Improved Message logging versus Improved coordinated checkpointing for fault tolerant MPI

Pierre Lemarinier Aurelien Bouteiller Thomas Herault Geraud Krawezik
Franck Cappello
LRI, Université de Paris Sud, Orsay, France
E-mail: {lemarini, bouteill, herault, gk,fci}@lri.fr

Abstract

Fault tolerance is a very important concern for critical high performance applications using the MPI library. Several protocols provide automatic and transparent fault detection and recovery for message passing systems with different impact on application performance and the capacity to tolerate a high fault rate. In a recent paper, we have demonstrated that the main differences between pessimistic sender based message logging and coordinated checkpointing are 1) the communication latency and 2) the performance penalty in case of faults. Pessimistic message logging increases the latency, due to additional blocking control messages. When faults occur at a high rate, coordinated checkpointing implies a higher performance penalty than message logging due to a higher stress on the checkpoint server. In this paper we extend this study to improved versions of message logging and coordinated checkpoint protocols which respectively reduces the latency overhead of pessimistic message logging and the server stress of coordinated checkpoint. We detail the protocols and their implementation into the new MPICH-V fault tolerant framework. We compare their performance against the previous versions and we compare the novel message logging protocols against the improved coordinated checkpointing one using the NAS benchmark on a typical high performance cluster equipped with a high speed network. The contribution of this paper is two folds: a) an original message logging protocol and an improved coordinated checkpointing protocol and b) the comparison between them.

Keywords: Fault tolerant MPI, coordinated checkpoint, message log, high speed networks, performance.

1 Introduction

A current trend in clusters is an increasing number of processors installed in clusters over the time [6]. As a consequence, clusters with hundreds or thousands of processors will become more and more common. Large clusters are under construction like the Sandia Red Storm with 10,368 processors. In such a cluster, hardware and software failures are not rare [4]. Even strong selection of the cluster components cannot avoid failures.

Another current trend is the use of MPI as message passing environment for high performance parallel applications. MPI in its specification and most deployed implementations (MPICH[13] and LAMMPI[7]) follows the _fail stop_ semantic (specification and implementations do not provide mechanisms for fault detection and recovery) [20]. Thus, MPI applications running on a large cluster may be stopped at any time during their execution due to an unpredictable failure.

The consequences of a failure may be very significant due to the loss of hours of computation. Some clusters are used for critical applications where the crash of the application may simply preclude delivering the result on time. Failures may also be the origin of massive waste of energy. For example, an application running on 500 CPUs that stops after 15 hours of computation because one component of the system has failed, wastes about the equivalent of 1 year of sequential computation. While the waste of hours of computation might be acceptable once, the uncertainty raised by the impossibility to guaranty that the same application will not encounter another failure during its reexecution is unacceptable for many users.

These risks have recently reactivated the research in the domain of fault tolerant MPI. Several research projects are investigating fault tolerance at different levels: network [18] , system [5], applications [11]. Different strategies have been proposed to implement fault tolerance in MPI: a) user/programmer detection and management, b) pseudo automatic guided by the programmer and c) fully automatic/transparent. For the last category, several protocols have been discussed in the literature. Thus for the user and system administrator, there is a choice not only among a variety of fault tolerance approaches but also among various fault tolerance protocols.

Despite fault tolerance in distributed system has a long history of researches, there are very few comparisons between protocols for the fully automatic and transparent approach. In a recent study, we have compared the merits of uncoordinated checkpointing based on pessimistic message logging and coordinated checkpointing based on a Chandy-Lamport like algorithm [6]. Pessimistic message logging adds an overhead even on fault free executions. Coordinated checkpoint implies the restart of all processes if a failure occurs during the execution, leading to a high stress of the checkpoint repository process. Using an experimental study, we demonstrated that the main differentiating parameter between the two approaches is the fault frequency: when the number of faults during a single execution increases, the restart cost of the coordinated checkpoint approach tends to compensate the overhead of the message logging one. There is a crosspoint from which the message logging approach outperforms the coordinated checkpoint one.

In this paper, we push forward this comparison between the two forms of fault tolerant protocols by comparing improved version of the two approaches. Namely, we compare a version of coordinated checkpoint strongly reducing the stress of the process checkpoint repository on restart and a version of message logging protocol based on a novel causal strategy strongly reducing the overhead of this approach. Thus the contribution of this paper is two folds: improved versions of well known protocols and a comparison between the two candidate approaches.

The paper is organized as following. The second part of the paper presents the related works highlighting the originality of this work. The third part presents a new organization of our fault tolerant protocol framework (an improved version compared to the one presented in [6]). Section 4 presents experiments to validate the new framework, study the performance the two improved protocols using NAS benchmarks, and compare them against previous versions. The final section concludes and presents future works.

2 Related work

Several projects are working on implementing a fault tolerant MPI using different strategies. An overview can be found in [12].

A first method consists in non automatic fault tolerant protocols such as FT-MPI [11]. Special instructions have then to be introduced in the MPI code in order to exploit the error returned by MPI instructions on failure detection. In this study we concentrate only on automatic and transparent fault tolerant protocols providing fault tolerance without any change of the MPI application code.

Global checkpointing and message logging protocol are well known automatic and transparent fault tolerance techniques. Nevertheless few comparisons exist between these two methods. Extended descriptions of these techniques can be found in [10].

Global checkpointing consists in taking a coherent snapshot of the system at a time. A snapshot is a collection of checkpoint images (one per process) with each channel state. A snapshot is said to be coherent if for all messages m from P_i to P_j, if the checkpoint on P_j has been made after reception of m then checkpoint on P_i has been made after emission of m. For our global checkpoint protocol implementation, we use the Chandy-Lamport algorithm [8].

LAMMPI [7] is one of the widely used reference implementations of MPI. It has been extended to support fault tolerance and application migration with coordinated checkpoint using the Chandy-Lamport algorithm [18, 8]. LAMMPI does not include any mechanism to include other kinds of fault tolerant protocols. In particular it does not provide an easy mechanism to implement message log protocols. It uses high level MPI global communications that are not comparable in performance with other fault tolerant implementations.

Message logging protocols are fault tolerant protocols based on the Piecewise determinism assumption [21, 10]: an execution is lead by the sequence of all its non deterministic events. During the initial execution, all non deterministic events are logged. When a process crashes, it is restarted from a local checkpoint image, then its reexecution is lead by all events logged in order to restore the same state it has before the crash. Many message logging protocols assume that non deterministic events consist only in receptions of messages [10]. There exist three classes of messages protocols, determined by the way events are logged: Optimistic, pessimistic and causal message logging [10].

Pessimistic message logging protocols ensure that all events of a process P are safely logged on stable storage before P can impact the system (sending a message) at the cost of synchronous operations. Optimistic protocols assume faults will not occur between an event and its logging, avoiding the need of synchronous operations. As a consequence, when a fault occurs, some non crashed processes may have to rollback. Causal protocols try to combine the advantages of both optimistic and pessimistic protocols: low performance overhead during failure free execution and no rollback of any non crashed process. This is realized by piggybacking events to message until these events are safely logged. A formal definition of the three logging techniques may be found in [1].

A more detailed related work on pessimistic protocols and global checkpointing protocols can be found in [6] which presents a first comparison of a global checkpointing technique and a pessimistic message logging technique. In this paper we focus on causal protocols.

Egida [17] is a framework allowing to compare fault tolerant protocols for MPI. It is implemented using an object model. It conforms to the MPICH chameleon interface, and essentially adds an interposition layer between a MPI channel interface and the upper layers of MPICH. This architecture allows benefiting from improvements of both the MPI channel interface (for example zero-copy communication over channels supporting it) and the upper layers. It provides various pre built objects and templates to specify new fault tolerant protocols such as Coordinated checkpoint and various message logging protocols. However, two elements limit its use as a generic platform to study fault tolerance protocols for a large variety of platforms: 1) the MPI channel interface may use internal low level control mechanisms (and messages) and retain messages in buffers. This may hide some nondeterministic events to Egida, making it subject to incorrect reexecutions in case of faults depending on the channel implementation; 2) it does not decouple the MPI application from the network interface. Once compiled and linked with MPICH, the application can only run for one network interface type. This precludes a key element of fault tolerance: migration on heterogeneous networks. Comparison between pessimistic and causal message logging have been studied using Egida [16]. As expected, pessimistic are faster than causal techniques for restarting since all events can be found on stable storage, but have much more overhead during failure-free execution. However the comparison we focus in this paper, between causal message log and coordinated checkpoint, has not been studied in Egida.

An estimation of the overhead introduced by causal message protocols have been studied by simulation in [3].

There are different strategies to reduce the weight of piggybacked information. Manetho [9] presents the first implementation of a causal message logging protocol. Each process maintains an *antecedence graph* which records the causal relationship between nondeterministic events. When a process sends a message to another one, it does not send the complete graph but an incremental piggybacking: all events preceding one initially created by the receiver need not to be sent back to it. An other algorithm have been proposed in [14] to reduce the amount of piggybacking on each message. It partially reorders events from a log inheritance relationship. Moreover it requires no additional piggybacking information. This allow to have some information about the causality a receiver may already hold.

In this paper, we propose a novel causal message logging protocol, using a stable node to limit the amount of causality piggybacking.

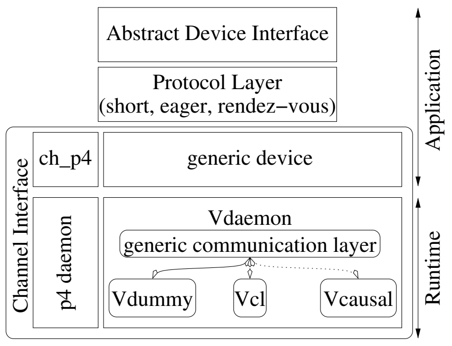

Figure 1. General Architecture of MPICH-V

3 Fault tolerance framework

MPICH-V is based on the MPICH 1.2.5 library [13], which builds a full MPI library from a channel. A channel implements the basic communication routines for a specific hardware or for new communication protocols. MPICH-V consists of a set of runtime components and a channel (ch_v) for the MPICH library.

Our first fault tolerant architecture [6] was built upon a generic device, which was interfaced with the MPICH library, and a specific communication daemon implementing the different fault tolerance protocols. The message communication routines were instantiated slightly similarly in each specific communication daemons. We designed the new architecture (figure 1) presented here in order to separate more clearly the generic lower layer from the protocol specific parts and ease the development and integration of fault tolerance protocols.

The generic architecture provides all the communication routines between the different kind of components involved in the MPICH-V architecture and is detailed below. Fault tolerant protocols are designed through the implementation of a set of hooks called in relevant routines of the generic subsystem and some specific components. We call *V-protocol* such an implementation. Currently, we provide three V-protocols (Vdummy, Vcl and Vcausal) described in the following subsections. Vdummy is a trivial implementation of these hooks which does not provide any fault tolerance (equivalent to the MPICH-P4 reference implementation). It is used to measure the raw performances of the generic communication layer. Vcl implements the Chandy-Lamport Algorithm [8] for global checkpointing of distributed applications. Vcausal implements a new causal message logging protocol.

3.1 Generic Architecture

In the generic architecture, the MPI process does not connect directly to the other ones. It communicates with a generic communication daemon, through a pair of system pipes. These daemons are connected together and relay the communications. This separation is mainly due to checkpoint-related constraints.

The daemon handles the effective communications, namely sending, receiving, reordering messages, establishing connections with all components of the system and detects failures. In each of these routines, protocol dependant functions are called. The collection of all these functions is defined through a fault tolerance API and each protocol implements this API (see next subsection). In order to reduce the number of system calls, communications are packed using *iovec* related techniques by the generic communication layer. The different communication channels are multiplexed using a single thread and the *select* system call. This common implementation of communications allows a fair comparison between the different protocols.

Three other components are involved in fault tolerance protocols: the dispatcher, the checkpoint scheduler and the checkpoint server (see typical deployments in figure 2). All these components should be run on a reliable system (potentially the same stable machine) and a more detailed description may be found in our previous papers on MPICH-V [4, 6].

3.2 Global checkpointing implementation

The conclusion of our previous comparison [6] stated that a major drawback of the global checkpointing technique was the restart time after a single crash. This time is mainly due to the stress of the checkpoint servers.

A strategy to reduce this stress is to use a simple local checkpoint image cache on each node and limit the access of the checkpoint servers during restart to the crashed processes only. Every process makes a local copy of the checkpoint image they send to the checkpoint servers (at the speed of the slowest component between the disk and the network, in order to limit the amount of memory used). When a restart occurs, instead of collecting their last checkpoint image from the checkpoint server, non-crashed processes access them from the local filesystem.

Since every component does not connect to a single repository when a restart is triggered, special care has to be taken to ensure the coherence of the global image. The cache-coherence algorithm is implemented in the checkpoint scheduler, which computes an identifier of global coherent views. When a process has successfully checkpointed its state, it notifies the checkpoint scheduler, which validates the global view when every component has checkpointed (locally and remotely) its state. The implementation of the Chandy-Lamport algorithm [8], in the Vcl V-protocol, remains the same as the one presented in [6]. The checkpoint image is taken transparently using the condor standalone checkpointing library [15].

3.3 Message logging implementation

Pessimistic message logging techniques use a reliable component to store the causality of an initial execution. In order to ensure the completeness of this causality, every pessimistic protocol does not allow the process to influence the system until every previous nondeterministic event is safely logged. In an implementation where the reliable component is a remote process, this introduces a high network latency.

In order to obtain a low latency, a process must be able to influence the system at any time, without waiting for acknowledge. This may lead to lose relevant causality information. To avoid this potential information lost, Causal Message Logging protocols attach causality information on all messages.

A main drawback of causal message logging protocols is that the amount of causality information piggybacked with every message transmission may grow with the number of messages exchanged. In the previous architecture, the reliable component used to log the causality information was a remote process called the event logger. In this improved version, we use the same component, as a protocol specific component, to reduce the amount of causality information added to messages.

Roughly speaking, the protocol is the following (see figure 3): when A receives a message from B ①, the daemon of A associates a unique identifier to the reception (causality information) and sends asynchronously this causality information to the event logger. When A has to send a message ②, the causality information of all previous receptions is added to the message only if they have not been acknowledged by the event logger yet. When the event logger acknowledges some causality information ③, this information is discarded by the communication daemon of A. If A fails, it is restarted in its last checkpoint state by the dispatcher. It collects from the event logger and from every other alive nodes all the causality information and conforms its execution to this information until it reaches the state preceeding the crash. Then, the execution continues normally. This protocol is implemented in the Vcausal protocol.

Another improvment to reduce the number of transmitted causality is used in the implementation. Each daemon remembers the last events it sends to or receive from a neighbor B. As there is a total order between the causal events generated by a single node A, no event created by the same node A preceding the last event have to be piggybacked when sending to this neighbor B.

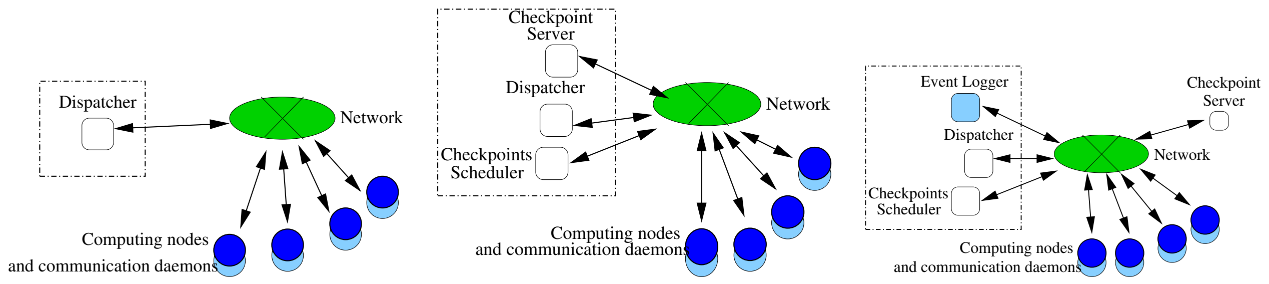

Figure 2. Typical deployment of MPICH-V for Vdummy, Vcl and Vcausal

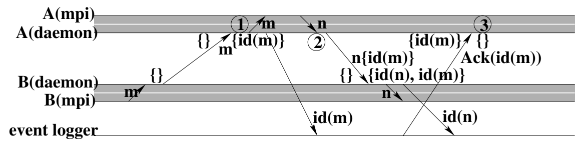

Figure 3. Sample of execution without fault for the causal logging protocol

Network	Ethernet 100Mbit/s					
Protocol	TCP	P4	Vdummy	Vcl	Vcausal	V2
Latency	75.55	99.56	134.84	138.27	156.92	291.78
Network	Myrinet 2000					
Protocol	TCP	P4	Vdummy	Vcl	Vcausal	V2
Latency	42.93	52.96	94.22	98.96	112.31	183.38
Network	SCI					
Protocol	TCP	P4	Vdummy	Vcl	Vcausal	V2
Latency	23.07	34.21	76.33	81.19	116.04	355.31

Figure 5. Latency comparison (in micro-seconds) for 1 Byte message size between Raw TCP, MPICH-P4, MPICH-Vdummy, MPICH-Vcl, MPICH-Vcausal, MPICH-V2 on Fast-Ethernet, Myrinet 2000 and SCI networks.

4 Experiments

4.1 Experimental conditions

We present a set of experiments in order to evaluate the different components of the system.

Ethernet experiments are run on a 32-nodes cluster. Each node is equipped with an AthlonXP 2800+ processor, running at 2GHz, 1GB of main memory (DDR SDRAM), and a 70GB IDE ATA100 hard drive and a 100Mbit/s Ethernet Network Interface card. All nodes are connected by a single Fast Ethernet Switch. Myrinet experiments are run on a 8-nodes cluster. Each node is similar to Ethernet nodes but are equipped with Dual AthlonXP-MP 2200+ processors, running at 1.8GHz. Myrinet network is Myrinet2000 connected by a single 8-ports Myrinet switch. SCI experiments are run on the same 32-nodes cluster as Ethernet experiments. All nodes are connected by SCI cards using a 2D-torus topology.

All these nodes use Linux 2.4.20 as operating system. The tests and benchmarks are compiled with GCC (with flag -O3) and the PGI Fortran77 compilers. All tests are run in dedicated mode. Each measurement is repeated 5 times and we present a mean of them.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. We use the NetPIPE [19] utility to measure bandwidth and latency. This is a ping pong test for several message sizes and small perturbations around these sizes. The second set of experiments is the set of kernels and applications of the NAS Parallel Benchmark suite [2], written by the NASA NAS research center to test high performance parallel machines.

For all experiments, we consider a single checkpoint server connected to the rest of the system by the same network as the MPI traffic. While other architectures have been studied for checkpoint servers (distributed file systems, parallel file systems), we consider that this system impacts the performance of checkpointing similarly for any fault tolerant protocol.

4.2 Fault Tolerant Framework performances

To perform a fair comparison between all fault tolerant protocols, we have to identify the overhead sources. We defined a shared framework for all the fault tolerant protocols. The overhead related to this framework can be mesured using the V-protocol Vdummy that does not provide any fault tolerance. We compare our framework without any fault tolerance with the reference implementation MPICH-P4 in order to summarize the framework related overhead.

The figures 4 and 5 compares bandwidth and latency of the NetPIPE ping-pong benchmark for various protocols and networks.

On the Ethernet network, Vdummy shows only a small overhead on bandwidth compared to P4. It shows a 30 percent increase in latency, this outlines the lack of a zero-copy implementation.

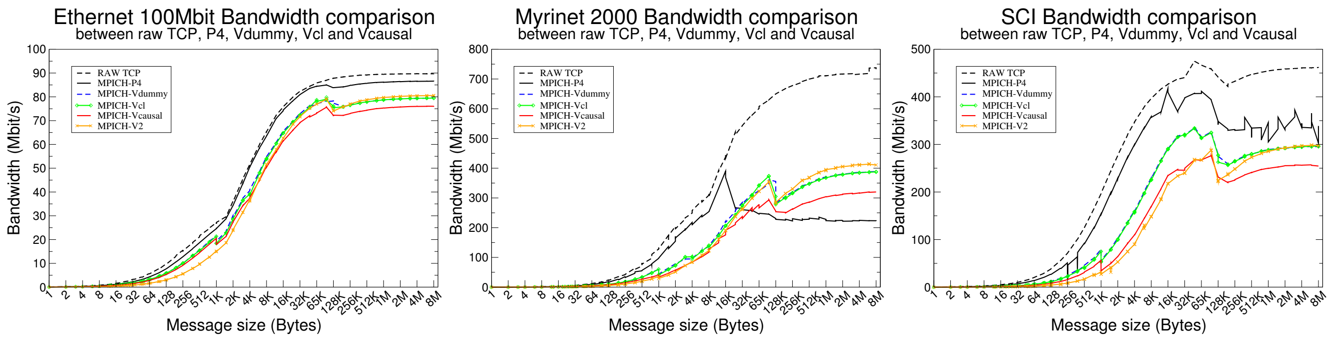

Figure 4. Bandwidth comparison between Raw TCP, MPICH-P4, MPICH-Vdummy, MPICH-Vcl, MPICH-Vcausal, MPICH-V2 on Fast-Ethernet, Myrinet 2000 and SCI networks.

Myrinet and SCI experiments are performed using the TCP/IP emulation interface of the network cards. Because of memory copies at kernel level, the Raw TCP bandwidth is half the nominal bandwidth of the hardware. On high performance networks, the copies the V framework introduces have a non negligible overhead on bandwidth and latency: bandwidth is half the one of P4 on SCI network and latency is twice the one of P4 on SCI and Myrinet. On the Myrinet network, P4 does not reach good performances. We decided not to compare our implementation with P4 on Myrinet.

On each network, the performance decreases induced by our framework is a constant multiplicative factor. Framework overhead is well identified and related to copy and computation between the emission of each network frame. This statement allows to identify which overhead is framework related and which is fault tolerant protocol related.

We also validated performances of our framework and compared each fault tolerant protocol on the set of kernels and applications of the NAS parallel benchmark on Ethernet without checkpointing (figure 6). On latency driven tests like CG and MG the V-framework reaches the same performance as the reference implementation P4. On bandwidth driven tests like BT, our framework reaches better performance than P4. This is due to architectural differences between P4 and V that allow V to perform full duplex communications.

4.3 Optimizing global checkpointing and message logging

4.3.1 Global checkpointing optimization

We introduced in Vcl the new feature of performing local checkpointing overlapped with remote checkpointing.

The figure 7 presents the impact of local checkpointing on the overall checkpointing performance of the BT class A benchmark with a single checkpoint server over Ethernet network. The overhead induced by local checkpointing is negligible compared to the total checkpoint time.

Number of nodes	4	9	16	25
Remote only	23.84	23.17	24.96	24.11
Local and remote	23.86	23.14	24.92	24.17

Figure 7. Time (in seconds) to perform remote coordinated checkpoint for BT Class A, with or without local checkpointing.

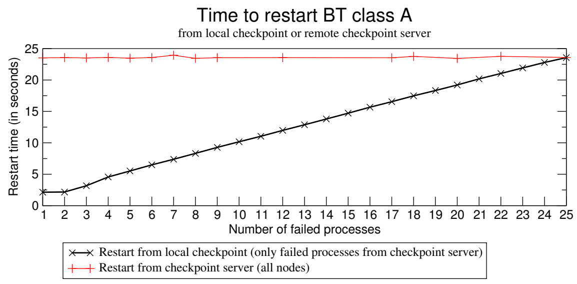

Figure 8. Time (in seconds) to restart after an increasing number of failures for BT Class A, with or without local checkpointing.

In figure 8 we compare the time to restart BT class A benchmark after an increasing number of failures for ch_cl (the non improved version of Chandy-Lamport) and Vcl with a single checkpoint server over the Ethernet network. Time to restart a single process of BT A 25 from a local checkpoint is 0.54s compared to 2.94s from a remote dedicated checkpoint server. On one hand, the time to restart in ch_cl is a linear factor of the total number of nodes. On the other hand, the time to restart in Vcl is a linear factor of the number of failed nodes: the time to restart non failed nodes from local checkpoint is totally overlapped by the time to restart failed nodes from remote checkpoint server. Moreover the overhead of the checkpoint server is limited, as the number of nodes simultaneously requesting their checkpoint image is reduced.

The use of a high performance network may remove the network bandwidth bottleneck related to remote checkpointing. However, at the checkpoint server side, the disk bandwidth is shared between all concurrent checkpoints.

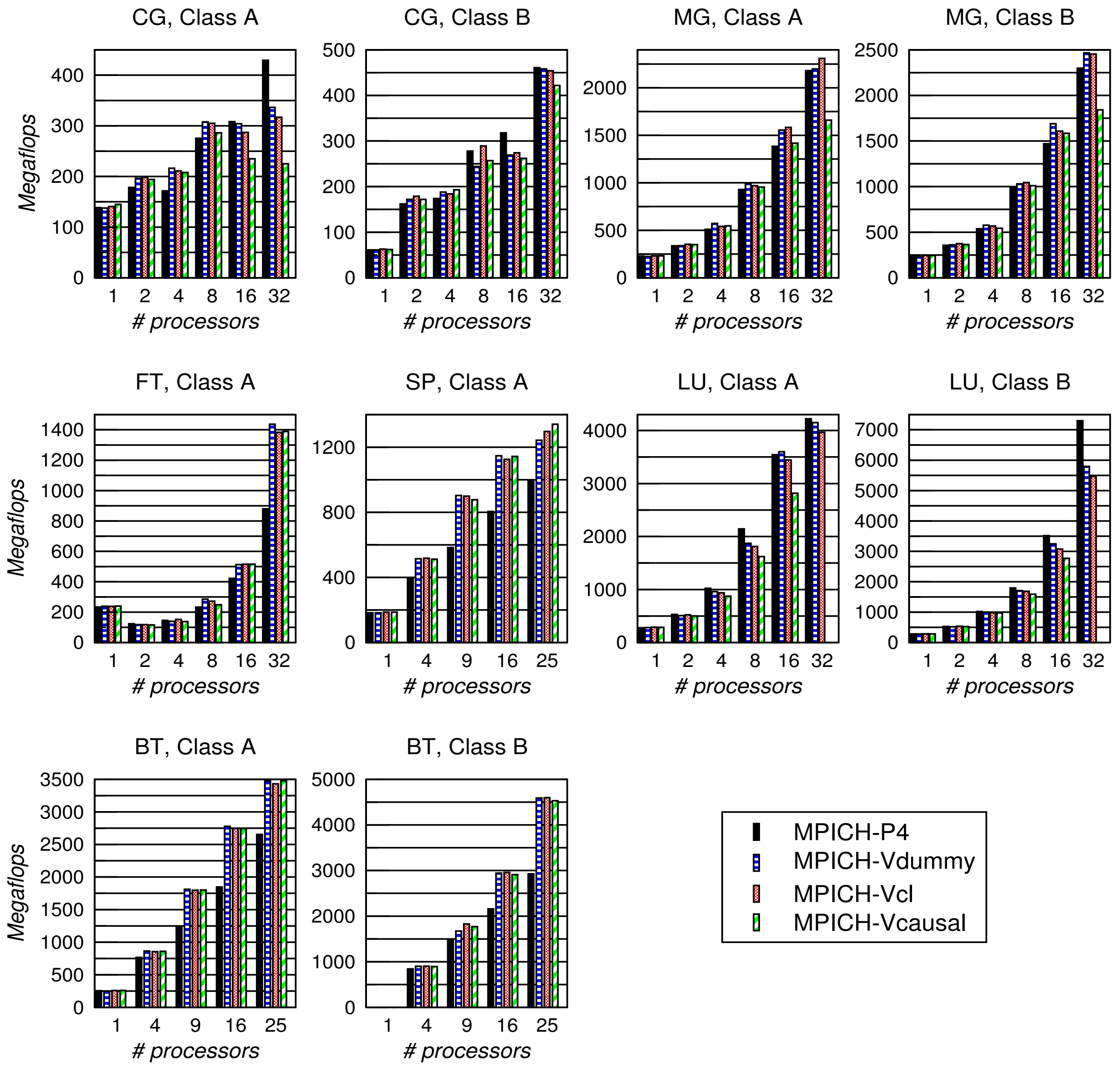

Figure 6. NAS parallel benchmark comparison between MPICH-P4, MPICH-Vdummy, MPICH-Vcl, MPICH-Vcausal on Fast-Ethernet network.

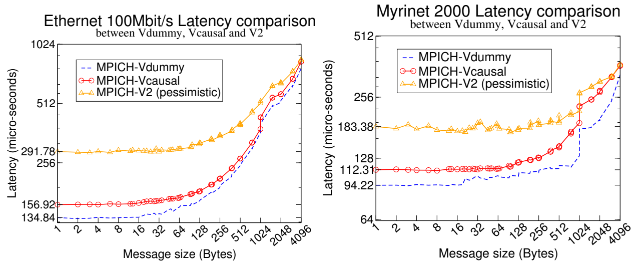

Figure 9. Ping-Pong latency comparison between MPICH-Vdummy, MPICH-Vcausal and MPICH-V2 on Fast-Ethernet and Myrinet 2000 networks.

Thus, it is sensible to consider that local disk's bandwidth is higher than shared checkpoint server disk's bandwidth.

4.3.2 Message logging optimization

In figure 9 we compare the latency between V2 (pessimistic protocol) and Vcausal. The causal protocol latency is half the one of the pessimistic protocol. The acknowledge protocol with the event logger in V2 introduces a three time higher latency compared with raw MPI communications (P4). On the one hand, as message logging is done asynchronously, it is possible that this acknowledge protocol is finished when the MPI application requests a send, inducing almost no latency. On the other hand, the event logger may introduce very high overhead when multiple computing nodes are accessing it at the same time, leading to a very high latency. The latency values presented are average values over 5000 measurements. The difference around the average values varies up to 93 percent. This is due to simultaneous requests to the event logger from multiple nodes, one of the request is delayed, and this node waits longer for it's acknowledge.

In Vcausal, in this ping-pong test, for 90 percent of the exchange, the size of the message is increased by causality information of the last reception. For small messages this leads to double the size of messages. This induces a protocol related overhead on latency in causal protocol, explaining that it does not reach the latency of raw MPI protocol.

On the Myrinet network, differences between protocols are decreased by the constant 1-copy implementation overhead. However the same behavior can be exhibited.

In figure 4 we compare the bandwidth of all the protocols we present. The low bandwidth performance in the causal protocol is due to the computation of which causal events have to be piggybacked.

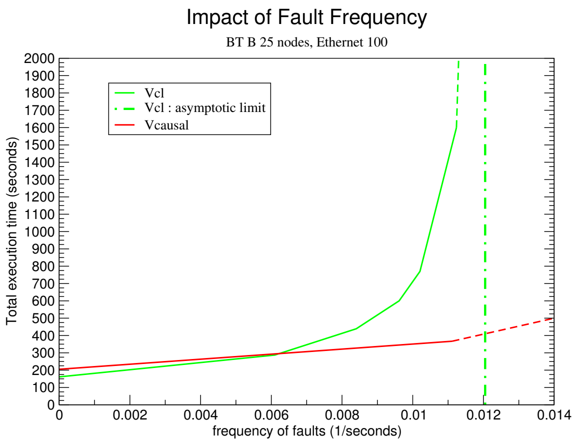

Figure 10. Fault frequency impact on execution time of BT B 25 nodes on Fast-Ethernet, using coordinated checkpoint or causal message logging as fault tolerant protocol.

4.4 Global checkpointing versus message logging comparison

The bandwidth and latency figures 4,5 present comparison between Vcl, Vcausal, with Vdummy as the reference.

As expected, the Chandy-Lamport algorithm does not induce any overhead on fault free execution. The latency of Vcl is very close to the one of Vdummy. Vcl reaches the same bandwidth as Vdummy.

The Vcausal implementation induces a 13.6 percent increase in latency compared to Vcl on Ethernet, 13.5 percent on Myrinet and 42 percent on SCI. The impact on bandwidth of the piggybacking computation is about 10 percent on each networks.

The figure 6 compares Vcl, Vcausal, and Vdummy on the NAS benchmarks. Results for FT class B are not presented as the benchmark exceeds memory requirement of our test platform, even for the P4 implementation. Vcl reaches the performance of Vdummy on all tests. Vcausal suffers from a performance decrease on fault free execution on kernel tests CG and MG, but reaches the same performance as Vdummy for all other tests.

Vcausal is unable to successfully finish LU 32 benchmark without checkpointing. This is due to the huge amount of memory used by sender-based log of message payload. In message log protocols, checkpoint frequency is related to garbage collecting of sender-based message payload. In Coordinated checkpoint it is only related to the expected fault frequency.

The figure 10 compares fault frequency impact on execution time when using whether coordinated checkpoint or causal message logging as fault tolerant protocol. This experiment consists in running BT B 25 and introduces faults. In Vcl, checkpoints are scheduled using a fixed period related to the expected fault frequency. Given a checkpoint period T and a checkpoint date T_c, a fault is introduced on a random node at $T_c + \frac{T}{2}$, defining a fault frequency f. In Vcausal checkpoint is scheduled using a round robin policy. Faults are generated on a random node using the same fault frequency f.

A similar comparison was performed in [6] between standard Chandy-Lamport algorithm (ch_cl implementation) and pessimistic message logging protocol (V2 implementation). This article outlines 1) a 40 percent overhead of V2 compared to non checkpointed execution of ch_cl and 2) that remote restart overhead of ch_cl leads pessimistic message log to perform better at high fault frequency (more than 0.002 faults per seconds).

Current comparison between improved Chandy-Lamport and causal message logging, using the same application and experimental conditions, outlines a 20 percent overhead of Vcausal over a non checkpointed execution of Vcl, reducing the fault free performance difference between message logging and coordinated checkpoint strategies. On the other hand, Vcl does not suffer from the high remote restart overhead. It has a better fault resilience than ch_cl and supports higher fault frequencies. As a consequence, the crosspoint between the two protocols appears at higher fault rates (more than 0.006 faults per seconds), even if causal protocol performs better than pessimistic ones without faults. However, Vcl stills not ensure progression of the computation when reaching 0.012 faults per seconds, while causal protocol stills perform at 50 percent of its fault free performance.

5 Conclusion

Large scale cluster and Grid system raise the issue of tolerance to frequent and numerous faults. Since these systems are mostly programmed using MPI, the use of a fault tolerant MPI implementation will become unavoidable. Among the automatic/transparent fault tolerant approaches, two main classes can be considered, using either coordinated checkpoint or message logging. It has been previously proven that there is a crosspoint from which pessimistic message logging performs better than coordinated checkpoint. The purpose of this paper was to study two optimizations of coordinated checkpoint and message logging, in the potential perspective to find that one technique would always perform better.

We have implemented a shared framework from the MPICH 1.2.5. From this framework we have implemented 1) coordinated checkpoint with the local checkpoint capability and 2) causal message logging with asynchronous stable component to store causality (MPICH-Vcl and MPICH-Vcausal respectively). After having validated the performance of our generic shared framework, we compared local checkpoint improved Chandy-Lamport implementation to remote checkpoint standard Chandy-Lamport one (MPICH-CL), and causal message logging to pessimistic message logging (MPICH-V2) for various networks including Fast-Ethernet, Myrinet 2000 and SCI. We have demonstrated that recovery overhead of Vcl is significantly lower than ch_cl without any additional fault free overhead. We have demonstrated that latency overhead of Vcausal is reduced at the cost of a slight bandwidth decrease compared to V2. We have demonstrated that the fault free performance difference between Vcausal and Vcl is smaller than between V2 and CL, and that Vcl tolerates higher fault frequency than ch_cl, but still does not reach fault resilience of message logging techniques due to checkpoint server stress during checkpoint.

Due to better fault resilience, the minimal fault frequency from which message logging outperforms coordinated checkpoint is increased from 0.002 faults per seconds to 0.006 faults per seconds. If we consider an application with a larger data set of 1GB, this crosspoint should appear at one fault every 9 hours. If we consider that real clusters's MTBF are greater than 9 hours, it appears that coordinated checkpoint are more appropriate in such environments.

Improving the shared framework to perform more accurate experiments on high performances networks belongs to a set of planned experiments a) to find the crosspoint for high performance networks, b) to understand in practice (with real software and cluster) the impact of checkpoint server architecture and c) to use large number of nodes to understand the cost of the Chandy-Lamport algorithm by itself over various or heterogeneous network configurations.

6 Acknowledgments

MPICH-V belongs to the ”Grand Large” project of the PCRI (Pole Commun de Recherche en Informatique) of Saclay (France) and the INRIA Futurs.

MPICH-V project is partially funded, through the CGP2P project, by the French ACI initiative on GRID of the ministry of research.

References

[1] L. Alvisi and K. Marzullo. Message logging : Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 229–236. IEEE CS Press, May-June 1995.

[2] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[3] Karan Bhatia, Keith Marzullo, and Lorenzo Alvisi. The relative overhead of piggybacking in causal message logging protocols. In *17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 348–353. IEEE CS Press, 1998.

[4] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky,

Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *High Performance Networking and Computing (SC2002)*, Baltimore USA, November 2002. IEEE/ACM.

[5] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *High Performance Networking and Computing (SC2003)*. Phoenix USA, IEEE/ACM, November 2003.

[6] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, december 2003.

[7] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[8] K. M. Chandy and L.Lamport. Distributed snapshots : Determining global states of distributed systems. In *Transactions on Computer Systems*, volume 3(1), pages 63–75. ACM, February 1985.

[9] Elnozahy, Elmootazbellah, and Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers*, 41(5), May 1992.

[10] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375 – 408, september 2002.

[11] G. Fagg and J. Dongarra. FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *7th Euro PVM/MPI User's Group Meeting2000*, volume 1908 / 2000, Balatonfred, Hungary, september 2000. Springer-Verlag Heidelberg.

[12] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[13] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[14] Byoungjoo Lee, Taesoon Park, Heon Young Yeom, and Yookun Cho. An efficient algorithm for causal message logging. In *17th Symposium on Reliable Distributed Systems (SRDS 1998)*, pages 19–25. IEEE CS Press, October 1998.

[15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report Technical Report 1346, University of Wisconsin-Madison, 1997.

[16] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The cost of recovery in message logging protocols. In *17th Symposium on Reliable Distributed Systems (SRDS)*, pages 10–18. IEEE CS Press, October 1998.

[17] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 48–55. IEEE CS Press, 1999.

[18] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[19] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems.*, June 1996.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[21] E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, August 1985.