different normalization base introduces no significant bias on the comparison between fault-tolerant protocols and (ii) this ratio reflects the exact cost of fault tolerance compared to a similar non-fault-tolerant MPI implementation, which is exactly what needs to be outlined. The IPoMX performance is also provided as a reference only to break down MPICH-V2 overhead.

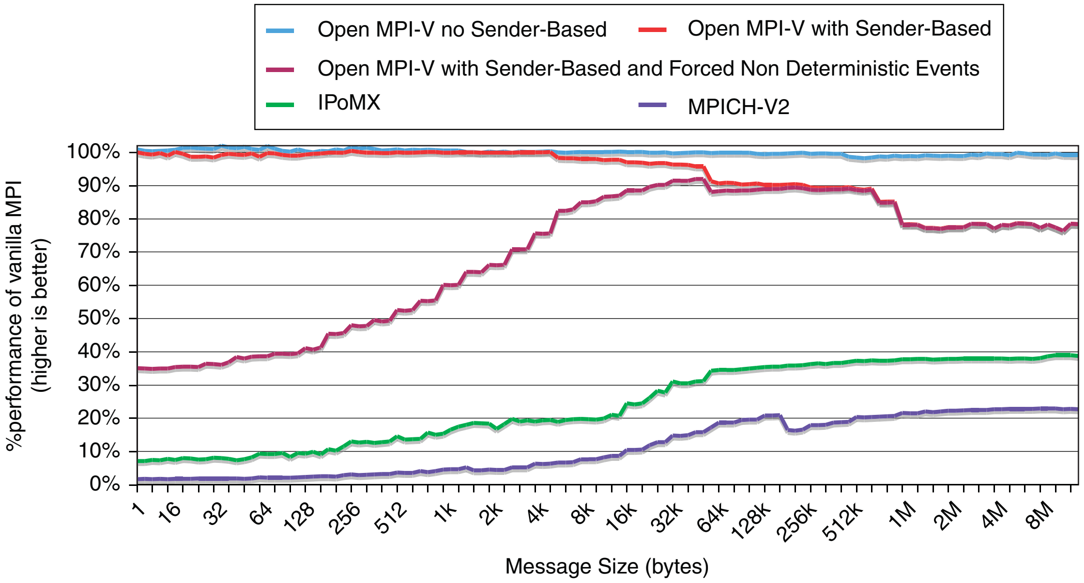

Figure 4. Myrinet 2000 ping-pong performance of pessimistic message logging in percentage of non-fault-tolerant MPI library.

In this ping-pong test, all Recv operations are well-specified sources and there is no WaitAny. As a consequence, Open MPI-V pessimist does not create any event during the benchmark and reaches exactly the same latency as Open MPI (3.79 μs). To measure the specific cost of handling non-deterministic events in Open MPI-V pessimist, we modified the NetPIPE benchmark code; every Recv has been replaced by the sequence of any source iRecv and a WaitAny. This altered code generates two non-deterministic events for each message. The impact on Open MPI-V pessimist latency is a nearly three-time increase in the latency. The two events are merged into a single message to the event logger; the next send is delayed until the acknowledge comes back. This is the expected cost on the latency of pessimistic message logging. Yet, the better detection of non-deterministic events removes the message logging cost for some common types of messages.

Because MPICH-V does not discard deterministic events from logging, there is a specific overhead (40 μs latency increase to reach 183 μs) for every message, even on the original deterministic benchmark. This specific overhead comes on top of those from memory copies.

4.3. Benefits from zero-copy receptions

Figure 4 shows the overhead of MPICH-V. With the pessimistic protocol enabled, MPICH-V reaches only 22% of the MPICH-MX bandwidth. This bandwidth reduction is caused by the number of memory copies in the critical path of messages. Because the message logging model used in MPICH-V assumes that delivery is atomic, it cannot accommodate the MPI matching and buffering logic; therefore, it does not fit the intermediate layer of MPICH (similar to the PML layer of Open MPI). As a consequence, the event logging mechanism of MPICH-V replaces the low level *ch_mx* with a TCP/IP-based device. The cost of memory copies introduced by this requirement is estimated by considering the performance of the NetPipe TCP benchmark on the IP emulation layer of MX: IPoMX. The cost of using TCP, with its internal copies and flow control protocol, is as high as 60% of the bandwidth and increases the latency from 3.16 to 44.2 μs. In addition, the `ch_v` device itself needs to make an intermediate copy on the receiver to delay matching until the message is ready to be delivered. This is accountable for the 20% remaining overhead on the bandwidth and increases the latency to 96.1 μs, even without enabling event logging.

On the contrary, in Open MPI-V the model fits tightly with the behavior of MPI communications. The only memory copy comes from the sender-based message payload logging; there are no other memory copies. As a consequence, Open MPI-V is able to reach a typical bandwidth as high as 1570 Mbit/s (compared to 1870 Mbit/s for base Open MPI and 1825 Mbit/s for MPICH-MX). The difference between Open MPI-V with or without sender-based logging highlights the benefits of our cache reuse approach. While the sender-based copy fits in cache, the performance overhead of the extra copy is reduced to 11% and jumps to 28% for messages larger than 512 kB.

4.4. Sender-based impact

While the overall number of memory copies has been greatly reduced, the sender-based message payload copy is mandatory and cannot be avoided. Figure 5 explains the source of this overhead by comparing the performance of Open MPI and Open MPI-V pessimist on different networks. As the sender-based copy is not on the critical path of messages, there is no increase in latency, regardless of the network type. On Ethernet, the bandwidth is unchanged as well, because the time to send

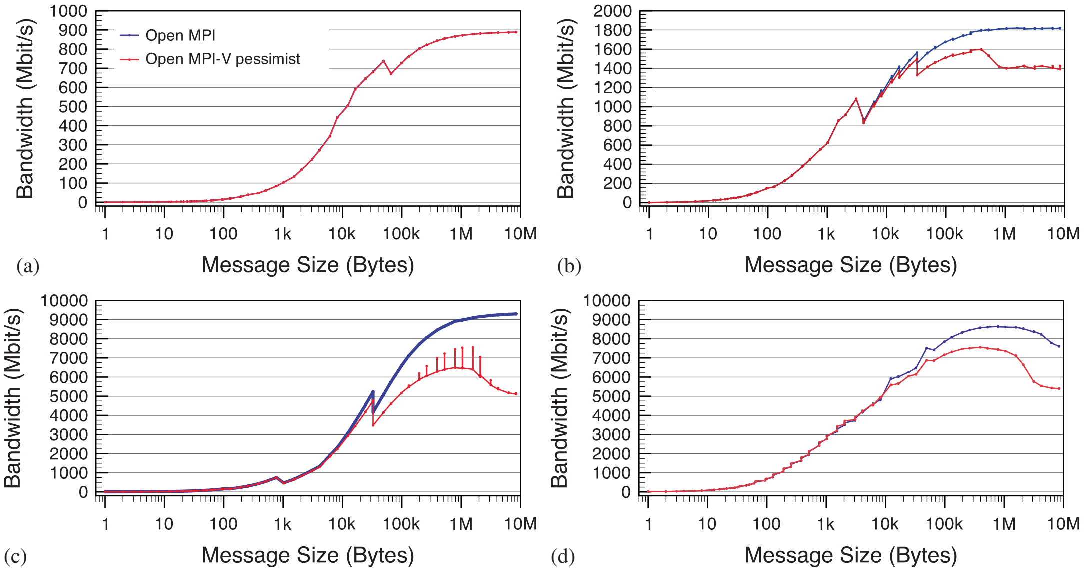

Figure 5. Ping-pong performance comparison between Open MPI and Open MPI-V pessimist on various networks: (a) TCP Gigabit Ethernet; (b) MX Myrinet 2 G; (c) MX Myrinet 10 G; and (d) Shared Memory.

the message on the wire is much larger than the time to perform the memory copy, thus a perfect overlap.

Counter-intuitively, Open MPI bandwidth for the non-fault-tolerant version is better on Myrinet 10 G than on shared memory: the shared memory device uses a copy-in copy-out mechanism between processes, producing one extra memory access for each message (i.e. physically reducing the available bandwidth by two). Adding a third memory copy for handling sender-based logging to the two involved in regular shared memory transfer has up to 30% impact on the bandwidth for large messages, even when this copy is asynchronous. This is the expected result considering that the performance bottleneck for the shared memory network is the pressure on the memory bus bandwidth.

As the sender-based message logging speed depends on memory bandwidth, the faster the network, the higher the relative copy time becomes. Myrinet 2 G already exhibits imperfect overlap between memory copies and network transmission, although when the message transfer fits in cache, the overhead is reduced by the memory reuse pattern of the sender-based mechanism. With the faster Myrinet 10 G, the performance gap widens to 4.2 Gbit/s (44% overhead). As the pressure on the memory subsystem is lower when using Myrinet 10 G network than when using shared memory, one could expect sender-based copy to be less expensive in this context. However the comparison between Open MPI-V on Myrinet 10 G and shared memory shows a similar maximum performance on both media, suggesting that some memory bandwidth is still available for improvements from better software engineering. Similarly, the presence of performance spikes for message sizes between 512 kB and 2 MB indicates that the cache reuse strategy does not fit well with the DMA mechanism used by this NIC.

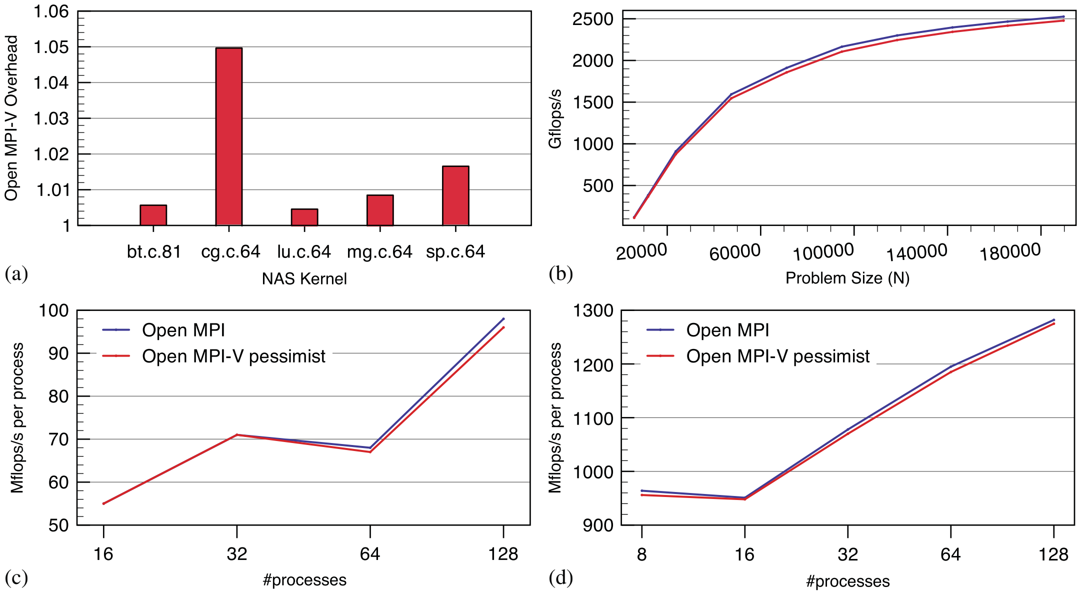

Figure 6. Application behavior comparison between Open MPI and Open MPI-V pessimist on Myrinet 10 G: (a) NAS normalized performance (Open MPI = 1); (b) Weak scalability of HPL (90 procs, 360 cores); (c) Scalability of CG Class D; and (d) Scalability of LU Class D.

4.5. Application performance and scalability

Figure 6(a) presents the performance overheads of various numerical kernels on a Myrinet 10 G network with 64 nodes. Interestingly, the two benchmarks exhibiting non-deterministic events suffer from a mere 1% overhead compared to a non-fault-tolerant run. The more synchronous CG shows the highest performance degradation, topping at only a 5% increase in the execution time. Because there are no non-deterministic events in CG, the overhead is solely due to sender-based payload logging.

Figure 6(b) compares the performance of a fault-tolerant run of HPL with regular Open MPI on 90 quad core processors connected through Myrinet 10 G, one thread per core. While the performance overhead is limited, it is independent of the problem size. Similarly, for CG and LU (Figures 6(c) and (d)), the scalability when the number of processes increase follows exactly the same trend for Open MPI and Open MPI-V. For up to 128 nodes, the scalability of the proposed message logging approach is excellent, regardless of the use of non-deterministic events by the application.

5. RELATED WORKS

Fault tolerance can be managed fully by the application [8,9]. However, software engineering costs can be greatly decreased by integrating fault-tolerant mechanisms at the communication middleware level. FT-MPI [10,11] aims at helping an application to express its failure recovery policy by

DOI: 10.1002/cpe

taking care of rebuilding internal MPI data structures (communicators, rank, etc.) and triggering user provided callbacks to restore a coherent application state when failure occurs. Although this approach is very efficient to minimize the cost of failure recovery techniques, it adds a significant level of complexity to the design of the application code.

Automatic fault-tolerant MPI libraries totally hide failures from the application, thus avoiding any modification of the user's code. A good review of the various techniques used to automatically ensure the successful recovery of distributed applications from a checkpoint set is provided by [12].

Consistent recovery can be achieved automatically by building a coordinated checkpoint set where there exists no orphan message (with the Chandy & Lamport algorithm [13], CIC [14] or blocking the application until channels are empty). The blocking checkpointing approach has been used in LAM/MPI [4] whereas the Chandy & Lamport algorithm has been used in CoCheck [15], MPICH-Vcl [16], and D*éjà*-vu. In all coordinated checkpoint techniques, the only consistent recovery line is when every process, including non-failed ones, restarts from checkpoint. The message logging model we propose does not have this requirement, which according to [3] allows for faster recovery.

Another way to ensure automatic consistent recovery is to use message logging. Manetho [17], Egida [18], and MPICH-V [19] use several flavors of message logging (optimistic, pessimistic, and causal). They rely on the classical message logging model, and as a consequence, they are hooked into the lowest MPI level. Compared to our current work, they cannot distinguish between deterministic and non-deterministic events and they introduce some extra memory copies leading to a performance penalty on the recent high throughput networks.

6. CONCLUSION

In this paper we introduced a refinement of the message logging model intended to reduce the raw overhead of this fault-tolerant protocol. Unlike the classical model, it does not consider the message delivery as a single atomic step. Instead, a message may generate two kinds of events: matching events at the beginning of any source receptions and deliver events to count the number of times that a message has been involved in a non-deterministic probe before delivery. The advantages of this model are (i) better fitting the actual MPI communication pattern, (ii) removing the need for an intermediate copy of each message, and (iii) allowing implementation of fault-tolerant mechanisms at a higher level of the software hierarchy and then distinguishing between non-deterministic and deterministic events.

We implemented a pessimistic message logging algorithm according to this model in Open MPI and compared its performance to the previous reference implementation of pessimistic message logging MPICH-V2. The results outline a drastically lower cost of the fault-tolerant framework. Thanks to the removal of intermediate message copies, Open MPI-V latency is 10.5 times better than MPICH-V2 in the worst case, whereas bandwidth is multiplied by 4. Furthermore, because of the better detection of deterministic events, most common types of messages do not have any message logging overhead, leading to a 35 times better latency.

As a consequence, uncoordinated checkpointing results in less than 5% overhead on the application performance, whereas scalability both in terms of number of nodes and data volume is close to perfect. This is a major improvement compared to previous uncoordinated fault-tolerant approaches.

Future works

A direct consequence of our study is to try to eliminate the remaining cost from sender-based message logging. We plan to improve the pipelining between the fragment emission into the network and the sender-based copy to increase overlap and improve the cache reuse. Although it is impossible to reduce sender-based overhead on shared memory, the failure of a single core usually strongly correlates with the breakdown of the entire computing node. Therefore, coordinating checkpoint inside the same node and disabling intra-node sender-based copy could totally remove this cost while retaining the better scalability of the uncoordinated approach from a node perspective.

Next, we plan to better characterize the source of non-deterministic events. Those events may be generated either by the numerical algorithm itself, using any source receptions or non-deterministic probes, or by the implementation of collective communications over point-to-point inside the MPI library. In this second case, some collaborative mechanisms may be involved to better reduce the cost of semantically determinist messages. With deeper modifications, we could even envision completely disabling logging during collective communication by adding some collective global success notification, which would allow for replaying collectives as a whole instead of individual point-to-point messages.

One of the weaknesses of message logging, when compared to coordinated checkpoint, is a higher failure-free overhead. Because it has been greatly improved by our work, the relative performance ordering of those two protocols could have changed. We plan next to make a comprehensive comparison between improved message logging and coordinated checkpoint, in terms of failure free overhead, recovery speed, and resiliency.

Finally, we could investigate the consequence of using application threads on the piecewise deterministic assumption, a scenario which will be more common with the dominance of multicore processors. Because of a different ordering of MPI operations, the unique request identifier and the probe clock may vary during recovery. Some mechanisms may be designed to ensure that the same identifiers are assigned during replay of threaded applications.

REFERENCES

1. The IBM LLNL BlueGene/L Team. An overview of the BlueGene/L supercomputer. *Supercomputing '02*: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 2002; 1–22.
2. The MPI Forum. MPI: A message passing interface. *Supercomputing '93*: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. ACM Press: New York, NY, U.S.A., 1993; 878–883.
3. Lemarinier P, Bouteiller A, Herault T, Krawezik G, Cappello F. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. *IEEE International Conference on Cluster Computing* (*Cluster 2004*). IEEE Computer Society Press: Silver Spring, MD, 2004.
4. Sankaran S, Squyres JM, Barrett B, Lumsdaine A, Duell J, Hargrove P, Roman E. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *Proceedings*, *LACSI Symposium*, Sante Fe, NM, U.S.A., October 2003.
5. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings*, *11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004; 97–104.
6. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
7. Snell Q, Mikler A, Gustafson J. Netpipe: A network protocol independent performance evaluator. *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

8. Roy-Chowdhury A, Banerjee P. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE Transactions on Computations* 1996; **45**(11):1239–1247.
9. Chen Z, Fagg GE, Gabriel E, Langou J, Angskun T, Bosilca G, Dongarra J. Fault tolerant high performance computing by a coding approach. *PPoPP '05*: *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press: New York, NY, U.S.A., 2005; 213–223.
10. Fagg G, Dongarra J. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Seventh Euro PVM/MPI User's Group Meeting2000*, vol. 1908/2000, Balatonfüred, Hungary. Springer: Heidelberg, 2000.
11. Fagg GE, Bukovsky A, Dongarra JJ. HARNESS and fault tolerant MPI. *Parallel Computing* 2001; **27**(11): 1479–1495.
12. Elnozahy M, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* (*CSUR*) 2002; **34**(3):375–408.
13. Chandy KM, Lamport L. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems* 1985; **3**(1):63–75 (ACM).
14. Hélary J-M, Mostefaoui A, Raynal M. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(9):865–877.
15. Stellner G. CoCheck: Checkpointing and process migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium* (*IPPS '96*), Honolulu, Hawaii. IEEE Computer Society Press: Silver Spring, MD, 1996.
16. Bouteiller A, Lemarinier P, Krawezik G, Cappello F. Coordinated checkpoint versus message log for fault tolerant MPI. *IEEE International Conference on Cluster Computing* (*Cluster 2003*). IEEE Computer Society Press: Silver Spring, MD, 2003.
17. Elnozahy, EN, Zwaenepoel W. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers* 1992; **41**(5):526–531.
18. Rao S, Alvisi L, Vin HM. Egida: An extensible toolkit for low-overhead fault-tolerance. *29th Symposium on Fault-Tolerant Computing* (*FTCS'99*). IEEE Computer Society Press: Silver Spring, MD, 1999; 48–55.
19. Bouteiller A, Herault T, Krawezik G, Lemarinier P, Cappello F. MPICH-V project: A multiprotocol automatic fault tolerant MPI. *International Journal on High Performance Computing and Applications* (*IJHPCA*) 2006; **20**:319–333.