MPICH-V PROJECT: A MULTIPROTOCOL AUTOMATIC FAULT-TOLERANT MPI

A. Bouteiller
T. Herault
G. Krawezik
P. Lemarinier
F. Cappello

INRIA/LRI, UNIVERSITÉ PARIS-SUD, ORSAY, FRANCE
(BOUTEILL@LRI.FR)

Abstract

High performance computing platforms such as Clusters, Grid and Desktop Grids are becoming larger and subject to more frequent failures. MPI is one of the most used message passing libraries in HPC applications. These two trends raise the need for fault-tolerant MPI. The MPICH-V project focuses on designing, implementing and comparing several automatic fault-tolerant protocols for MPI applications. We present an extensive related work section highlighting the originality of our approach and the proposed protocols. We then present four fault-tolerant protocols implemented in a new generic framework for fault-tolerant protocol comparison, covering a large spectrum of known approaches from coordinated checkpoint, to uncoordinated checkpoint associated with causal message logging. We measure the performance of these protocols on a micro-benchmark and compare them with the NAS benchmark, using an original fault tolerance test. Finally, we outline the lessons learned from this in depth fault-tolerant protocol comparison of MPI applications.

Key words: fault-tolerant MPI, performance evaluation, coordinated checkpoint, message logging

The International Journal of High Performance Computing Applications,
Volume 20, No. 3, Fall 2006, pp. 319–333
DOI: 10.1177/1094342006067469

Figures 2–9 appear in color online: http://hpc.sagepub.com

1 Introduction

A current trend in high performance computing is the use of large scale computing infrastructures such as clusters and Grid deployments harnessing thousands of processors. Machines of the Top 500, current large Grid deployments (TERA Grid, NAREGI Grid, DEISA, etc.) and campus/company wide Desktop Grids are examples of such infrastructures. In the near future, these infrastructures will even become larger. The quest for petaflops scale machines leads us to consider clusters with 100000 nodes. Grids are expected to expand in terms of number of sites, applications and users. Desktop Grids are also expected to harness more participants thanks to an increasing software maturity and social acceptance. In all these infrastructures, node and network failures are likely to occur, leading to the necessity of a programming model providing fault management capabilities and/or runtime featuring fault-tolerant mechanisms.

Another current trend is the use of MPI as the message passing environment for high performance parallel applications. Because of its high availability on parallel machines from low cost clusters to clusters of vector multiprocessors, it allows the same code to run on different kind of architectures. It also allows the same code to run on different generations of machines, ensuring a long lifetime for the code. MPI also conforms to popular high performance, message passing, programming styles. Even if many applications follow the SPMD programming paradigm, MPI is also used for Master-Worker execution, where MPI nodes play different roles. These three parameters make MPI the first choice programming environment for high performance applications. MPI in its specification (Snir et al. 1996) and most deployed implementations, MPICH (Gropp et al. 1996) and LAMMPI (Burns, Daoud, and Vaigl 1994), follows the *fail stop* semantic (specification and implementations do not provide mechanisms for fault detection and recovery). Thus, MPI applications running on a large cluster may be stopped at any time during their execution as a result of an unpredictable failure.

The need for fault-tolerant MPI implementations has recently reactivated research in this domain. Several research projects are investigating fault tolerance at different levels: network (Sankaran et al. 2003), system (Bouteiller et al. 2003a), and applications (Fagg and Dongarra 2000). 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. As a consequence, for the user and system administrator, there is a choice not only among a variety of fault tolerance approaches but also among various fault-tolerant protocols.

Despite the long history of research on fault tolerance in distributed systems, there are very few experimental comparisons between protocols for the fully automatic and transparent approaches. There are two main approaches for automatic fault tolerance: coordinated checkpoint and uncoordinated checkpoint associated with message logging. Coordinated checkpoint implies synchronized checkpoints and restarts that may preclude its use on large scale infrastructures. However, this technique adds low overhead during failure-free execution, ensuring high performance for communication intensive applications. Uncoordinated checkpoint associated with message logging features has the opposite properties. It is efficient on reexecution of crashed processes because only these are reexecuted, but it adds a high communication overhead even on fault-free executions. Each of the two approaches can be implemented using different protocols and optimizations. In the context of MPI, very little is known about the merits of the different approaches in terms of performance on applications and capabilities to tolerate high fault frequency, a parameter which is correlated to the scale of the infrastructures.

To investigate this issue, we started the MPICH-V project in September 2001 with the objective of studying, proposing, implementing, evaluating and comparing a large variety of MPI fault-tolerant protocols for different kinds of platforms: large Clusters, Grids and desktop Grids. After three years of research, we have developed a generic framework and a set of four fault-tolerant protocols (two pessimistic message logging protocols: MPICH-V1 (Bosilca et al. 2002) and MPICH-V2 (Bouteiller et al. 2003a); a global checkpoint strategy based on Chandy-Lamport algorithm, MPICH-V/CL (Bouteiller et al. 2003c); and a causal message logging protocol, MPICH-V/causal). This paper presents the different protocols, their implementation within the framework, and sums up our learning in this still ongoing research on automatic and transparent fault tolerance for MPI.

The paper is organized as follows. Section 2 presents related work highlighting the originality of this work. Section 3 presents the different protocols we have implemented and compares their advantages and drawbacks. Section 4 presents performance, fault tolerance evaluation and comparison of all these protocols using NAS benchmarks. Section 5 sums up what we learned from the experience of these protocols.

2 Related Works

Automatic and transparent fault-tolerant techniques for message passing distributed systems have been studied for a long time. We can distinguish few classes of such protocols: replication protocols, rollback recovery protocols and self-stabilizing protocols. In replication techniques, every process is replicated f times, f being an upper bound on the number of simultaneous failures. As a consequence, the system can tolerate up to f concurrent faults but divides the total computation resources by a factor of f. Self-stabilizing techniques are used for non-terminating computations, such as distributed system maintenance. Rollback recovery protocols consist of taking checkpoint images of processes during initial execution and rollback some processes to their last images when a failure occurs. These protocols take special care to respect the consistency of the execution in different manners. Rollback recovery is the most studied technique in the field of fault-tolerant MPI. Several projects are working on implementing a fault-tolerant MPI using different strategies. An overview can be found in Gropp and Lusk (2002). Figure 1 summarizes fault tolerant MPI implementations classification with respect to fault tolerant technique and level in the software stack.

Rollback recovery protocols include global checkpoint techniques and message logging protocols. Extended descriptions of these techniques can be found in Elnozahy et al. (2002).

2.1 Global Checkpoint

Three families of global checkpoint protocols have been proposed in the literature (Elnozahy et al. 2002). The first family gathers uncoordinated checkpoint protocols: every process checkpoints its own state without coordination with other processes. When a fault occurs, the crashed processes restart from previous checkpoints. Processes that have received a message from a rolled back process also have to rollback if this message was initially received before the checkpoint image of this process. This may lead to a domino effect where a single fault makes the whole system rollback to the initial state. As a consequence, this kind of protocol is not used in practice.

The second family of global checkpoint protocols gathers the coordinated checkpoint protocols. Coordinated checkpoint consists of taking a coherent snapshot of the whole system at a given time. A snapshot is a collection of checkpoint images (one per process) with each channel state (Chandy and Lamport 1985). A snapshot is said to be coherent if for all messages m from process P to process Q, if the checkpoint on Q has been made after reception of m then the checkpoint on P has been made after emission of m. When a failure occurs, all processes are rolled back to their last checkpoint images.

The first algorithm to coordinate all the checkpoints is presented in Chandy and Lamport (1985). This algorithm supposes all channels are FIFO queues. Any process can decide to start a checkpoint. When a process checkpoints, it sends special messages called *markers* in its communication channels. When a process receives a marker for the first time, it checkpoints. After beginning a checkpoint,

all messages received from a neighbor are added to the checkpoint image, until the marker reception.

Other fault-tolerant MPI implementations use this algorithm (Burns, Daoud, and Vaigl 1994; Stellner 1996; Agbaria and Friedman 1999). For example, Cocheck (Stellner 1996) is an independent application implemented on top of the message passing system (tuMPI) which can be easily adapted for different systems.

Starfish (Agbaria and Friedman 1999) modifies the MPI API in order to allow users to integrate some checkpointing policies. Users can choose between coordinated and uncoordinated (for trivial parallel applications) checkpoints strategies. For an uncoordinated checkpoint, the environment sends to all surviving processes a notification of the failure. The application may take decisions and corrective operations to continue the execution (i.e. adapts the data sets repartition and work distribution).

LAMMPI (Burns, Daoud, and Vaigl 1994) 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 (Chandy and Lamport 1985; Sankaran et al. 2003). LAMMPI does not include any mechanism for other kinds of fault-tolerant protocols. In particular it does not provide straightforward mechanisms to implement message logging protocols. It uses high level MPI global communications that are not comparable in performance with other fault-tolerant implementations.

Clip (Chen, Lee, and Planck 1997) is a user level coordinated checkpoint library dedicated to IntelParagon systems. This library can be linked to MPI codes to provide semi-transparent checkpoint. The user adds checkpoint calls in his code but does not need to manage the program state on restart.

Checkpointing adds an overhead increasing the application execution time. There is a tradeoff between the checkpoint cost and the cost of restarts due to faults that lead to the selection of the best checkpoint interval (Wong and Franklin 1993; Plank and Elwasif 1998; Planck and Thomason 2001). However, the delay between checkpoints should be computed from the application execution time without checkpoint and from the cluster mean time between failure (MTBF). Thus, it should be tuned for each configuration of application, platform performance and number of processes.

The third family of global checkpoint protocols gathers Communication Induced Checkpointing (CIC) protocols. Such protocols try to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggybacks causality dependencies in all messages and detects risk of inconsistent state. When such a risk is detected, some processes are forced to checkpoint. While this approach is very appealing theoretically, relaxing the necessity for global coordination, it turns out to be inefficient in practice. Alvisi et al. (1999) present a deep analysis of the benefits and drawbacks of this approach. The two main drawbacks in the context of cluster computing are 1) CIC protocols do not scale well (the number of forced checkpoints increases linearly with the number of processes) and 2) the storage requirement and usage frequency are unpredictable and may lead to checkpoint as frequently as coordinated checkpoint.

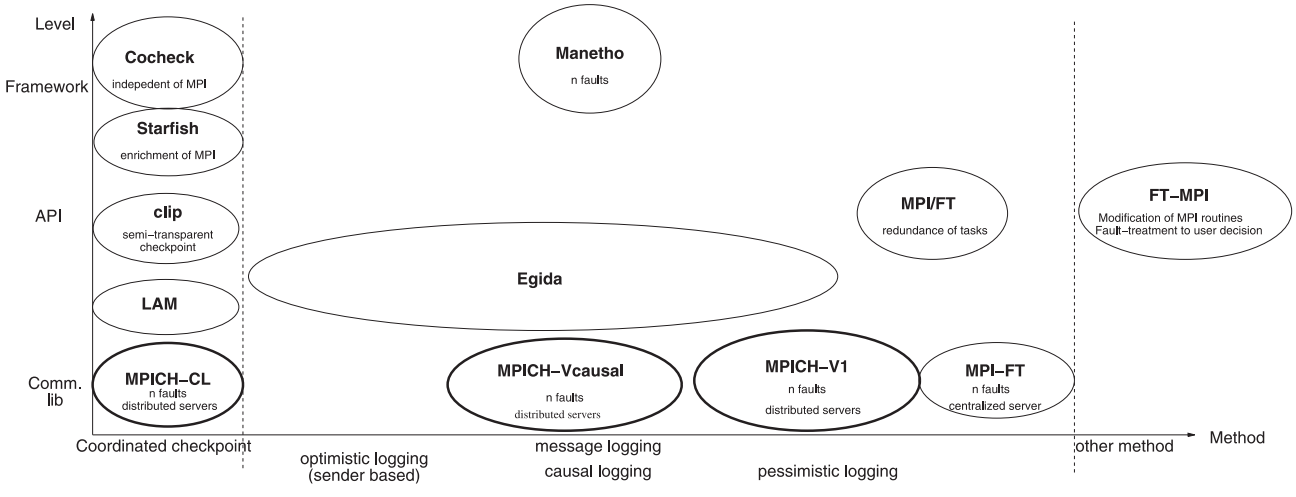


Fig. 1 Classification by techniques and level in the software stack of fault-tolerant message passing systems.

2.2 Message Logging

Message logging consists of forcing the reexecution of crashed processes from their last checkpoint image to reach the state immediately preceding the crashing state, in order to recover a state coherent with non-crashed ones. All message logging protocols suppose that the execution is *piecewise deterministic*. This means that the execution of a process in a distributed system is a sequence of deterministic and nondeterministic events and is led by its nondeterministic events. Most protocols suppose that the reception events are the only possible nondeterministic events in an execution. Thus message logging protocols consists of logging all reception events of a crashed process and replaying the same sequence of receptions.

There are three classes of message logging protocols: pessimistic, optimistic and causal message logging. Pessimistic message logging protocols ensure that all events of a process *P* are safely logged on reliable storage before *P* can impact the system (send 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 Alvisi and Marzullo (1995).

2.2.1 Optimistic message logging A theoretical protocol (Strom and Yemini 1985) presents the basic aspect of optimistic recovery. It was first designed for clusters, partitioned into a fixed number of *recovery units* (*RU*s), each one considered as a computation node. Each *RU* has a message logging vector storing all messages received form the other *RU*s. Asynchronously, an *RU* saves its logging vector to a reliable storage. It can also checkpoint its state. When a failure occurs, it tries to replay input messages stored in its reliable storage from its last checkpoint; messages sent since last checkpoint are lost. If the *RU* fails to recover a coherent state, other *RU*s concerned by lost messages should be rolled back too, until the global system reaches a coherent state.

Sender based message logging (Johnson and Zwaenepoel 1987) is an optimistic algorithm tolerating one failure. Compared with Strom and Yemini (1985), this algorithm consists of logging each message in the volatile memory of the sender. Every communication requires three steps: send; acknowledge + *receipt count*; acknowledge of the acknowledge. When a failure occurs, the failed process rolls back to its last checkpoint. Then it broadcasts requests to retrieve initial execution messages and replays them in the order of the *receipt count*.

Pruitt (1998) presents an asynchronous checkpoint and rollback facility for distributed computations. It is an implementation of the sender-based protocol proposed by Juang and Venkatesan (1991). This implementation is built from MPICH.

2.2.2 Pessimistic message logging MPI-FT Louca et al. (2000) uses a special entity called Observer. This process is presumed to be reliable. It checks the availability of all MPI peers and respawns crashed ones. Messages can be logged following two different approaches. The first one logs messages locally to the sender in an optimistic message logging way. In the case of a crash, the Observer controls all processes asking them to re-send old messages. The second approach logs all the messages on the Observer in a pessimistic message logging way. Our pessimistic protocol always relies on distributed components in order to be scalable.

Another pessimistic protocol is presented in Strom, Bacon, and Yemeni (1988). However, the study provides no architecture principle, theoretical foundation, implementation detail, performance evaluation or merit comparison against non-fault-tolerant MPI and other fault-tolerant MPI implementations.

In this paper, we present two novel pessimistic logging protocols using some remote reliable components.

2.2.3 Causal message logging Manetho (Elnozahy and Zwaenepoel 1992a, 1992b) 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, it does not send the complete graph but an incremental piggybacking: all events preceding one initially created by the receiver do not need to be sent back to it.

Another algorithm has been proposed in Lee et al. (1998) 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 permits having some information about the causality that a receiver may already hold.

An estimation of the overhead introduced by causal message protocols has been studied by simulation in Bhatia, Marzullo, and Alvisi (1998).

We will propose and test a new causal protocol, which relies on a stable component to reduce optimally the size of the piggyback in all messages.

2.3 Other Fault Tolerant MPI

FT-MPI (Fagg and Dongarra 2000; Fagg, Bukovsky, and Dongarra 2001) handles failures at the MPI communicator level and lets the application manage the recovery. Special instructions have then to be added to the MPI code in order to exploit the error returned by MPI instructions on failure detection. The main advantage of FT-MPI is its performance since it does not checkpoint nor log messages, but its main drawback is the lack of transparency for the programmer.

Egida (Rao, Alvisi, and Vin 1999) is a framework allowing comparison between fault-tolerant protocols for MPI. Rao, Alvisi, and Vin (1998) present a comparison between pessimistic and causal message logging with this framework. As expected, pessimistic are faster than causal techniques for restarting because all events can be found on stable storage, but have much more overhead during failure-free execution. We will demonstrate that this is still the case with our novel pessimistic and causal protocols. We will also compare coordinated checkpoint with message logging techniques, an issue left unaddressed by Egida.

MPI/FT (Batchu et al. 2001) considers task redundancy to provide fault tolerance. It uses a central coordinator that monitors the application progress, logs the messages, manages control messages between redundant tasks and restarts failed MPI process. A drawback of this approach is the central coordinator which, according to the authors, scale only up to 10 processors. Our new protocols use centralized architecture for mechanisms requiring infrequent communications and decentralized architecture for mechanisms requiring high communication workload.

The research presented in this paper differs from previous work in many respects. We present and study original pessimistic and causal protocols. We present an optimization of the Chandy-Lamport algorithm never discussed in the literature. We present the implementation of these protocols within a new framework, located at an original software-stack level. We compare protocols that have never been confronted: coordinated checkpoint and message logging. Finally, we include in our comparison the fault tolerance merit, based on a new criteria: the performance degradation according to fault frequency.

3 Protocols Implemented in MPICH-V

We have developed four different protocols, each protocol has been proven correct on a theoretical basis. All protocols have been implemented within the MPICH-V framework that is briefly presented.

An example of execution including two MPI communications and one fault for each of these protocols is presented in Figures 3 to 6. First, we developed a pessimistic message logging protocol called MPICH-V1 for Desktop Grids. The key points of the proof of this protocol can be found in Bosilca et al. (2002). The main goal of MPICH-V2 (Bouteiller et al. 2003a), our second pessimistic protocol, was to reduce the number of required stable components and enhance performance compared with MPICH-V1. We designed this protocol for a cluster usage trying to increase significantly the bandwidth of MPICH-V1. On some applications, it appears that MPICH-V2 suffers from a high latency. Our causal message logging protocol was an attempt to decrease this latency at the cost of a higher reexecution time. We implemented MPICH-V/CL (Bouteiller et al. 2003c) a coordinated checkpoint protocol, as a reference for comparison with message logging approaches, in terms of overall performance and fault tolerance. All these protocols feature a different balance between latency, bandwidth and reexecution cost in case of failure.

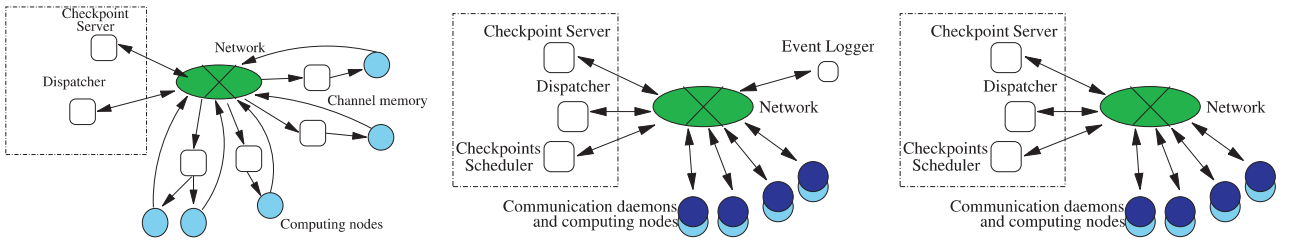
3.1 The MPICH-V Generic Framework

MPICH-V is based on the MPICH library (Gropp et al. 1996), 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.

The different fault-tolerant protocols are implemented at the same level of the software hierarchy, between an MPI high level protocol management layer (managing global operations, point to point protocols, etc.) and the low level network transport layer. This is one of the most relevant layers for implementing fault tolerance if criteria such as design simplicity and portability are considered. All fault-tolerant protocols also use the same checkpoint service.

The checkpoint of the MPI application is performed using the Condor Standalone Checkpoint Library (CSCL; Litzkow et al. 1997). When a checkpoint is requested, the MPI application forks. The original process continues to compute, while the forked copy closes all communications (with the daemon or the Channel Memories), performs the checkpoint, and then exits. The checkpoint is sent to the checkpoint server without intermediate copy in order to pipeline checkpoint image creation and transmission.

The MPICH-V fault tolerance frameworks sit within the channel interface, thus at the lowest level of the MPICH software stack. Among the other benefits, this permits use of the unmodified MPICH implementation of point to point and global operations, as well as complex concepts such as topologies and communication contexts. A potential drawback of this approach might be the necessity to implement a specific driver of all types of Network Interface (NIC). However, several NIC vendors are providing low level, high performance (zero copy) generic socket interfaces such as Socket GM for Myrinet, SCI Socket



(a) MPICH-V1 (b) MPICH-V2 & MPICH-V/causal (c) MPICH-V/cl

Fig. 2 Typical deployment for our fault-tolerant protocol. White components are supposed to be stable.

for SCI and IPoIB for Infiniband. MPICH-V protocols typically sit on top of these low level drivers.

Figure 2 compares typical deployments of the MPICH-V frameworks. Many components such as the checkpoint server, the dispatcher, and the checkpoint scheduler are shared on these deployments. The scheduling of process checkpoints has been added in the MPICH-V2 architecture, and the components were extended to fit the needs of new protocols over time. The checkpoint server and the dispatcher have been slightly optimized since their first version, but their functionalities and interfaces have not evolved.

3.2 MPICH-V1: A Pessimistic Protocol for High Volatility and Heterogeneous Resources

This first implementation of a pessimistic message logging is based on the original concept of the Channel Memory (Bosilca et al. 2002). The Channel Memory (CM) is a remote stable component intended to store the payload and the order of reception of MPI messages of a particular receiver. Each MPI process is associated with its own CM. However, in the implementation, a single CM can serve several computing nodes. Figure 3 presents an execution example of MPICH-V1. When a process MPI_A sends a message m to a process MPI_B, it sends it to the CM (CM_B) of the receiver. When a process MPI_B wants to receive a message, it requests it from its own CM (CM_B) which replies by sending the requested message m. Thus when a process MPI_A fails, it is restarted from a checkpoint image and contacts its CM (CM_A) to get the messages to reexecute.

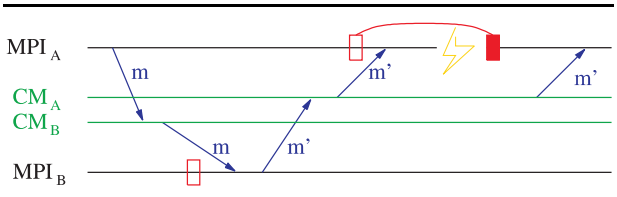


Fig. 3 Example of execution with MPICH-V1.

The main drawbacks of this approach are 1) every message is translated into two communications, which drastically impacts the latency and 2) the bandwidth of the nodes hosting the channel memories is shared among several computing nodes. This may lead to a bandwidth bottleneck at the Channel Memory level, and may require a large number of *CM* (stable nodes). Nonetheless 1) reexecution of a crashed process is faster than for every other protocol we have developed, particularly when multiple faults occur at the same time, and 2) considering a heterogeneous network composed of slow connections to computing nodes and fast connections to stable resources, the need for a large number of channel memories to reach the non-fault-tolerant MPI performance is decreased. This makes the V1 protocol adapted for highly volatile systems with heterogeneous networks such as Desktop Grids. In addition, the CM implements an implicit tunnel between computing nodes and thus enables communications between nodes protected by firewalls or behind NAT or proxies, which is typically the case of Desktop Grid deployments.

3.2.1 Implementation notes on the MPICH-V framework As in all the other fault-tolerant protocols presented here, the dispatcher of the MPICH-V environment has two main purposes: 1) to launch the whole runtime environment (encompassing the computing nodes and the auxiliary "special" nodes) on the pool of machines used for the execution, and 2) to monitor this execution, by

detecting any fault (node disconnection) and relaunching the crashed nodes.

All protocols use the same checkpoint server. This component is intended to store checkpoint images on a reliable node. However, the checkpoint server can survive intermittent faults. It is a high performance multi-process, multiple connection, image holder. Images are marked by a sequence number, and the checkpoint server replies to requests to modify, add, delete or give back a particular image. All checkpoint requests are transactions: in case of a client failure, the overall transaction is canceled, with no modification of the checkpoint server state.

MPICH-V1 does not use a checkpoint scheduler and checkpoints are taken as a result of a local decision, independently for each computing node.

The driver is the part of the fault-tolerant framework linked with the MPI application. It implements the Channel Interface of MPICH. Our implementation only provides synchronous functions (bsend, breceive, probe, initialize and finalize), as the asynchronism is relayed to another component of the architecture.

The driver of MPICH-V1 is basically a client of the CM servers. Connexion to all CM servers is performed during the initialization function of the driver. It implements synchronous send and receive operations through synchronous TCP connexion to CM servers. The performance does not suffer as a result of this synchronous implementation, as the asynchronism is relayed to the CM server.

The CM stores all received message payloads and delivery orders of a specific MPI process. Each computation process is connected to one Channel Memory for its receptions. The implementation is slightly different: a CM server is a process that can act as a single Channel Memory for many computing nodes.

The CM is a multi-threaded application that handles a set of TCP connections using a pool of threads. Threads are queued following a round-robin strategy for entering a select/communicate loop. Their overlapping is designed to improve the CM availability for new requests.

No specific QoS policy is implemented, so all client computing nodes of a CM share the bandwidth of the stable node hosting it.

3.3 MPICH-V2: A Pessimistic Protocol for Large Clusters

In MPICH-V2, processes communicate directly. Figure 4 presents an example execution of this protocol. This protocol relies on a sender based approach: all payloads are stored on the volatile memory of message senders (between brackets on the figure). Only the message causality is stored on a stable storage called Event Logger. ① When a process receives a message m, it sends to the event logger information about the reception $id(m)$. ② When a process

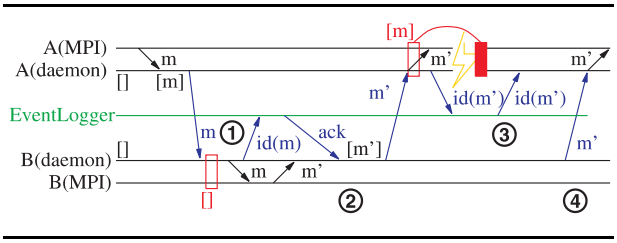


Fig. 4 Example of execution with MPICH-V2.

has to send a message m', it first waits for acknowledgment of causality information previously sent to the event logger and stores the message payload in its own memory. The termination of these two operations triggers the authorization to send the message m'. This leads to a high latency for short messages. ③ When a failure occurs, the failed process is restarted from its last checkpoint and retrieves from the Event Logger the ordered list of the receptions it has to reexecute. ④ Then it requests the initial sender of each message to send it again from its local sender based payloads. Thus, the sender based message payload must be included in checkpoint images to tolerate simultaneous faults. The reexecution of a crashed process is slower than for MPICH-V1 since all message payloads needed for the reexecution are requested from several and possibly failed processes.

However, a deployment of MPICH-V2 requires many fewer event loggers than channel memories for MPICH-V1 because of the sender based approach and the reduced amount of information logged on stable components. In a typical deployment (Figure 2(b)), all stable components except checkpoint servers can be launched in a single stable node without impact on performance (Bouteiller et al. 2003a). Moreover the direct communication between daemons enhances raw communication performance compared with MPICH-V1.

3.3.1 Implementation notes on the MPICH-V framework The event logger is a specific component for message logging protocol used in MPICH-V2 and MPICH-V/causal. Message logging protocols rely on nondeterministic events, namely reception events (see Section 2.2). Such an event can be registered as a determinant: a tuple of a fixed number of pieces of basic information where the most important are: which process has sent the message; at which logical clock it was sent; and at which logical clock it was received (Alvisi and Marzullo 1995). Note that the determinant does not record the payload of the message. Event loggers are used as remote stable databases to collect for each process the sequence of reception events it has made. When a daemon delivers a message to its MPI process, it sends the determinant to the event log-

ger of this process, using a synchronous or asynchronous acknowledge protocol. When a failure occurs the restarting daemon asks the event logger for the sequence of events it has to replay.

The checkpoint scheduler requests computation processes to checkpoint according to a given policy. The policy is protocol dependent. However, the checkpoint scheduler includes requesting information to computing nodes, which is useful for implementing a smart policy for message logging protocols. It also implements a global checkpoint sequence number, usable in a coordinated checkpoint policy to ensure success of a global checkpoint.

In the MPICH-V2 protocol, the sender-based message logging has a major impact on the amount of memory used by a process, increasing the checkpoint image size of this process. Nonetheless checkpointing a process allows the processes that have sent messages to it to flush the corresponding message payloads. A special process called checkpoint scheduler has been added in this framework, not to ensure the fault tolerance but to enhance its overall performance.

The checkpoint scheduler can ask processes for information about their amount of message payload and a best effort checkpoint scheduling policy can be implemented using this information. Other checkpoint scheduling policies such as random and round-robin selections may be used.

3.4 MPICH-V/causal: A Causal Message Logging Protocol

The main drawback of MPICH-V2 is the high latency introduced by the necessity to wait for the event logger acknowledgment of previous reception before sending a message. Causal protocols address this problem by piggybacking non-acknowledged causality to the application messages. Nonetheless, if this piggybacking size increases with the number of exchanged messages during the computation, it can have a major impact on the bandwidth. Thus designing and implementing a protocol reducing as much as possible the overall piggyback size is the cornerstone for limiting the protocol overhead.

Figure 5 presents an example execution of our causal protocol. ① When a process receives a message, it stores and asynchronously sends the resulting causality information to the event logger and stores it in a list of causality information to piggyback (between braces in the figure). ② The event logger acknowledges this information asynchronously as fast as possible. If this acknowledge does not come back before the application needs to send a new message m', then the causality informations list $\{id(m)\}$ is piggybacked to the message. ③ There is only the need to wait for completion of the local (sender based) logging before the message can be sent. Moreover every daemon

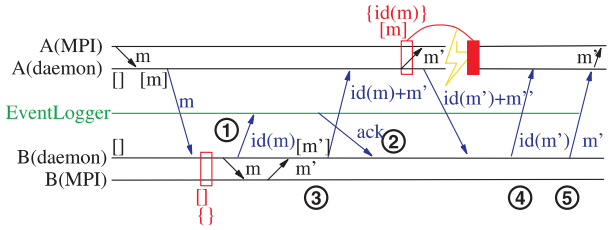


Fig. 5 Example of execution with MPICH-V/causal.

stores the last events, one per process, corresponding to emissions or receptions to/from a neighbor B. When sending to this neighbor B, and because there is a total order between the events generated by a single node, no event created by a same node and preceding the last event concerning this node has to be piggybacked. Note that causal information that a process has to piggyback in messages have to be included in checkpoint image, in addition to sender-based message payloads. ④ When a failure occurs, the failed process retrieves the causality information from the event logger, and also from all other nodes. ⑤ It then requests all messages to reexecute from the other nodes as in the pessimistic protocol of MPICH-V2.

Typical deployments of MPICH-V/causal use the same components as MPICH-V2 (Figure 2(b)).

3.4.1 Implementation notes on the MPICH-V framework MPICH-V/causal uses the same components as MPICH-V2. The event logger is different because in causal message logging protocols, in order to collect the unused memories of the node from causal information propagated through piggybacking, the nodes have to be informed about the events stored on a reliable event logger.

Two strategies can be used. The first one is to broadcast all acknowledge messages to all nodes. This is expected to have dramatic impact on performance. We implemented the second one: the event logger has been extended to send back the last stored clock for all receivers during the acknowledge protocol. This avoids broadcasting all acknowledge messages and requires using only one event logger.

3.5 MPICH-V/CL: A Coordinated Checkpoint Protocol

We also implemented a coordinated checkpoint protocol based on the Chandy-Lamport algorithm (Chandy and Lamport 1985). The main goal of this implementation was to compare such an approach with message logging protocols. The coordination of all checkpoints is ensured by the checkpoint scheduler and, in contrast to message logging deployments, where multiple checkpoint schedulers

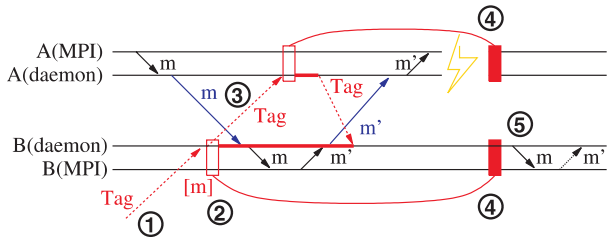


Fig. 6 Example of execution with MPICH-V/CL.

could be used, here only one checkpoint scheduler must be launched (Figure 2(c)).

The Chandy-Lamport algorithm relies on a wave of synchronization propagated through communication channels. Figure 6 presents an example execution of this protocol. ① When a process receives a checkpoint tag, it begins its checkpoint, and stores any message *m* incoming from another communication channel as an in-transit message until the checkpoint tag is received. ② All these in-transit messages are included in the checkpoint image. ③ It also sends a checkpoint tag to all neighbors to flush its own output channels. The checkpoint of a node is finished when all input channels have been flushed by a checkpoint tag.

Our protocol is an optimization of classical coordinated checkpoint, intended to reduce the stress of the checkpoint servers during the checkpoints and restarts. The optimization consists of storing the checkpoint image not only on a remote server, which is mandatory to ensure fault tolerance after a crash, but also on the local disks of computing nodes. This local storage allows a restart from the local checkpoint image for all non-failed processes, reducing the stress of the checkpoint server to only the load related to the restart of faulty processes. The evaluation section will highlight the benefits of this optimization, in term of tolerance to high fault frequency.

④ When a failure occurs, all processes are restarted from a checkpoint image belonging to the same coordinated checkpoint phase. All non-failed processes load their image from their local disk. Failed processes, restarted on other nodes, download their checkpoint images from their checkpoint servers. ⑤ In-transit messages are delivered from the checkpoint to the application.

3.5.1 Implementation notes on the MPICH-V framework The Chandy-Lamport algorithm does not need the use of an event logger, or channel memories. The deployment is thus slightly simpler.

MPICH-V/CL uses a checkpoint scheduler policy slightly different from the one used for the uncoordinated checkpoint protocols. The checkpoint scheduler maintains a global checkpoint sequence number. When requested by its policy (a periodic policy), it requests every process to checkpoint with this sequence number, acting as a computing node initiating a Chandy-Lamport coordinated wave. When a process successfully finishes its checkpoint, it sends the checkpoint tag to the checkpoint scheduler. Thus, the checkpoint scheduler can ensure that a global checkpoint is successful and notify every computing node to remove older remote and local checkpoint files. When a failure is detected, all nodes retrieve the global checkpoint sequence number from the checkpoint scheduler, and then restart from an image tagged by this number.

4 Performance Evaluation

We present a set of experiments to evaluate our frameworks in comparison with reference implementation, measure framework related overheads and compare the four protocols.

Other performance evaluations concerning the different components (channel memory, checkpoint servers, etc.) and impact of blocking and non-blocking checkpoint on the execution time are presented in our previous papers on MPICH-V (Bosilca et al. 2002; Bouteiller et al. 2003a, 2003c; Lemarinier et al. 2004).

4.1 Experimental Conditions

All measurements presented in this paper are performed on an Ethernet network. Other measurements, not presented in this paper, have been made for Myrinet and SCI networks (Lemarinier et al. 2004). Since MPICH-V1 is not adapted to these types of network, we restrict our comparison to an Ethernet network.

Experiments are run on a 32-node 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.

All these nodes were operating under Linux 2.4.20. The tests and benchmarks were compiled with GCC (with flag –O3) and the PGI Fortran77 compilers. All tests were run in dedicated mode. Each measurement was repeated 5 times and we present a mean of them.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. We used the NetPIPE (Snell, Mikler, and Gustafson 1996) 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 (NPB 2.3; Bailey et al. 1995), written by the NASA NAS research center to test high performance parallel machines.

For all the experiments, we considered 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 or parallel file systems), we consider that this system impacts the performance of checkpointing similarly for any fault-tolerant protocol.

4.2 Fault-Tolerant Framework Performance Validation

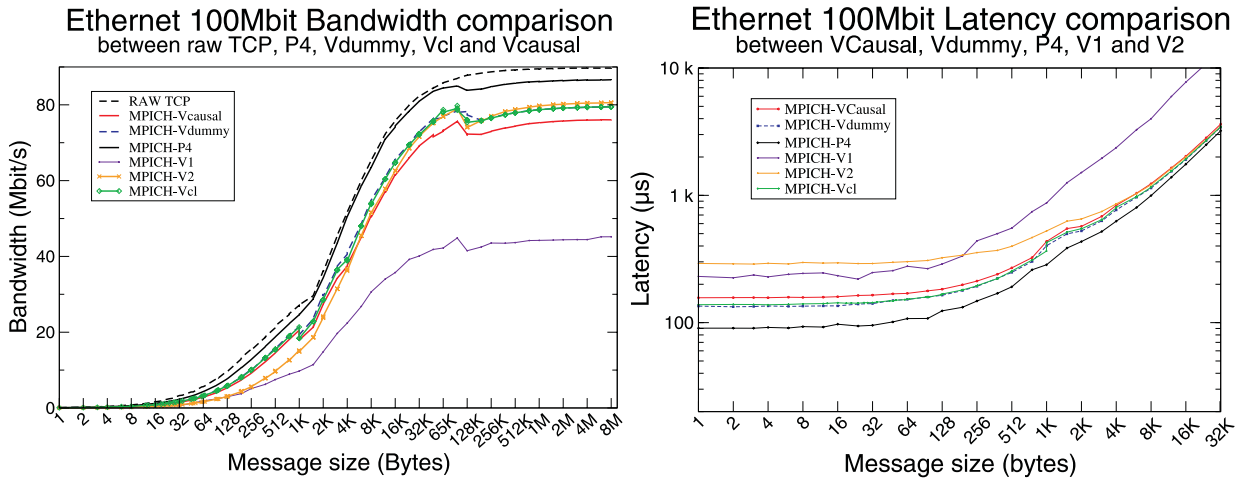
It is important to validate our fault-tolerant frameworks and to understand which overheads are framework related and which are protocol induced. In order to validate these points, we compared the performance of the MPICH-V framework without fault tolerance (MPICH-Vdummy) to the reference non-fault-tolerant implementation MPICH-P4. The architecture of the MPICH-V1 protocol, with stable channel memories, is intrinsically fault tolerant. In other words, there is no way to remove fault tolerance from MPICH-V1. Thus a comparison of the framework of MPICH-V1 without fault tolerance does not have any meaning. However, we compare the global overhead of the MPICH-V1 architecture to the one of MPICH-P4, since the framework overhead is closely related to the protocol architecture.

4.3 Protocol Performances

4.3.1 Performances without faults Figures 7(a) and 7(b) compare the bandwidth and the latency of the NetPIPE (Snell, Mikler and Gustafson 1996) ping-pong benchmark for the different protocols.

On the Ethernet network, the non-fault-tolerant protocol of the MPICH-V Framework, Vdummy, adds only a small overhead on bandwidth compared with P4. It adds a 30% increase in latency, because of the implementation of the communication between the channel interface and the daemon. This implementation adds delays related to context switch, and system calls before every actual emission of a message on the network. Since all our protocols use the same mechanism, the 30% increase can be considered as the framework overhead. Note that we plan to remove additional system calls and context switches from the channel to the daemon IPC mechanism in order to remove this overhead.

The latency experience (Figure 7(b)) was conducted up to 8Mb messages, and shows asymptotic behaviors for all protocols similar to the one at 32Kb: their latencies increase linearly from 32Kb messages and keep the same gaps. MPICH-V1 presents a high and constant multiplicative factor compared with P4, for small as well as for large messages. This is due to the communication scheme, which imposes every message to cross a channel memory. Although for large messages all other implementations perform evenly compared with P4, for small messages, all implementations behave differently. MPICH-V2 clearly demonstrates a higher latency for small messages, because of the acknowledge algorithm of the event logger. The diagram shows that the causal implementation solves the latency problem raised by the pessimistic one.



(a) Bandwidth comparison

(b) Latency comparison

Fig. 7 (a) Latency and (b) bandwidth comparisons of MPICH-P4, MPICH-V1, MPICH-V2, MPICH-Vcl and MPICH-V/causal.

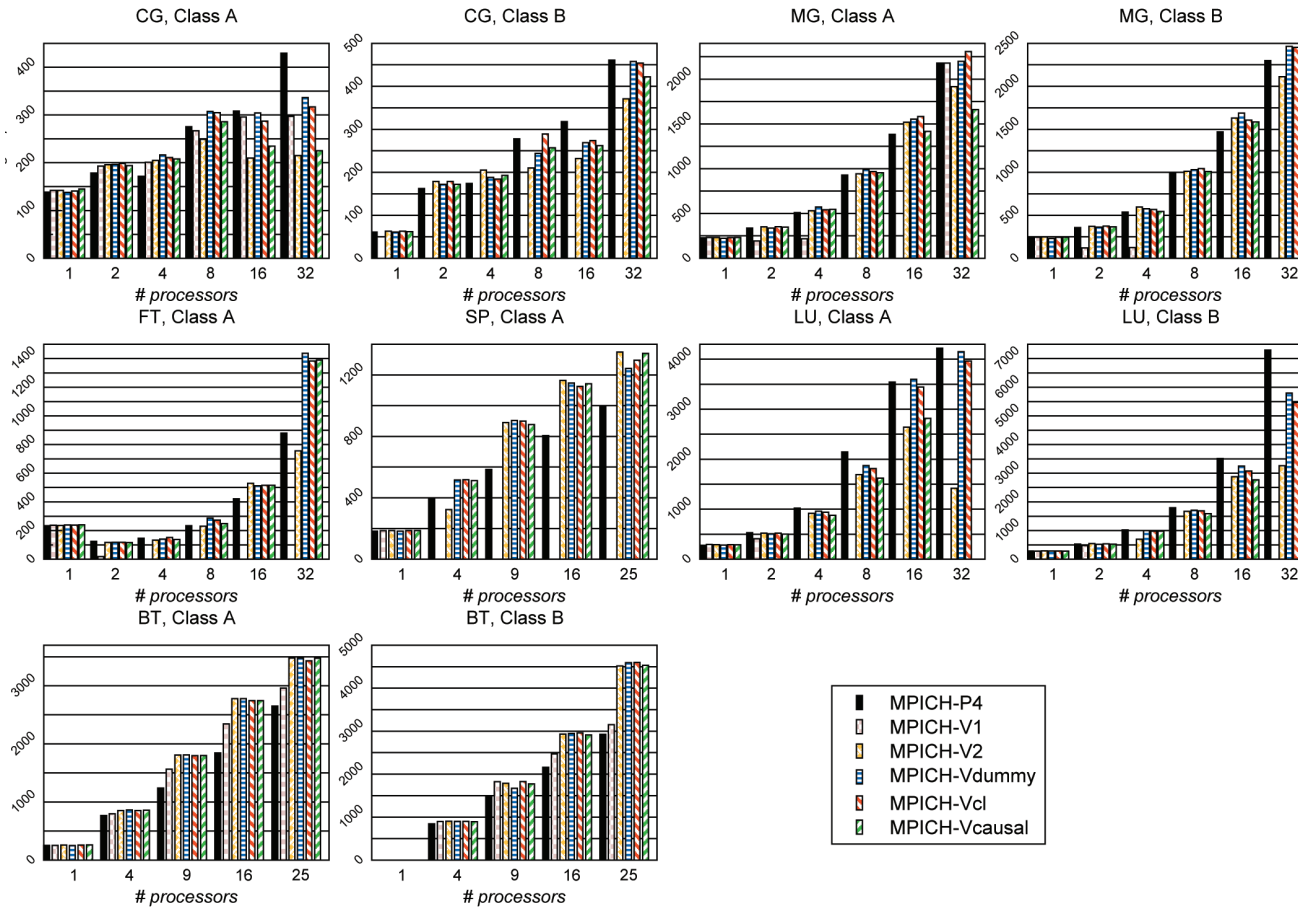


Fig. 8 Performance comparison of MPICH-P4, MPICH-V1, MPICH-V2, MPICH-Vcl and MPICH-V/causal for six of the NAS parallel benchmarks.

The bandwidth measurement (Figure 7(a)) shows a high communication cost for MPICH-V1. Since all communications have to pass through the channel memory, which is a remote computer, messages pass twice more on the network than for direct communication, thus dividing the observed bandwidth by two. The reduction is less for the other fault-tolerant protocols. MPICH-V/causal appends to each message a piggyback of causal information, which increases the global message size compared with the other protocols, thus decreasing slightly the observed bandwidth. The two other protocols do not introduce new performance reduction other than the reduction resulting from the MPICH-V framework implementation.

We compared the performance of the four fault tolerant protocols on the set of kernels and applications of the NAS parallel benchmark without checkpointing (Figure 8). The first protocol, MPICH-V1 was run completely on the CG and BT benchmarks. In order to provide fair comparison between the protocols, the experiment was run using one channel memory per set of four computing nodes. On other benchmarks, such a deployment produced a memory exhaustion of MPICH-V1, which clearly demonstrates a limitation of this protocol. The LU benchmark provides a high number of small communications. As expected, MPICH-V2, which has the highest latency, presents a high performance degradation compared with the two

other protocols (Causal and CL). All the NAS benchmarks demonstrate that MPICH-VCL and MPICH-V/causal reach performances similar to the ones obtained by the Vdummy protocol which provides no fault tolerance. On LU class B, the P4 reference implementation outperforms MPICH-V on 32 nodes, which results from our implementation of the channel interface in MPICH-V, while on BT class B, MPICH-V outperforms P4 because of the full duplex communication implemented in every protocol of MPICH-V. For BT, P4 uses only half duplex communications.

4.3.2 Performances with faults We evaluated the fault resilience of all protocols. Figure 9 presents a comparison of the overhead induced by an increasing fault frequency on the NAS benchmark BT between V1, V2, Vcausal and Vcl implementations.

Figure 9 shows the slowdown of execution time, in percent of BT class B running on 25 nodes, according to the fault frequency, and compared with the execution time of MPICH-VCL in a fault free context (label 0 in the x axis of the figure). We consider five fault-tolerant protocols: MPICH-V/causal, MPICH-V2, and two versions of MPICH-VCL: one where driver and daemon may keep a copy of checkpoint images on local disk, the other one where all checkpoint images are only kept on a remote checkpoint server. We ran the benchmark several times and introduced an increasing number of non-overlapping faults during the execution.

This figure clearly demonstrates the improvement of local copies of the checkpoint image for the classical

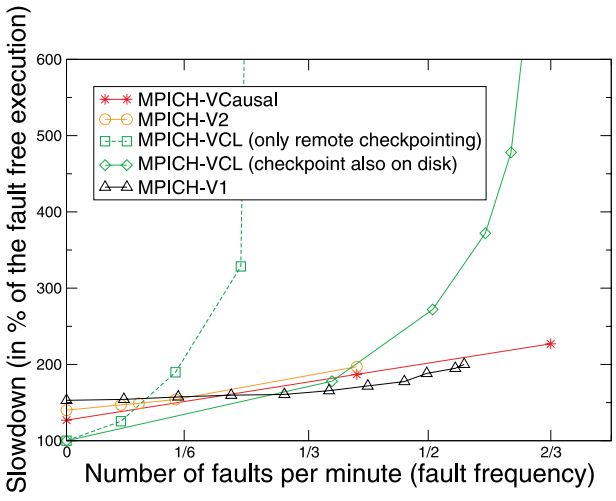


Fig. 9 Fault frequency impact on execution time of BT class B 25 nodes on Fast-Ethernet using five fault tolerance protocols.

Chandy-Lamport algorithm. When all processors are restarted, only the faulty one has to retrieve its checkpoint image from a remote checkpoint server. When only remote checkpointing is used, the checkpoint server becomes the bottleneck.

The two message-logging protocols show very similar behavior with respect to fault frequency. Despite the higher complexity for restarting failed processes, the causal protocol exhibits slightly better performances because of its higher performance between the faults.

For the remote message logging protocol, the fault-free execution shows a high overhead. As we used one channel memory for three computing nodes, the bandwidth of these channel memory are shared between client nodes. Note that this execution uses 30% supplementary nodes, assumed to be stable. However, the high fault resilience of the MPICH-V1 architecture is confirmed as it performs better than all other protocols for very high fault frequencies.

All protocols present different disruptive points from where the execution does not progress any further. The figure demonstrates that this disruptive point is reached by coordinated checkpoint protocols for lower fault frequencies than for message-logging protocols. This is due to the coordination of checkpoints, leading to simultaneous storage of all checkpoint images on the remote checkpoint server. If n is the number of processes to checkpoint, the total time for building a coherent cut in the system is n times higher than the time to save a single checkpoint image. When the delay between faults is lower than this time, the coordinated checkpoint protocol cannot progress anymore. Pessimistic message-logging protocols need only one successful checkpoint between two faults to ensure execution progression.

The crosspoint between coordinated checkpoint and message logging for our test is at one fault every three minutes. However, if we consider more realistic datasets, e.g. 1 GB memory occupation by all processes, then the checkpoint time would be around 48 minutes (according to a linear extrapolation of checkpoint time) for 25 nodes. In that case the minimum fault interval ensuring that the application still progresses is about one hour including all the protocol and restart overheads. Following the same extrapolation, if we consider a deployment with one checkpoint server per 250 nodes, which is likely to occur in very large platforms with ten thousand nodes or more, this crosspoint will become one fault every ten hours. This last value is typical MTBF for such very large platforms.

5 Conclusion and Future Work

In this article, we have presented several contributions: we have first introduced an extensive related work section presenting all previous research in the domain of message

passing distributed systems and especially in the context of a MPI environment. As a result of the lack of results comparing the two main classes of fault-tolerant protocols (global coordination and message logging), we have designed and implemented a generic framework for fault-tolerant protocols comparison. Using this framework, we have implemented three original and one optimization of a classical protocol. We have compared the merits of these different protocols in terms of performances on the NAS parallel benchmark and in terms of fault tolerance.

The main results of this work can be summarized as follows:

1. Direct communication between the computing nodes is mandatory for high performance on the NAS benchmark. This limits the use of pessimistic remote message logging protocol (MPICH-V1) to desktop grids where channel memories can be used as communication channels between computing nodes.
2. Remote synchronous pessimistic storage of causality information in message logging protocols (MPICH-V2) adds a latency overhead reducing significantly the performances for the latency sensitive NAS benchmarks.
3. Remote asynchronous pessimistic storage of causality information in message logging protocols (MPICH-V/causal) solves the latency problem, but reduces the observable bandwidth because of the piggyback of causality information in all exchanged messages.
4. Contrary to general belief, coordinated checkpoint provides very good performance compared with message logging in fault-free executions but also in the presence of faults. The synchronization time is not the primary limiting factor of this kind of protocol. The main performance degradation results from the stress of the checkpoint server during checkpoints and restarts. The stress from restarts can be solved by using local copies of checkpoint images.
5. The stress of the checkpoint server in coordinated checkpoints is the main differentiating factor compared with message-logging protocols which provide better fault tolerance for high fault frequencies.

Several issues remain unexplored by this work. On the one hand they correspond to potential usages of the developed protocols and on the other hand they consist of their improvement.

We are currently working on using the presented protocols to provide a novel approach for time sharing the cluster resources between several MPI executions. In the same research we are studying the performance of MPI execution migration between clusters using heterogeneous networks and the impact of migration on the execution time.

In order to improve the proposed protocols, we will compare their scalability on larger clusters. We will also develop zero copy implementations of these protocols for high bandwidth and low latency networks and study their respective impact on performance. With these two elements we will be able to investigate the performance cross-point of these protocols on high speed networks according to the fault frequency, on large scale clusters.

Another research direction has been presented in Bouteiller et al. (2003b): hierarchical fault tolerance in the context of Grid. We have designed and started the implementation of MPICH-V3 based on the components described in this paper and an original fault-tolerant protocol composed of an augmented version of MPICH-V/CL and one of the message logging protocols. The selection of the best message logging protocol for this purpose is an issue that will be addressed by this research.

Author Biographies

Aurélien Bouteiller is a PhD student in the Cluster and Grid group of the LRI laboratory at Paris-South University and is a member of the Grand-Large team of INRIA. He has obtained a Master in parallel computer science in 2002 from the french Paris-South university. He contributes to the MPICH-V project, a fault-tolerant MPI implementation comparing different fault-tolerant protocols. His research interests include high performance fault-tolerant MPI and checkpoint optimizations and performance evaluation.

Franck Cappello holds a Research Director position at INRIA, after having spent 8 years as CNRS researcher. He leads the Grand-Large project at INRIA and the Cluster and Grid group at LRI. He has authored more than 50 papers in the domains of High Performance Programming, Desktop Grids and fault-tolerant MPI. He is editorial board member of the "International Journal on GRID Computing" and steering committee member of IEEE/ACM CCGRID. He organises annually the Global and Peer-to-Peer Computing workshop. He has initiated and heads the Xtrem-Web (Desktop Grid) and MPICH-V (Fault tolerant MPI) projects. He is currently involved in two new projects: Grid eXplorer (a Grid Emulator) and Grid5000 (a Nation Wide Experimental Grid Testbed).

Thomas Herault is Associate Professor at the Paris South University. He defended his PhD on the mending of transient failure in self-stabilizing systems under the supervision of Joffroy Beauquier. He is a member of the Grand-Large INRIA team and works on fault-tolerant protocols in large scale distributed systems. He contributes to the

MPICH-V project, and to the APMC project on automatic and approximate probabilistic model checking of probabilistic distributed systems.

Géraud Krawezik has studied as a PhD student in the Cluster and Grid group at the University of Paris South, in Orsay, under the guidance of Franck Cappello. He is interested in High Performance Computing standards (MPI, OpenMP), and MPI volatile implementations (MPICH-V). He also worked with Professor Marc Snir on new parallel languages paradigms. He graduated in 2000 as an engineer in computer science and electronical design from the French School "ENSIETA".

Pierre Lemarinier has obtained a Master in computer science in 2002 from the french Paris-South University. He is a PhD student in the parallelism team and the Grand-Large team of the LRI laboratory of Paris-South. His research interests include fault-tolerant protocols conception and validation and MPI implementations (MPICH-V).

References

Agbaria, A. and Friedman, R. 1999. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *8th International Symposium on High Performance Distributed Computing (HPDC-8 '99)*. Los Alamitos, CA: IEEE CS Press.

Alvisi, L., Elnozahy, E., Rao, S., Husain, S. A., and Mel, A. D. 1999. An analysis of communication induced checkpointing. *29th Symposium on Fault-Tolerant Computing (FTCS'99)*. Los Alamitos, CA: IEEE CS Press.

Alvisi, L. and Marzullo, K. 1995. Message logging: Pessimistic, optimistic, and causal. *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pp. 229–236. Los Alamitos, CA: IEEE CS Press.

Bailey, D., Harris, T., Saphir, W., Wijngaart, R. V. D., Woo, A., and Yarrow, M. 1995. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center.

Batchu, R., Neelamegam, J., Cui, Z., Beddhua, M., Skjellum, A., Dandass, Y., and Apte, M. 2001. MPI/FT^{TM}: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. *Proceedings of the 1st International Symposium of Cluster Computing and the Grid (CCGRID2001)*, Melbourne, Australia. IEEE/ACM.

Bhatia, K., Marzullo, K., and Alvisi, L. 1998. The relative overhead of piggybacking in causal message logging protocols. *17th Symposium on Reliable Distributed Systems (SRDS'98)*, pp. 348–353. Los Alamitos, CA: IEEE CS Press.

Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fédak, G., Germain, C., Hérault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Néri, V., and Selikhov, A. 2002. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. *High Performance Networking and Computing (SC2002)*, Baltimore USA, IEEE/ACM.

Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., and Magniette, F. 2003a. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. *High Performance Networking and Computing (SC2003)*, Phoenix USA, IEEE/ACM.

Bouteiller, A., Lemarinier, P., and Cappello, F. 2003b. MPICH-V3 preview: A hierarchical fault tolerant MPI for multi-cluster grids. *IEEE/ACM High Performance Networking and Computing (SC 2003)*, poster session, Phoenix USA.

Bouteiller, A., Lemarinier, P., Krawezik, G., and Cappello, F. 2003c. Coordinated checkpoint versus message log for fault tolerant MPI. *IEEE International Conference on Cluster Computing (Cluster 2003)*. Los Alamitos, CA: IEEE CS Press.

Burns, G., Daoud, R., and Vaigl, J. 1994. LAM: An Open Cluster Environment for MPI. *Proceedings of Supercomputing Symposium*, pp. 379–386.

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

Chen, Y., Li, K., and Planck, J. S. 1997. CLIP: A checkpointing tool for message-passing parallel programs. *High Performance Networking and Computing (SC97)*. IEEE/ACM.

Elnozahy, E. N. and Zwaenepoel, W. 1992a. Replicated distributed processes in Manetho. *22nd International Symposium on Fault Tolerant Computing (FTCS-22)*, Boston, MA. Los Alamitos, CA: IEEE CS Press.

Elnozahy, E. N. and Zwaenepoel, W. 1992b. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers* 41(5).

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

Fagg, G. and Dongarra, J. 2000. FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world. *7th Euro PVM/MPI User's Group Meeting 2000*, vol. 1908, Balatonfüred, Hungary. Heidelberg: Springer-Verlag.

Fagg, G. E., Bukovsky, A., and Dongarra, J. J. 2001. HARNESS and fault tolerant MPI. *Parallel Computing* 27(11): 1479–1495.

Gropp, W. and Lusk, E. 2004. Fault tolerance in MPI programs. Special issue of the *Journal High Performance Computing Applications (IJHPCA)* 18(3): 363–372.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6):789–828.

Johnson, D. B. and Zwaenepoel, W. 1987. Sender-based message logging. *The 17th Annual International Symposium on Fault-tolerant Computing (FTCS'87)*. Los Alamitos, CA: IEEE CS Press.

Juang, T. T.-Y. and Venkatesan, S. 1991. Crash recovery with little overhead. *11th International Conference on Distributed Computing Systems (ICDCS'11)*, pp. 454–461. Los Alamitos, CA: IEEE CS Press.

Lee, B., Park, T., Yeom, H. Y., and Cho, Y. 1998. An efficient algorithm for causal message logging. *17th Symposium on Reliable Distributed Systems (SRDS 1998)*, pp. 19–25. Los Alamitos, CA: IEEE CS Press.

Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., and Cappello, F. 2004. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. *IEEE International Conference on Cluster Computing (Cluster 2004)*. Los Alamitos, CA: IEEE CS Press.

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

Louca, S., Neophytou, N., Lachanas, A., and Evripidou, P. 2000. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters(PPL)* 10(4). World Scientific Publishing Company.

Planck, J. S. and Thomason, M. G. 2001. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing* 61(11): 1570–1590.

Plank, J. S. and Elwasif, W. R. 1998. Experimental assessment of workstation failures and their impact on checkpointing systems. *28th Symposium on Fault-Tolerant Computing (FTCS'98)*, pp. 48–57. Los Alamitos, CA: IEEE CS Press.

Pruitt, P. N. 1998. *An Asynchronous Checkpoint and Rollback Facility for Distributed Computations*. PhD thesis, College of William and Mary in Virginia.

Rao, S., Alvisi, L., and Vin, H. M. 1998. The cost of recovery in message logging protocols. *17th Symposium on Reliable Distributed Systems (SRDS)*, pp. 10–18. Los Alamitos, CA: IEEE CS Press.

Rao, S., Alvisi, L., and Vin, H. M. 1999. Egida: An extensible toolkit for low-overhead fault-tolerance. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, pp. 48–55. Los Alamitos, CA: IEEE CS Press.

Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. 2003. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA.

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

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

Stellner, G. 1996. CoCheck: Checkpointing and process migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii. Los Alamitos, CA: IEEE CS Press.

Strom, R. and Yemini, S. 1985. Optimistic recovery in distributed systems. *Transactions on Computer Systems* 3(3):204–226. ACM.

Strom, R. E., Bacon, D. F., and Yemini, S. A. 1988. Volatile logging in n-fault-tolerant distributed systems. *18th Annual International Symposium on Fault-Tolerant Computing (FTCS-18)*, pp. 44–49. Los Alamitos, CA: IEEE CS Press.

Wong, K. F. and Franklin, M. A. 1993. Distributed computing systems and checkpointing. *2nd International Symposium on High Performance Distributed Computing (HPDC'93)*, pp. 224–233. Los Alamitos, CA: IEEE CS Press.