MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging

Aurélien Bouteiller[*], Franck Cappello[* †], Thomas Hérault[*],
Géraud Krawezik[*], Pierre Lemarinier[*], Frédéric Magniette[*]

[*]*LRI, Université de Paris Sud, Orsay, France*
[†]*INRIA Futurs, Saclay, France*
{bouteill,fci,herault,gk,lemarini,magniett}@lri.fr
URL: http://www.lri.fr/ ˜gk/MPICH-V

Abstract

Execution of MPI applications on clusters and Grid deployments suffering from node and network failures motivates the use of fault tolerant MPI implementations.

We present MPICH-V2 (the second protocol of MPICH-V project), an automatic fault tolerant MPI implementation using an innovative protocol that removes the most limiting factor of the pessimistic message logging approach: reliable logging of in transit messages. MPICH-V2 relies on uncoordinated checkpointing, sender based message logging and remote reliable logging of message logical clocks.

This paper presents the architecture of MPICH-V2, its theoretical foundation and the performance of the implementation. We compare MPICH-V2 to MPICH-V1 and MPICH-P4 evaluating a) its point-to-point performance, b) the performance for the NAS benchmarks, c) the application performance when many faults occur during the execution. Experimental results demonstrate that MPICH-V2 provides performance close to MPICH-P4 for applications using large messages while reducing dramatically the number of reliable nodes compared to MPICH-V1.

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, and current large Grid deployments (TERA Grid, J-Grid, DEISA, etc.), campus/company wide

desktop Grids are examples of such infrastructures. In all these infrastructures, node and network faults are likely to occur, obliging the use of a programming model allowing fault management and/or a fault tolerant runtime.

Users of these platforms are very familiar with explicit message passing and their applications often use MPI [21] as the message passing library. If the need for fault tolerant MPI implementations is well accepted, the way how faults should be managed is still an open issue [14]: a) the programmer of the application may save periodically intermediate results on reliable media during the execution in case of an entire restart, b) the functions of the MPI implementation may be augmented to return information about faults and accept communicator reconfiguration [13] or c) the MPI implementation hides the faults to the programmer and the user by providing a fully automatic fault detection and recovery. The latter approach, while very interesting for the end user, suffers either from limited fault tolerant capabilities or high resource cost. Examples of such automatic fault tolerant MPI implementations are based on the optimistic or causal message logging approach. While in theory these protocols may tolerate any number of faults if augmented by appropriate mechanisms, none of their existing implementation tolerates more than one fault, involving the restart of the full system in case of multiple faults. Examples of automatic n-faults tolerant MPI protocols that tolerate n concurrent faults of MPI process, n being the total number of MPI processes, follow the pessimistic message logging principle (storing all in transit messages on reliable media) and thus require a large number of non computational reliable resources.

MPICH-V is a research effort with theoretical studies, experimental evaluations and pragmatic implementations aiming to provide a MPI implementation based on MPICH[15], featuring multiple fault tolerant protocols. In this paper we present the second protocol for MPICH-V

called MPICH-V2 associating the low additional resource cost of sender based message logging and the capacity to tolerate n concurrent faults of the pessimistic message logging strategy. (note: for the rest of the paper, MPICH-V1 stands for the first version of MPICH-V presented in [5]). This original protocol still follows uncoordinated checkpoint, distributed message logging and uses a reliable coordinator and checkpoint servers. The features of MPICH-V2 make it attractive for a) large clusters, b) clusters made from collection of nodes in a LAN environment (Desktop Grid), c) Grid deployments harnessing several clusters and d) campus/industry wide desktop Grids with volatile nodes (i.e. all infrastructures featuring synchronous network or controllable area network). This paper presents the key principles of MPICH-V2, its global architecture, the theoretical foundations of the protocol, the architecture of its components and the evaluation of its performance on the NAS benchmark compared to MPICH-V1 and MPICH-P4. Following our expectations, MPICH-V2 reaches a performance close to the one of MPICH-P4 and still features the same fault tolerance properties of MPICH-V1 while requiring much less reliable resources.

2 Challenges for Scalable Automatic/Transparent Fault Tolerant MPI

Building a scalable automatic/transparent fault tolerant MPI implementation means finding solutions for several difficult issues.

MPI Process Volatility Tolerance In large scale distributed systems like large clusters, Grid deployments and Desktop Grids, any number of nodes may leave the system at any time. Thus, there are two challenges: 1) survive massive lost of nodes and 2) high frequency of faults. An example of massive lost of nodes in a Grid infrastructure is when all the nodes of a cluster disconnect the system due to a network connection failure between the cluster and the rest of the Grid. Note that conversely, a cluster may join the Grid and continue the execution of the lost MPI processes. An example of high fault frequency is the large Desktop Grids where nodes may join/leave the system independently and unpredictably. A volatility tolerant MPI implementation should be able to detect the failure, rollback the appropriate MPI process possibly on a different node, and continue the execution independently of the fault number and frequency. This means that the fault tolerance system should itself survive node failure during the repair protocol.

MPI Implementation Independence Designing a fault tolerant protocol/runtime for MPI is a strong effort which should work independently of any particular specificities of an MPI implementation. This is a very important issue because, the protocol should work for a large variety of MPI implementations, the protocol integration within MPI should not compromise the design and the semantic of the implementation (how global operations are implemented, multi-protocols point-to-point communication), the protocol should be easily adaptable to new MPI implementation versions. The direct implication of these requirements is a minimum modification of the MPI implementation. For example, in MPICH-V, the MPICH implementation is not aware of fault and recovery events.

Scalable Architecture In fault tolerant distributed systems, scalability concerns 1) the protocol requirements in terms of synchrony and message complexity and 2) the number of reliable nodes. Checkpoint and restart protocols should not rely on global synchronization simply because some nodes may leave the system during the synchronization. Redundancy of MPI processes would involve an active or passive replication strategy. These strategies require an atomic broadcast for each message which is proved to be reducible to the consensus problem. While removing completely the need of reliable nodes is very difficult, a scalable design should limit their number. If the MPI implementation is to tolerate n concurrent faults (n being the number of MPI processes), then a reliable coordinator and a set of reliable remote checkpoint servers should be used. The design of such reliable nodes would typically use the active or passive replication strategy.

Nondeterministic Reception Order Some MPI low level control messages as well as the user level API allow nondeterministic reception order at the receiver side. For the execution correctness after failures, internal task events and task communications of restarted tasks should be replayed in a consistent way according to the non failed tasks. The execution is considered consistent if after the failure, the execution is one of the possible executions without failure. Thus, a mechanism should be designed to record the reception order at the receiver side and force the replayed receptions, after a failure, in the order occurred before the failure. Forcing this property should not lead to compute a global value (consensus) or to force a restarted process to contact all other processes of the application to get the messages to be received in a correct order.

3 Related Work

Fault tolerance for explicit message passing in distributed system has been investigated by many research projects. Many techniques are presented and discussed in [10] and [20]. The first paper about MPICH-V [5] presents a classification of fault tolerant MPI environments, distinguishing two axes: the fault tolerant technique and the level of the software stack where the fault tolerance is managed.

Some other works to make MPI fault tolerant exist. MPI/FT [4] considers task redundancy to provide fault tol-

erance. It uses a central coordinator that monitors the application progress, logs the messages, manages control messages between redundant tasks and restarts failed MPI process. FT-MPI [12, 13] handles failures at the MPI communicator level and lets the application manage the recovery. When a fault occurs, all MPI processes of the communicator are informed about the fault. This information is transmitted to the application through the returning value of MPI calls. The main advantage of FT-MPI is its performance since it does not checkpoint nor log, but its main drawback is the lack of transparency for the programmer.

In this paper we focus on automatic/transparent fault tolerance techniques.

3.1 Automatic/Transparent Fault Tolerance in Distributed Systems

There are several ways to provide automatic/transparent fault tolerance in distributed systems. The main approaches are crash and recovery protocols that consist in restarting a set of processes when a failure occurs. The theoretical foundation of these protocols considers that a distributed execution can be entirely described by the state of all distributed processes plus the in-transit messages. Two main techniques are used for saving the distributed execution state and recovering from systems failures: coordinated checkpoint and uncoordinated checkpoint associated with message logging.

The first kind of techniques computes a snapshot of the distributed execution. In all cases, the process states are saved in reliable media. Several algorithms have been proposed for saving the in-transit messages. In the Connection Machine CM5[16], the state of all network routers, including the in-transit messages is saved. Other algorithms remove the in-transit messages from the network (stopping message injection and waiting for the reception of all sent messages) and save them along with the process images. The most known algorithm of this family is the one proposed by Chandy and Lamport [6]. Coordinated checkpoint involves the rollback of all processes from the last snapshot when a faulty situation is detected, even when a single process crashes. Cocheck [22], Starfish [1] and Clip [7] are fault tolerant MPI based on coordinated checkpoint.

Uncoordinated checkpoint protocols allow all processes to execute a checkpoint independently of the others. The theoretical foundation of this technique considers that a process execution (sequence of state changes) in a distributed execution is determined only by its message receptions[23]. Thus this technique relies on message logging in addition to process checkpointing to ensure the complete description of a process execution state in case of its failure. Message logging protocols consist in 1) logging all received messages and 2) re-sending the same relevant messages, in the same order, to the crashed processes during their re-execution.

This principle provides the guaranty that a re-executed process starting from a previous correct state (the beginning of the execution or a consistent checkpoint image) will reach a state matching the rest of the system, as before the crash. There are three kinds of message logging protocols : optimistic, pessimistic and causal. Pessimistic protocols ensure that all messages received by a process before it sends information in the system are logged on reliable media so that they can be re-sent later and only if necessary during rollback. Optimistic protocols just ensure that all messages will eventually be logged. So one usual way to implement optimistic logging is to log the messages on non-reliable media. Finally causal protocols log message information of a process in all causally dependent processes. Property formalism of those three techniques can be found in [2] and different crash and recovery protocols are listed in [10]. MPICH-V2 is based on uncoordinated checkpoint associated with pessimistic message logging.

3.2 Fault Tolerant MPI Implementations Based on Message Logging

MPICH-V1[5] is the first protocol of MPICH-V. It has been designed to tolerate a high node volatility. It is based on uncoordinated checkpointing and pessimistic message logging protocol storing all communications of the system on reliable media. To ensure this property, every computing processes is associated with a reliable process called Channel Memory. Every communication sent to a process is stored and ordered on its associated Channel Memory. To receive a message, a process sends a request to its associated Channel Memory. After a crash, a re-executing process retrieves all lost receptions in the correct order by requesting them to its Channel Memory. A main property of MPICH-V1 is the uncoordinated restart: a process re-execution is independent of the other processes of the system.

The use of Channel Memory has a major impact on the performance (dividing the bandwidth by a factor of 2) and the cost of the fault tolerance system (high performance requires a large number of Channel Memories). The main motivation for MPICH-V2 is to remove the need of Channel Memories in the perspective of reducing the overhead and cost of fault tolerance.

MPI-FT[18] is based on message logging protocol. It uses a special entity called Observer. This process is supposed reliable. It checks the aliveness of all MPI peers and re-spawns crashed ones. Messages can be logged following two different approaches. The first way 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.

Manetho[8, 9] follows a causal logging protocol. It logs

3

the messages locally to the sender and adds in all MPI communications the antecedence graph, mainly describing the partial order of all communications of the execution. Thus either the information about a message to be replayed can be given by an alive process or all processes depending on that message have crashed. In this last case, there is no need to force the same execution, since all dependent processes will play another equivalent execution. The main difference between MPICH-V2 and Manetho, separating them in different classes of protocol, is that information -about receptions- is logged on a reliable medium rather than appended to the messages.

Egida [20] is based on a language specifying the rollback recovery protocol mechanism to implement. A library has been integrated with MPICH[15] over P4's lower layer. This library provides basic components needed in general rollback recovery protocols such as information sending to reliable media. The programmer/user can re-implement some components and specify the protocol. All the send-receive calls of MPICH are replaced by those of the library. Contrary to this work, we only replace the P4 driver by another one and do not change anything else in MPICH.

A protocol close to MPICH-V2 is presented in [11]. While this study and MPICH-V2 shared the same concepts, the study provides no architecture principle, theoretical foundation, implementation detail, performance evaluation and merit comparison against non fault tolerant MPI and other fault tolerant MPI implementations.

4 MPICH-V2 Architecture

MPICH-V2 is based on an original pessimistic sender-based message logging protocol. The current implementation and its components are described in this section.

4.1 The Pessimistic Sender-Based Message Logging Protocol

The context of non coordinated checkpoints implies the use of message logging protocols. They are used to ensure that every restarted process will reach a state consistent with the rest of the system, before contributing to it again. On this purpose, any logging protocol keeps a trace of events happening on every process, in order to replay the logged events during re-execution.

We consider the following model of failure: a process may stop its execution at any point, then after a finite time, restarts in any arbitrary state already reached. Since there are mechanisms to ensure the completeness of messages, we assume that a message is always completely received or not at all if the sender or the receiver crashes during the transfer.

Under a piecewise deterministic assumption (every event following a logged event is deterministic or logged [23]),

it is proven that if every non deterministic event occurs in the same causal order during the re-execution, the system reaches a configuration (description of the complete state of the system) accessible by a fault-free execution.

Roughly speaking, pessimistic logging protocols are defined as logging protocols ensuring that every logged event is retrievable whatever are the failures. In typical MPI processes, non deterministic events are only message receptions.

The MPICH-V2 protocol is pessimistic sender-based: it keeps a copy of each message payload at the sender side (potentially volatile), and logs some causality information on a reliable media.
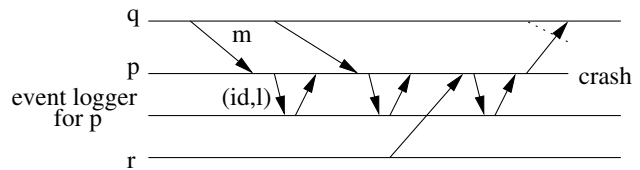
Figure 1. Protocol in execution phase for p

Each time a process sends a message, or receives one, it increases a local logical clock. Every message m sent from q to p has a unique identifier id (see figure 1). A copy of m is kept in q; when p receives m, it delivers m to the MPI process, then logs (id, l) on a reliable media, where l is its logical clock. Nevertheless, the process p is not allowed to send a message (and thus to have an effect on the system) before being ensured that the message is correctly logged (through the use of acknowledge messages).
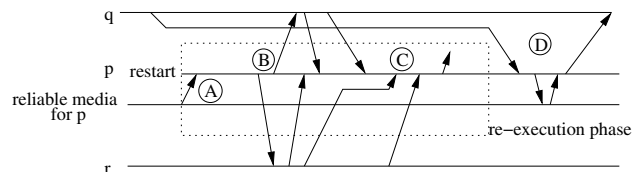
Figure 2. Protocol in re-execution phase for p

When a process p crashes, after a finite time, it is restarted in any previous state it reached (see figure 2). It retrieves then its set of logged couples (id, l) (phase A), and asks every other process to start again to send messages from the oldest identified one of the set (phase B). Note that the neighbor processes are not rolled back, they just send again their copy of the old messages as they concurrently continue their normal execution. The couples (id, l) and the logical clocks are used to determine which messages are to be replayed, in which order, and which are to be discarded (phase C). Then process p reaches back the crash point, and the execution continues normally.

For efficiency reasons, keeping a backup of every message during the whole execution is not reasonable in terms of storage usage. As soon as a process has successfully

4

checkpointed its state, it will never need the other processes to send the messages received before the checkpointed state again. The protocol may use a garbage collector to free the storage device of the corresponding copies.

When a process crashes, it restarts from the last successful checkpoint. If another process also crashes, and restarts earlier in its history than the first one, it will require old messages from the first. One solution would be to rollback the first process in a state preceding the emission of the required messages. However, this would directly lead to the well known domino effect. To avoid this situation, the first process has to restart with the copy of old messages, which are thus to be included in the checkpoints.

The appendix A presents a formalization of this protocol, used to prove its fault tolerance property.

4.2 Theoretical Foundations

In [23], the authors have demonstrated the fault tolerance property of pessimistic message logging protocols. We prove here that MPICH-V2 communication protocol falls into this category. In order to demonstrate the correctness of the protocol, we define the following notions, using classical definitions. A system is a collection of processes, connected by communication links. Each process follows an algorithm; they can exchange information with connected processes by sending or receiving a message. The union of the algorithms for every process defines a communication protocol. An atomic step is the application of an algorithm rule, or a failure. Failures are defined as the rollback of one process to any previously reached state. To each state, a process associates a unique logical clock. A configuration describes the state of every component of the system. A transition is the simultaneous application of an atomic step per process, for a subset of the processes leading one configuration into another. An execution is an alternate sequence of configurations and transitions.

Definition 1 ($Depend(m)$) *Let P be a communication protocol, and E an execution of P. Let C be a configuration of E and m a message of E. $Depend_C(m)$ is the set of processes that causally depend on the reception of m in C.*

Definition 2 (Re-Executable message) *Let m be a message received by q in the atomic step a_m, leading to state s. Let c be the logical clock associated to s.*

m is re-executable if c is logged on reliable media, and if m is retrievable from the sender.

Definition 3 (Pessimistic Logging protocol) *Let P be a communication protocol, and E an execution of P with at most f concurrent failures. Let M_C denotes the set of messages transmitted between the initial configuration and the configuration C of E.*

P *is a pessimistic message logging protocol if and only if*

$$\forall C \in E, \forall m \in M_C,$$
$$(|Depend_C(m)| > 1) \Rightarrow Re - Executable(m)$$

Theorem 1 *Let P be a communication protocol, and E be an execution of P, if P is a pessimistic logging protocol, E is equivalent to a fault-free execution.*

This theorem is proved in [23] using another model. We now use this result as a foundation to demonstrate that the MPICH-V2 protocol is fault tolerant.

Theorem 2 *The protocol of MPICH-V2 is a pessimistic message logging protocol.*

The full proof and the complete model of this theorem are given in appendix A and B.

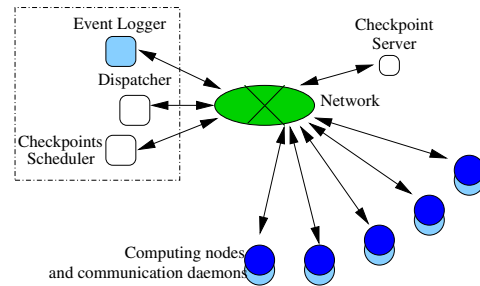## 4.3 Overview of the MPICH-V2 Architecture

Figure 3. Typical setup of a MPICH-V2 system

MPICH-V2 implements the pessimistic sender-based protocol on top of MPICH 1.2.5, using a dispatcher, a checkpoint scheduler, some event loggers, checkpoint servers, computing nodes and their communication daemons. The figure 3 presents a typical setup of a running MPICH-V2 system, where the dispatcher, the event logger and the checkpoint scheduler seat on the same computer.

Note that on a theoretical point of view, the checkpoint scheduler and the checkpoint servers may be unreliable. In the case where such a component fails, the computing nodes requiring checkpoint images will not be served by the failed checkpoint components and may restart from scratch, at worst. However a typical setup would execute the checkpoint scheduler on the same node as the dispatcher and the event logger, which is the single node in the system that must be reliable.

These architectural concepts inherit from the MPICH-V1 concepts, but are improved to reduce the cost of communications and the number of reliable components. Figure 4 compares the architectures of MPICH-V1 and MPICH-V2.

The main difference between MPICH-V1 and MPICH-V2 is the amount of reliable components required for reasonable performances. In MPICH-V1, all the messages are
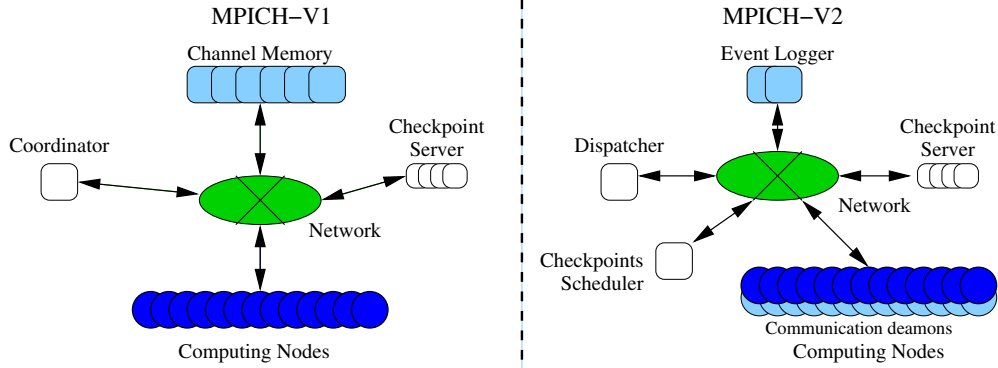
Figure 4. Architectural comparison of MPICH-V1 and MPICH-V2.

stored in Channel Memories, involving a large number of reliable media. In MPICH-V2, the message logging is split in two parts: on the one hand, the message data is stored on the computing node, following a sender-based approach. On the other hand, the corresponding event (the date and the identifier of the message reception) is stored on an event logger which is located on a reliable machine. The amount of information stored on the Event Logger is proportional to the number of transmitted messages and not proportional to the size of the payload like in MPICH-V1.

MPICH-V1 requires a complex communication scheme: every message has to transit through the Channel Memory. This involves two serialized TCP streams and increases the communication time, doubling the communication traffic over the network. In MPICH-V2, a message transit involves two kinds of communications: a direct TCP stream from the sender Computing Node to the receiver and a small message (in the order of 20 bytes) to the Event Logger. Thus, the communication time is expected to be better than the one of MPICH-V1.

Compared to MPICH-V1, MPICH-V2 requires an additional component: the checkpoint scheduler. In MPICH-V1, the decision of checkpointing was periodically triggered by a local timer on each node. In MPICH-V2, checkpoints do not need more coordination than in V1, but are scheduled for efficiency reasons.

4.4 MPI Process

MPICH is a layered implementation of MPI described in [15]. The MPI API is implemented by high-level primitives of the Abstract Device Interface (ADI). The ADI is implemented over another layer: the protocol layer which implements the short, eager and rendez-vous protocols. At last, these protocols are implemented over very basic primitives: the channel interface.

MPICH-V2 is implemented as a channel for MPICH: it implements a set of six primitives used by the protocol layer. The channel includes two communication func-

tions PIbrecv and PIbsend which are blocking operations for receiving or sending a block of data. It includes four other functions: PInprobe to check if a message is pending; PIfrom to get the identifier of the last message sender; PIiInit to initialize the channel and PIiFinish to finish the execution.

As in the P4 channel, the reference implementation for TCP/IP, the MPI process does not connect directly to all the other computing nodes. This is the job of a communication daemon running on the same machine and which is connected to the MPI process and handles the asynchrony of the network. This communication daemon establishes a UNIX socket and then spawns the MPI process. There are two kinds of messages exchanged along this socket: control messages (for init, finish and probe) and protocol messages (bsend, breceive).

The daemon is basically a select loop: it handles one socket for every computing node and one socket for every server (event logger, checkpoint server and checkpoint scheduler). These sockets are TCP streams and every send or receive operation is asynchronous. Thus, a communication is not blocked by another slower one. At contrary, the communication across the UNIX socket to the MPI process is synchronous and its granularity is the whole protocol message.

4.5 The Logging System

The sender based pessimistic message logging protocol of MPICH-V2 assumes that the logging of messages is split in two parts. One part uses a sender based logging method storing the messages payload on a non reliable media. The other part (the event logger) is used to store dependency information associated to these messages and must be run on a reliable system.

As described in the architecture section, each process increments a local logical clock when it sends or receives a message. The message payload logging system is integrated into the communication daemon located on the Computing

6

Node. Every time a message is sent to a computing node, it is stored locally in a list for further usages (sender based). Moreover the value of the sender logical clock is stored with the message copy.

Because of the non-reliability of the computing nodes, all this information is lost in case of a crash, but will be reconstructed if necessary during the re-execution.

The event logger is a repository executed on a reliable component of the system. It stores and delivers dependency information about messages exchanged by the computing nodes. The dependency information is composed of four fields associated to every received message: (sender's identity; sender's logical clock at emission; receiver's logical clock at delivery; number of probes since last delivery).

When a process receives a message in a non faulty execution, it aggregates the sender's identity and the sender's logical clock, found in the core of the message, with its own logical clock on delivery and the number of unsuccessful probes since the last delivery. In MPICH the upper layer can probe the existence of messages to be received, in order to implement non-blocking operations on top of blocking communication routines. We assume that a reception always follows a successful probe. The number of probes made since the last reception influences the next reception, so the receiver counts this number to add it to the dependency information, in order to replay exactly the same execution.

This information is collected during receptions of messages and sent asynchronously to the event logger. However, this information must be sent and acknowledged by the event logger before the node can modify the state of another MPI process by performing a send action. In order to implement this, the communication daemon does not send messages before the event logger has acknowledged the reception of the preceding reception events.

When a fault occurs, the dispatcher detects it and spawns new processes to finish the computation. Re-executing nodes connect to their respective event logger and get the dependency information of all their receptions. Then, they execute the MPI program and ask for old messages logged on their respective senders. Messages to be replayed are identified by the couple (sender's identity; sender's logical clock) of the dependency information that is part of the re-emitted message. If some of these senders have crashed too, the missing messages are eventually sent during senders's own re-execution.

For scalability reasons, several event loggers may be used in a system, but every communication daemon must be connected to exactly one event logger. Since there is no dependency information to be managed between replaying nodes, event loggers do not have to communicate with each other.

4.6 The Checkpoint System

The checkpoint system is split in two parts: a server which stores the checkpoint images and a scheduler coordinating the checkpoints across all the system.

4.6.1 Checkpoint Management and Storage

The checkpoint server is a reliable repository storing the checkpoint images of the MPI processes and of the communication daemons. The checkpoint of the MPI process uses the Checkpoint Condor standalone library [17]. When it receives an order of checkpoint from the communication daemon, the process forks in two parts. The first part sends its process image to the communication daemon. Simultaneously, the second forked part continues the MPI application execution such that the checkpoint transfer is completely overlapped by the computation time. As the checkpoint is triggered by the communication daemon, this insures that there are no active communication at fork time.

On the MPI process point of view, there is no difference between execution and re-execution. The program is called by the communication daemon with a special argument handled by the Condor library. The process image is sent by the daemon in a special pipe and the process jumps to the last checkpoint and continues the execution. All the complexity of the communication replay is handled by the daemon.

The checkpoint of the communication daemon is not handled by the Condor library but by a user level method, serializing all the message information. When a checkpoint is triggered, the daemon sends the checkpoint order to the MPI process and gets back the process image. Then it serializes all its message data and sends them to the checkpoint server. During whole this phase, the communications with the other computing nodes are not suspended insuring that the checkpoint transfer can be overlapped by the MPI process computation.

When a computing node completes a checkpoint, it notifies all other communication daemons of the logical clock of this checkpoint. Once a checkpoint has been done at a particular logical clock, all the messages received before will never be requested again. Thus all these messages can be removed on their respective sender (garbage collector).

4.6.2 Checkpoint Scheduling

The checkpoint scheduling is motivated by two main reasons: 1) the sender-based pessimistic message logging protocol induces an extensive use of the memory in the communication daemons and in the checkpoint servers. The checkpointing is not only triggered by the objective of reducing the impact of faults on the execution time, but also for reducing the storage occupation by the logged messages. 2) Checkpointing the communication daemon induces a traffic

7

proportional to the size of the emitted messages. This traffic competes with the application communication traffic for the network bandwidth and thus should be reduced as much as possible.

The role of the checkpoint scheduler is to evaluate the cost and the benefit of a checkpoint, at any specific time, and to order the checkpoints accordingly. Periodically, it asks the communication daemons to send their status (in terms of the amount of logged messages), and evaluates the benefit of a checkpoint. Note that since the protocol does not need checkpoint coordination, the scheduling does not have to be fair. We have developed two checkpointing policies: a round robin, and an adaptive one.

The main advantage of the round-robin algorithm is its lack of communication between the scheduler and the nodes. Its main problem comes from the asymmetry of some communications schemes. If the MPI program communications are very symmetric (for example all-to-all scheme), the amount of data stored on all the nodes is similar and the round robin algorithm is fair. If the communication is asymmetric, some nodes have to be checkpointed more often than others.

For these communications schemes, we provide the adaptive algorithm, considering the ratio "amount of received messages" over "amount of sent messages" for each computing node. It computes a scheduling following a decreasing order of this ratio across the nodes. We have built a simulator and have compared the two policies with classical communication schemes (point to point, synchronous all to all, broadcasts and reduces). The comparison demonstrates that the adaptive algorithm never provides a worse scheduling (w.r.t. bandwidth utilization) and often provides better scheduling (up to n times better, n being the number of computing nodes for asynchronous broadcast).

4.7 MPIrun

Our goal while implementing the mpirun of MPICH-V2 is, like the rest of the project, to provide the users with a completely seamless integration into the standard MPICH architecture. Thus, despite the fact that MPICH-V2 might require more information about the configuration of the different machines, the user just runs a parallel program using the standard mpirun command. Advanced users might manually provide more customized information such as the communication ports between the different entities.

The program uses a modular architecture, mainly because the checkpoint/restart feature of MPICH-V2 requires a dispatcher for launching the different programs and also for monitoring the execution of these programs. The two main phases of a run are: the preparation of 'program files' describing the characteristics of the run and the execution of the MPI program.

The run preparation consists in a shell script (which may

inter-operate with a batch scheduler) creating a 'program file' from a list of available machines for a run and the MPICH-V2 specific commands (executable, number of processors to be used). The obtained program file is the equivalent of a 'P4PGFILE' for the original MPICH-P4. It describes the run, with for each machine 1) its role inside the system (Computing Node, Event Logger, Checkpoint Server, Checkpoint Scheduler) and 2) the list of options for that role. The user can specify these options for each machine through the use of an extended MPICH-like machines file, or with general defined attributes by using special option flags, or with default configurations.

The execution monitor first launches the execution (by rsh or ssh) of the different programs (CS, EL, SC, CN), and then monitors the execution potentially re-launching the crashed programs. To detect faulty nodes, we assume that the whole execution runs on a synchronous (e.g. a Cluster) or controlled area network (e.g. a Grid). In such networks, a socket disconnection is considered as a trusty fault detector. Basically, at the beginning of the execution, one socket is open for every computing node. The monitor pulls these sockets for detecting disconnections. The number of open sockets might be quite large. This has a negligible influence on the overall network usage, since only very few messages are sent during the whole execution. At the end of the program, the monitor receives a finalize message from every computing node. It cleans the execution pool by stopping the different auxiliary programs.

5 Performance Evaluation

The experimental tests presented in the following section have been run on a cluster of PCs under Linux 2.4.18. The cluster has been used in dedicated mode to ensure a fair comparison between the different implementations. The cluster consists in two major parts: 32 computing nodes, and 12 auxiliary machines connected to a single 48-port Ethernet 100 Mbit/s switch.

The computing nodes are equipped with Athlon XP 1800+, running at 1.5 GHz and 1GByte of main memory plus 1GByte of swap on IDE disk. The second part of the cluster, which is used to run the different auxiliary programs (Event Loggers, Checkpoint Servers, Checkpoint Schedulers), is composed of dual-Pentium III machines, with processors running at 500MHz with 512MB of main memory and 1GByte of swap on IDE disk.

All MPI implementations use the MPICH version 1.2.5. Test programs were compiled using the GNU GCC, version 2.96 and PGI PGF77 compilers. To compile the MPICH implementations (P4, MPICH-V, MPICH-V2), we use the default optimizations as recommended by the MPICH team. In order to use the Condor checkpoint library, we used the condor_compile wrapper, integrated in MPICH-V2 wrap-

8

pers for the link phase. For the C MPI microbenchmarks, we used the compilation flags `-O3`. For the Fortran MPI programs we used the compilers `-O3 -tp=athlonxp`.

The performance and fault tolerance comparison will consider MPICH-P4 and MPICH-V1. While some of the others fault tolerance MPI implementations are available for download, they either did not match our experimental conditions (Egida runs on Sun Solaris 2.8) or implement other fault tolerance techniques: Starfish uses a coordinated checkpoint protocol and is implemented in OCaml, CoCheck is also based on coordinated checkpoint and requires the tuMPI implementation, FT-MPI and LA-MPI target other fault tolerance approaches (user or network level).

5.1 Raw Communication Performance

Before presenting the performance of application benchmarks, we present the comparison between MPICH-P4, MPICH-V1 and MPICH-V2 on bandwidth and latency, for a synchronous ping-pong test. As usual, the performance evaluation considers a mean over a large number of measurements. For MPICH-V1, we use 2 Memory Channels in addition to the computing nodes.

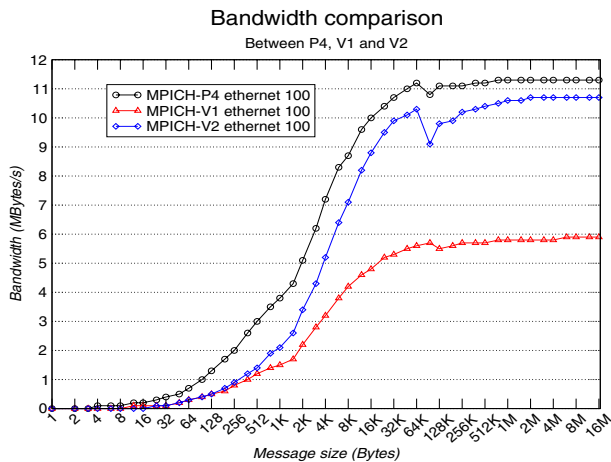Figures 5 and 6 present the comparison of the bandwidth and latency obtained with this ping pong test.

Figure 5. Bandwidth comparison for a ping-pong test for the 3 different MPI libraries

The maximum bandwidth obtained with MPICH-V2 for large message sizes is 10.7 MByte/s, while for the same messages, MPICH-P4 reaches 11.3 MByte/s. As expected, MPICH-V2 is slightly slower than MPICH-P4, but remains always close to MPICH-P4. The difference is explained by the acknowledge of message logging with the event loggers. MPICH-V1, tested with one memory channel per computing node, is down to two times slower than MPICH-P4.

However, as presented in [5], the slowdown of MPICH-V1 is reduced by two when using asynchronous communication routines. This slowdown is explained by the communication protocol which involves crossing a Channel Memory for each communication. MPICH-V2 has been designed to overcome this penalty by removing the need of Channel Memories.
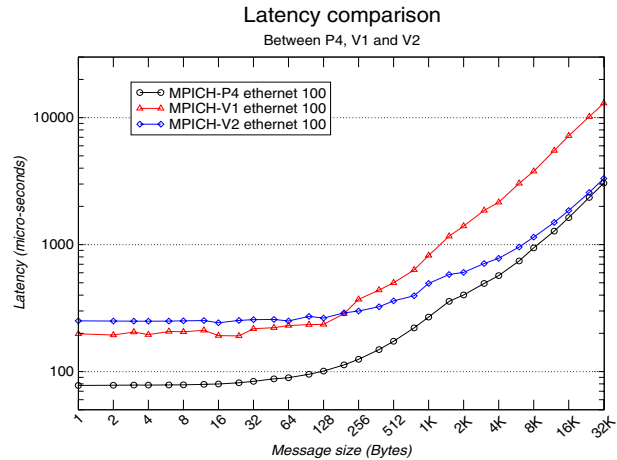

Figure 6. Latency comparison for a ping-pong test for the 3 different MPI libraries

The minimum latency obtained with MPICH-V2 for short messages (0-byte long) is 237 microseconds, while with P4, it is 77 microseconds. For short messages, the overhead of MPICH-V2 is explained by the cost of the synchronization with the event logger: the next message of a sequence can not be sent until the logging of the previous one is acknowledged. Between two sends, six TCP messages are sent with MPICH-V2 (P4 only sends two). For large messages, the overhead of event logging becomes negligible because most of the communication time is spent on the transfer of the message payload.

5.2 The NAS Benchmarks

In order to test the performance of MPICH-V2 on a wide set of well established and optimized MPI programs, we test the performance of the NAS Parallel Benchmark ([3]) NPB 2.3 with MPICH-P4 and MPICH-V2. For all tests with MPICH-V2 we execute the Event Logger, the Checkpoint Server and the Checkpoint Scheduler on a single reliable node. However no checkpoint is executed during the runs.

We use the following kernels: CG, MG, FT, and mini-applications: LU, BT, SP with dataset size A and B. We present the performances up to 32 processors, except for BT and SP which require a square number of processors (maximum: 25 in these cases).
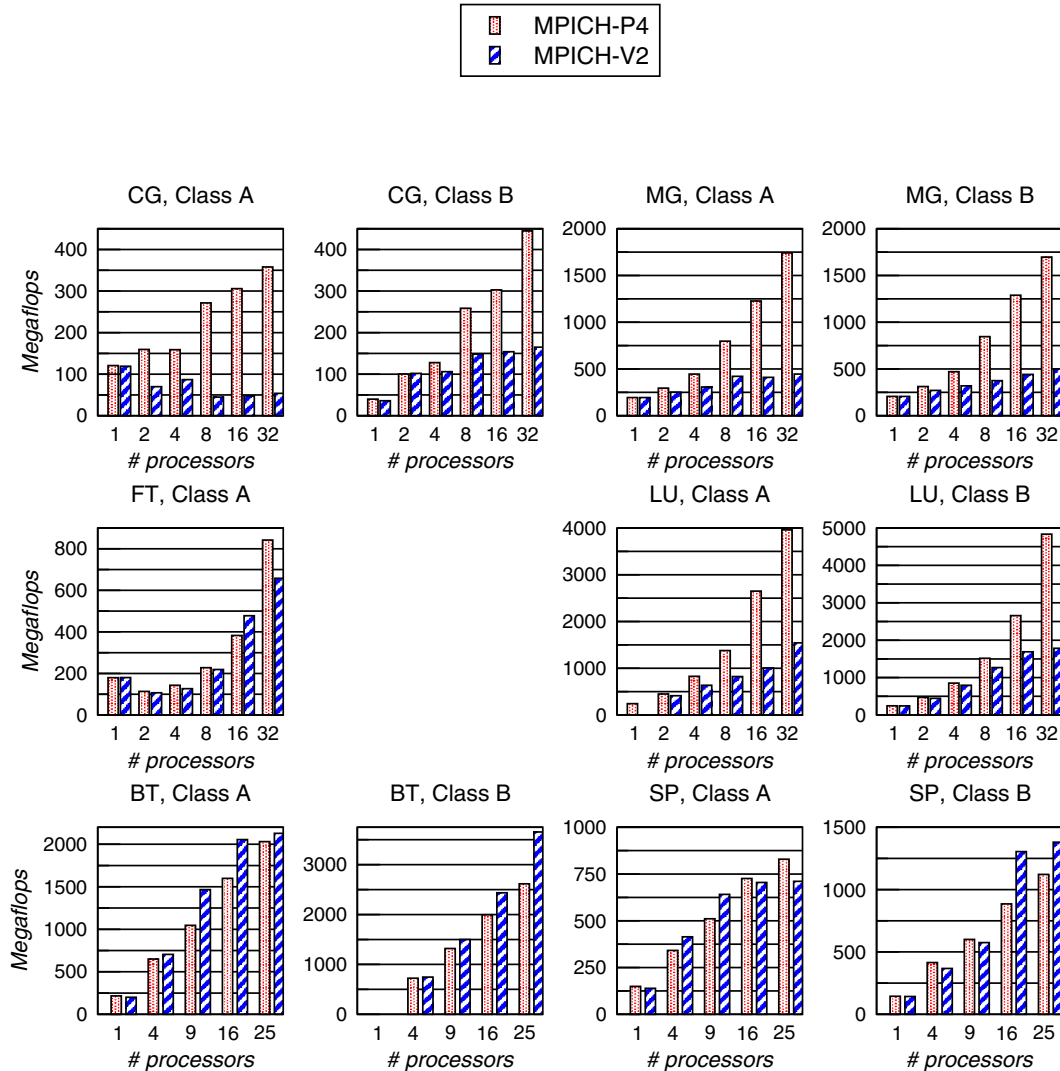
9

Figure 7. Performance comparison of the P4 and V2 MPI implementations for the NPB 2.3 Class A and B

Figure 7 presents the performance comparison of the 3 tested MPI implementations for the NPB 2.3.

Figure 7, clearly demonstrates that the higher latency of MPICH-V2 leads to a high performance penalty compared to MPICH-P4 for benchmarks using a lot of short messages, like CG and MG. FT uses an All-to-All communication pattern involving many large messages. The bandwidth of MPICH-V2, which is close to the one of MPICH-P4 for large messages, allows MPICH-V2 to reach the performance of MPICH-P4 on FT. We do not present the performance of FT Class B due to memory size limitations. We use a maximum storage size of 2 GB (1 GB on memory + 1 GB on disk) per node for message logging. This value is exceeded when executing FT Class B. Thus, checkpoint-

ing is recommended in such a case not only for fault tolerance but also for removing logged messages on the computing nodes. The poor performance of LU is explained by the use of the disk storage which reduces dramatically the performance of the message logging system. Note that for LU, a decomposition of the execution time (computation time + communication time) shows that the message logging daemon becomes a competitor of the MPI process for CPU resources, increasing the computation time compared to MPICH-P4. The last two benchmarks, SP and BT, demonstrate that MPICH-V2 can provide the same performance as MPICH-P4 or even better ones. It is obvious that MPICH-V2 will not be used to run very short duration kernels like the NAS CG and MG very sensitive to communica-

10

tion latency. MPICH-V2 targets long duration applications involving large messages. The performance on SP and BT clearly shows that MPICH-V2 is well suited to this kind of applications.

Figure 8 presents execution time breakdown of CG-A and BT-B (two performance extremes for MPICH-V2). We also recall the performance of MPICH-V1 for the same programs. The system setup for MPICH-V1 uses N/4 Memory Channels, N being the number of computing nodes.
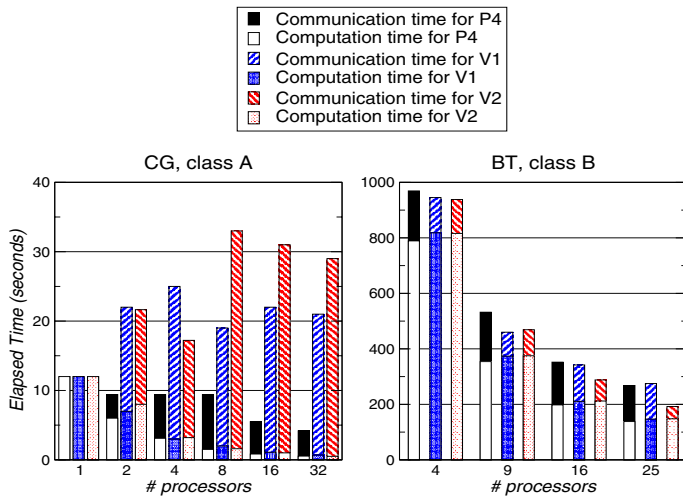
Function	BT A 9		CG A 8	
	P4	V2	P4	V2
MPI_(I)send	44.9s	3.4s	3.5s	0.6s
MPI_Irecv	0.32s	0.32s	0.0038s	0.0130s
MPI_Wait	4s	17.5s	1.6s	13.8s
Total time	49.22s	21.22s	5.1s	14.413s

Table 1. Time decomposition of MPI communication functions for BT and CG benchmarks for MPICH-V2 and MPICH-P4

Figure 8. Execution time breakdown of the 3 MPI implementations for CG-A and BT-B

Figure 8 shows that the computation times are the same for all the implementations for the two benchmarks. The poor performance of MPICH-V1 and MPICH-V2 for CG-A is explained by the communication time, which increases dramatically, due to the overhead of the message logging protocols. MPICH-V1 performs better than MPICH-V2 for this small communications due to its lower latency for small messages. For BT-B, the communication performance of MPICH-V2 is better than both MPICH-P4 and MPICH-V1. This is due to lower bandwidth of MPICH-V1 for large messages and the different ways how asynchronous communications are handled in MPICH-P4 and MPICH-V2. MPICH-V2 requires much less reliable nodes than MPICH-V1 (1 versus 9 for 32 computing nodes).

Table 1 presents a decomposition of communication time for BT and CG benchmarks (Class A).

This communication time breakdown clearly highlights the architecture difference between MPICH-V2 and MPICH-P4. In MPICH-V2, all communications are executed by the daemon processes associated with the MPI processes. If the communication is an asynchronous send, MPICH-P4 sends the message payload during the execution of the ISend function, while MPICH-V2 only posts a mes-

sage notification. In MPICH-V2, the actual message transmission is done during the execution of the Wait function. The table demonstrates that MICH-V2 increases the communication time for CG-A-8 by a factor of 3. The reason behind this poor performance is the fault tolerant protocol which imposes that any reception event must be logged on the Event Logger before any subsequent emission. The table also confirms the better performance of MPICH-V2 for the BT benchmark.

To explain the difference, we present a synthetic benchmark (Figure 9) measuring the bandwidth of a communication pattern identical to the one of BT and SP benchmarks. The test performs a ping-pong of 10 non-blocking sends (MPI_ISend), 10 non blocking receives (MPI_IRecv) and then waits for all these communications to finish (MPI_Waitall).
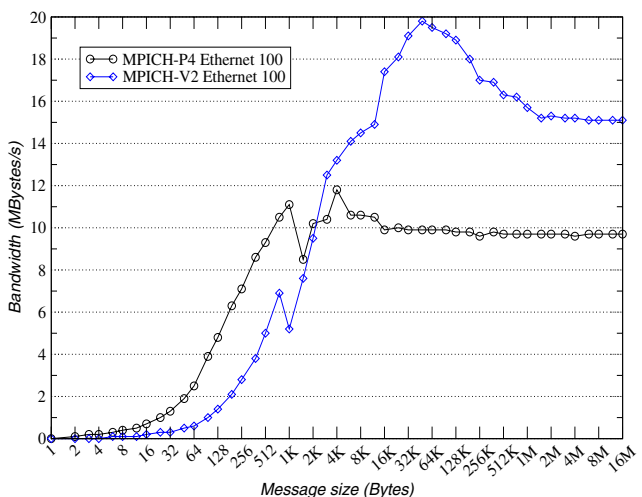

Figure 9. Bandwith comparison between MPICH-P4 and MPICH-V2 for a synthetic benchmark executing a communication pattern identical to the one of BT/SP benchmarks

11

Excepted for small messages where the higher latency of MPICH-V2 is predominant, MPICH-V2 performs better for non-blocking communications than MPICH-P4, reaching twice the P4 bandwidth for 64Kbytes messages. This higher communication performance of MPICH-V2 is due to its capability to handle full duplex communications. In contrary to the P4 driver, when sending a message, the V2 driver pools for incoming receptions after each transmission of a message chunk. This driver structure gives an advantage to MPICH-V2 for long messages.

5.3 Re-execution Performance

Figure 10 shows the re-execution performance. The benchmark consists of an asynchronous MPI token ring ran by 8 computing nodes and a server running the event logger. For measuring the re-execution time we stop the benchmark execution just before the MPI finalize call. Then we stop and restart some computing nodes from the beginning and measure their completion time. For this microbenchmark, the checkpoint features of MPICH-V2 are deactivated (checkpoint scheduler and server).
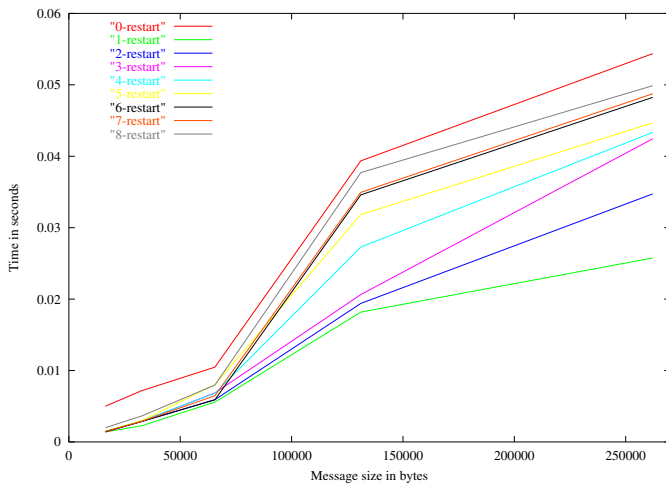

Figure 10. Performance of Re-execution

The reference curve (0-restart) shows the performance of the first complete run of the benchmark without restart. The "x-restart" curves show the time for re-executing this application on "x" nodes.

The non-linearity of the curves between 64kB and 128kB is due to the protocol change from eager to rendez-vous. The figure shows that re-execution time for one single restart is about half of the reference one because: 1) for this benchmark each node is executing the same amount of receptions and emissions and 2) during the re-execution, only the receptions are replayed.

When a large amount of nodes are re-executing, the re-

execution time becomes close to the reference one but stays lower because the communications to the event logger are not replayed.

5.4 Faulty Execution

The next evaluation consists in measuring the performance degradation of the BT benchmark when faults occur during the execution. Figure 11 presents the execution time of BT for the class A dataset size using 4 computing nodes and a single reliable node for executing the Checkpoint Server, the Checkpoint Scheduler and the Event Logger.
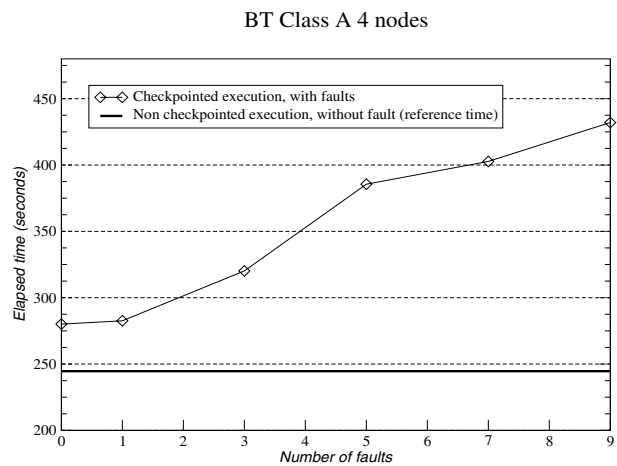
BT Class A 4 nodes

Figure 11. Performance of BT-A on 4 nodes when the number of faults increases during the execution

The checkpoint of a node immediately follows the one of another node. Thus, the system is always checkpointing a node. We use a scheduling policy randomly selecting the node to checkpoint. We simulate faults by sending a termination signal to a randomly selected MPI process. Faults may occur at any time during the execution, including during the checkpoint or during the re-execution. They are triggered randomly. The execution is restarted immediately from the checkpoint image provided by the checkpoint server. If no checkpoint image is available, the MPI process restarts the execution from the beginning.

Figure 11 demonstrates 1) the low overhead of the checkpoint system when no fault occurs 2) the smooth degradation of the execution time according to the number of consecutive faults and 3) an execution time lower than twice the reference execution time (without fault) when 9 faults occur during the execution. This last test clearly highlights the fault tolerance properties of MPICH-V2.

12

6 Conclusion and Future Works

We have presented the second protocol for MPICH-V using an original sender based pessimistic message logging protocol. This MPI implementation, based on MPICH 1.2.5, provides transparent/automatic fault tolerance. MPICH-V2 architecture relies on five components: a set of Computing Nodes, a Dispatcher, a set of Checkpoint Servers, a Checkpoint Scheduler and a set of Event Loggers. The main motivation for designing and implementing MPICH-V2 was to remove the cost and performance penalties involved by the use of Channel Memories in the first version of MPICH-V. A typical deployment of MPICH-V2 involves a set of volatile computing nodes, a reliable node for executing the Checkpoint Server, and a second reliable node for running the Dispatcher, the Checkpoint Scheduler and the Event Logger. Thus, compared to MPICH-V1, MPICH-V2 requires much less reliable nodes.

Using the Ping-Pong test, we have demonstrated that the bandwidth of the new protocol is close to the one of MPICH-P4 and outperforms the one of MPICH-V1. The performance comparison using the NAS Benchmarks, shows that MPICH-V2 can not compete in performance with MPICH-P4 on CG and MG kernels due to its high latency on small messages. However, MPICH-V2 targets long duration applications using large messages. The performance for the SP and BT benchmarks demonstrates that MPICH-V2 reaches the performance of MPICH-P4 for such applications. The execution of BT class A on 4 nodes, with up to 9 consecutive faults (1 fault every 45 seconds) highlights the fault tolerance properties of MPICH-V2, leading to a performance degradation compared to MPICH-P4 lower than a factor of 2.

Future works will consider improving the performance for small messages and test MPICH-V2 on large clusters and Grid deployments.

7 Acknowledgments

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *In 8th International Symposium on High Performance Distributed Computing (HPDC-8 '99)*. IEEE CS Press, August 1999.

[2] 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.

[3] 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.

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

[5] 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 *SC2002: High Performance Networking and Computing (SC2002)*, Baltimore USA, Novembre 2002. IEEE/ACM.

[6] 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.

[7] Yuqun Chen, Kai Li, and James S. Planck. Clip: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing (SC97)*. IEEE/ACM, November 1997.

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

[9] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in manetho. In *22nd International Symposium on Fault Tolerant Computing (FTCS-22)*,

pages 18–27, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[10] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[11] Robert E.Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *18th Annual International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 44–49. IEEE CS Press, June 1988.

[12] 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.

[13] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. Harness and fault tolerant mpi. *Parallel Computing*, 27(11):1479–1495, October 2001.

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

[15] 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.

[16] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[17] 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.

[18] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. In *Parallel Processing Letters(PPL)*, volume 10(4). World Scientific Publishing Company, 2000.

[19] M. Paterson M. Fischer, N. Lynch. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[20] 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.

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996. http://www.netlib.org/utk/papers/mpi-book/mpi-book.html.

[22] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, April 1996. IEEE CS Press.

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

[24] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

14

Appendix

A MPICH-V2 protocol

We present here the MPICH-V2 protocol, simplified for readability reasons. This version of the protocol is used in the appendix B for the proofs.

Variables for process p

EL_p list of events to replay (init : empty)

H_p logical clock (automatically increased at every computation step) (init : 0)

$HR_p[q]$ date of last received event from process q (in q's clock) (init : 0)

$HS_p[q]$ date of last sent event to process q (in p's clock) (init : 0)

$SAVED_p$ set of messages backup (init : empty)

Routines

$LOG(data, d)$ schedules the saving of information $data$ at date d in a reliable media

$WAITLOGGED()$ blocks until every previous $LOG()$ call have completed

$SEND(x, d)$ sends the data x to the processor d

$UNDETACTION(d)$ executes the undeterministic action with the corresponding data d

$POP(list)$ pops the first event of the list $list$

$DELIVER(m, p)$ delivers the message m from process p to the MPI process

$RECV(m, p)$ receives the message m from process p

$ROLLBACK()$ loads the last previous successful checkpoint, then finishes the actions and continues the execution where the checkpoint was taken

$DownloadEL(H_p)$ gives all data stored in the reliable media for all clocks $> H_p$.

Actions for process p
These actions implement the corresponding routines for the application.

send(m, q**)**
    if $H_p \geq HS_p[q]$
        $WAITLOGGED()$
        $SAVED_p = SAVED_p \bigcup \{(m, H_p, q)\}$
        $SEND((H_p, p, m), q)$
        $HS_p[q] = H_p$

UnDetAction($data$**)**
    if EL_p is empty
        $LOG(data, H_p)$
        $UNDETACTION(data)$
    else
        $logdata = POP(EL_p)$
        $UNDETACTION(logdata)$

recv()
    if EL_p is empty
        $RECV((H_q, q, m), q)$
        $HR_p[q] = H_q$
        $LOG((H_q, q), H_p)$
        $DELIVER(m, q)$
    else
        $(logH_q, logq) = POP(EL_p)$
        $RECV((logH_q, logq, m), logq)$
        $DELIVER(m, q)$

Rules for process p
These rules are called when the corresponding event occurs at process p

on Restart()
    $ROLLBACK()$
    $EL_p = DownloadEL(H_p)$
    $\forall q, SEND(RESTART1(HR_p[q]), q)$

on RECV(RESTART1(HP), q)
    $HS_p[q] = HP$
    $SEND(RESTART2(HR_p[q]), q)$
    $\forall (m, h, q) \in SAVED_p,$
        if$(h > HS_p[q]) SEND((h, p, m), q)$

on RECV(RESTART2(HP), q)
    $HS_p[q] = HP$
    $\forall (m, h, q) \in SAVED_p,$
        if$(h > HS_p[q]) SEND((h, p, m), q)$

B Theoretical Foundations

In order to prove the theorem 2, we consider the following model, derived from [24]: a system consists of a set of processes linked together by communication channels. A process is a state machine $(\Sigma, I \in \Sigma, F \subseteq \Sigma, \omega, \delta \subseteq \Sigma \times \omega \times \Sigma)$, where Σ is the set of states of the process, I the initial state, F a set of final states, ω the set of actions of the process, and δ the relationship of atomic steps of the process.

Basically, ω holds three kinds of actions : internal step (the jth internal step of process p is denoted i_p^j), receptions (a reception at the process p from the process q of the message m is denoted $r_p^q(m)$) and emissions (an emission at the process p to the process q of the message m is denoted $e_p^q(m)$). We add another special atomic step, CAR that we describe later, used to model faults.

15

A communication channel is a FIFO queue that links together two processes. When a process p does an emission to q of m, m is appended to the communication channel (p, q) ; when q does a reception from p of m, m is the first element of (p, q) and is removed from (p, q).

A configuration of the system is the couple (P, L), where P is the vector of the states of every process in the system, and $L = < (p, q) \backslash p, q$ processes $>$ the vector of the queues of the communication channels.

The configuration where every communication channel is empty and every process is in its initial state is called the initial configuration. Every configuration where every process is in one of its terminal state is called a terminal configuration.

Let C be a configuration of the system S, the application of at most one atomic step per process to C is called a transition.

An execution of the system S is an alternated sequence of configurations and transitions $C_0, T_0, C_1, \ldots, C_n$, where C_0 is the initial configuration of S, T_i is a transition where every atomic step is applicable to C_i and leads to C_{i+1}. C_n is a terminal configuration.

Crash with recovery model

The model of fault we consider includes permanent failures of processes. Within asynchronous systems, it is well known that the detection of such faults is impossible ([19]). We assume here the existence of failure detectors, which detect the crashes, and a monitor which restarts eventually every crashed process in the state where the only possible atomic step consists in execute the **on Restart()** rule.

Thus, we model a fault by a specific atomic step *CAR* (Crash And Recover). When doing a CAR step, a process resets its state to the initial state and call the **on Restart** rule, and every communication channel connected to it is emptied.

Let T be a transition from C to C'. Let a be an atomic step of process p, element of the transition T. We note $T|_p = a$.

Definition 4 (Causal dependency) *Let S be a system, P the set of processes of S and C_0 the initial configuration of S. The causal order of an execution $E = C_0, T_0, C_1, T_1, \ldots, C_n$ is a partial order $\lhd$ on the atomic steps of the processes such that:*

1. $i < j \Rightarrow T_i|_p \lhd T_j|_p$

2. $T_i|_p = e_p^{p'}(m) \wedge T_j|_{p'} = r_{p'}^p(m) \Rightarrow (i < j) \wedge (T_i|_p \lhd T_j|_{p'})$

3. $\exists g : e \lhd g \wedge g \lhd f \Rightarrow e \lhd f$

Definition 5 (deterministic event) *The application of an atomic step on a process p from a state s is called an event.*

An event on a process p is called a deterministic event if this atomic step is the only possible step to be executed from state s. If an event is not a deterministic event, it is called an undeterministic event.

Definition 6 (Piecewise determinism (PWD)) *Let $(e_0, e_1, \ldots, e_i)$ be the sequence of events on a process p during an execution E. The piecewise determinism assumption assume that this sequence can be split in intervals such that all intervals except the first one begin with one undeterministic event followed by a finite number of deterministic events. The first interval is composed of a finite number of deterministic events.*

The first theorem ensures that under the Piecewise determinism assumption, a process that reexecutes the same sequence of undeterministic events as in initial execution reexecutes the same total sequence of events computed in initial execution.

Theorem 2 *The protocol of MPICH-V2 is a pessimistic message logging protocol.*

Proof: Since the MPICH-V2 protocol splits the messages in two parts : the logical clock of the reception in one hand, and the message payload in the other hand, the proof is done in two parts : first, we prove that every event which must be re-executable, in order to be a pessimistic message logging protocol, has its logical date logged on a reliable media, then we prove that any undeterministic event whose logical date is logged on a reliable media is re-executable.

a) By definition of the pessimistic message logging protocols, in every configuration C of an execution E, for every message m transmitted between the initial configuration I of E and C, if $|Depend_C(m)| > 1$, the message m must be re-executable.

Let $r_p^{p'}(m)$ denotes the reception of m in transition T_i. By definition of $Depend$, in any configuration C_j, $Depend_{C_j}(m) = \{P$ s.t. $\exists k \in [i, j[, r_p^{p'}(m) \lhd T_k|_P\}$

We prove by contradiction that until m is logged, for any j, $|Depend_{C_j}(m)| \leq 1$. Assume, for the sake of contradiction, that at some configuration C_j such that the logical date of message m is still not logged, $|Depend_{C_j}(m)| > 1$: by definition of causal dependency, there exist a process $p1 \neq p$, $u \in [i, j - 1[$, $v \in]u, j[$ and m' a message such that $T_u|_p = e_p^{p1}(m')$, $T_v|_{p1} = r_{p1}^p(m')$ and $T_u|_p \lhd T_v|_{p1}$. Thus p is a member of $Depend_{C_j}(m)$ and $T_i|_p = r_p^{p'}(m) \lhd T_u|_p = e_p^{p1}(m')$

According to the action **send()** of the protocol, it thus means that $WAITLOGGED()$ has returned (since no message is actually sent before $WAITLOGGED()$ returned). But, according to the action **recv()** of the protocol, the logical date of m has been scheduled to be logged

before the emission of m'. So, $WAITLOGGED()$ has returned, ensuring m is logged in C_j and, by hypothesis, m is not logged in C_j. We reach a contradiction. Thus, until the logical clock of m is safely logged, $|Depend_C(m)| \leq 1$.

b) According to the **UnDetAction()** and **recv()** actions of the protocol, every undeterministic action is logged with every information necessary to re-execute them, except the message data. We prove now that whatever the faults are, the protocol ensures that every message which date has been logged can be re-executed, meaning that its data (which is not logged) can be retrieved or built.

Let E be an execution with a finite number of faults leading to a configuration C. Let L be the finite set of all logical dates needed to be replayed in order to be a pessimistic message logging protocol $|L| = l$ Let $[H_1, H_2, \ldots, H_n]_C$ be the vector of processes clock in the configuration C. Let $[]_{C|_p}$ be the clock of p in C.

According to the piecewise determinism propriety, all deterministic events from the configuration C will be replayed until the next undeterministic event of each process, leading to a configuration C'.

Now we have to prove that all events logged in L are re-executable.

We prove by contradiction that in every configuration reached from C' by replaying some logged events there exists at least one logged event that can be reexecuted.

In the configuration C' all processes are waiting for an undeterministic event. Assume there is no log re-executable in L. Thus for all logs $(h, p_i, []_{C'|_{p_j}})$ in L, $h > []_{C'|_{p_i}}$, or else according to the lemma 1 there exists an unique saved message (m, h, p_i) in $SAVED_{p_i}$. and m is re-executable.

Let $F = \{p_1, \ldots, p_m\}$ be the finite set of processes that have a log in L in configuration C', $|F| = m$

For all processes p in F, let $ri_p^t(m)$ be the event of the initial execution that have been logged such that it is the next event to be reexecute by p in C'. Let $ei_t^p(m)$ be the send event of the message received in $ri_p^t(m)$.

Let p_i be a process of F. There exist a process p_j and a message m_1 such that $ei_{p_j}^{p_i}(m_1) \lhd ri_{p_i}^{p_i}(m_1)$. p_j is a member of F. If p_j is not a member of F then it either have not crashed or, according to theorem 1, have achieved to reexecute its send event as in initial execution Thus, according to lemma 1, m_1 is in $SAVED_{p_j}$ and $ri_{p_i}^{p_j}(m_1)$ can be reexecuted. $p_j \neq p_i$ (assertion 1) or else m_1 is in $SAVED_{p_i}$, according to lemma 1, and thus $ri_{p_i}^{p_j}(m_1)$ can be reexecuted. Thus $p_j \in F \backslash \{p_i\}$

There exists a process p_k and a message m_2 such that $ri_{p_j}^{p_k}(m_2) \lhd ei_{p_j}^{p_i}(m_1)$. By definition of causal dependency, every couple of events $\{e_1, e_2\}$ happening on the same process are totally ordered: $e_1 \lhd e_2$ or $e_2 \lhd e_1$. If $ei_{p_j}^{p_i}(m_1) \lhd ri_{p_j}^{p_k}(m_2)$ then according to theorem 1, $ei_{p_j}^{p_i}(m_1)$ has been reexecuted and according to lemma 1, m_1 is in $SAVED_t$ and $ri_{p_i}^{p_j}(m_1)$ can be reexecuted in C', thus

$$ei_{p_k}^{p_j}(m_2) \lhd ri_{p_j}^{p_k}(m_2) \lhd ei_{p_j}^{p_i}(m_1) \lhd ri_{p_i}^{p_j}(m_1)$$

Moreover $p_k \neq p_j$ (same proof as assertion 1) and $p_k \neq p_i$. If $p_k = p_i$ then $ei_{p_i}^{p_j}(m_2) \lhd ri_{p_i}^{p_j}(m_1)$ and m_2 is in $SAVED_{p_i}$ according to lemma 1. So $ri_{p_j}^{p_k}(m_2)$ can be reexecuted, Thus $p_k \in F \backslash \{p_i, p_j\}$.

Recursively m times,

$$\underbrace{ei_{p_u}^{p_v}(m_3) \lhd ri_{p_v}^{p_u}(m_3)}_{1} \lhd \cdots \lhd \underbrace{ei_{p_j}^{p_i}(m_1) \lhd ri_{p_i}^{p_j}(m_1)}_{m}$$

and we reach $p_u \in F \backslash F = \emptyset$ in order to can not reexecute any logged event of L. Thus p_u is either not a crashed process and then, according to lemma 1, have a copy of the message m_3 in $SAVED_{p_u}$, or there exists a process $p_w \in F$ such that $ei_{p_u}^{p_v}(m_3) \lhd ri_{p_u}^{p_w}(m_4)$ and m_3 is in $SAVED_{p_u}$.

In either case $ri_{p_v}^{p_u}(m_3)$ can be reexecuted, in contradiction to hypothesis that no logged event in L can be reexecuted. Thus in configuration C' there exists a logged event e in L that can be reexecute similarly as in the initial execution, leading to a configuration C'' with $L \backslash e$ undeterministic events to be reexecuted, $|L| = l - 1$.

We prove similarly that in every configuration C'' reached from C by replaying some logged events $(e1, \ldots, e2)$, there is an event e of $L|(e1, \ldots, e2)$ that can be reexecuted, leading to configuration C''' with $L|(e1, \ldots, e2, e)$ remaining logged events to reexecute, and so on l times until $L = \emptyset$. Thus, for every logged event e in L, e can be reexecuted.

∎

Lemma 1 *Let C' be the configuration reaches after a finite number of faults such that all crash processes have replayed their execution until their first undeterministic event to be reexecuted, $\forall (h_p, p, h_q) \in$ reliable media s.t $h_p < []_{C'|_p}, \exists (m, h_p, q) \in SAVED_p$*

Proof: According to the **on Restart()** action, when restarted, a process p loads its checkpoint image including the clock h_p of this process state and the set $SAVED_p$ of all messages sent with a clock lower than h_p. Moreover according to the protocol, all **send()** events which are deterministic are replayed at the same clock with the same data and thus according to the **sent()** action are appended to respective $SAVED$ set. Thus for all processes p, for all logical dates $[]_{C'|_p}$, for all logs in reliable media (h_p, p, h_q) with $h_p < []_{C'|_p}$ then there ex¡ists an unique m such that $(m, h_p, q) \in SAVED_p$. ∎