Abstract—Nowadays, clusters and grids are made of more and more computing nodes. The programming of multi-processes applications is the most often achieved through message passing. The increase of the number of processes implies that these applications need to use a fault tolerant message passing library. In this paper, we present two implementations of fault tolerant protocols based on MPICH, a blocking one and a non blocking one. We then compare their efficiency and the overhead induced during a failure-free execution.

I. INTRODUCTION

A long trend in the high performance computing systems is the increase of the number of nodes. This is illustrated by the composition of the Top500 supercomputer list. The average number of processors in the top 500 supercomputers is greater than 1000. Moreover, more than three quarter of these supercomputers have between 257 and 1024 processors, and the three most powerful systems have more than 10,000 processors. As the number of processes increases, the probability of failure of a single component also increases. So fault-tolerance is desirable a property for distributed applications running on these systems. Moreover, in order to reduce system costs, the concept of grids has emerged last years. It consists in gathering resources of different clusters for increasing the computation capacity. In such a system, clusters are administrates in different ways. Thus it is necessary to provide fully automatic and transparent fault tolerant applications.

Message Passing Interface is currently the programming paradigm and communication library most used to program the supercomputers. Thanks to 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 life time 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 a first choice programming environment for high performance applications. MPI in its specification [1] and most deployed implementations (MPICH [2] and LAM/MMPI [3]) follows the fail stop semantic (specification and implementations do not provide mechanisms for fault detection and recovery). Thus, MPI applications may be stopped at any time during their execution due to an unpredictable failure.

In order to avoid the restarting from the beginning of an MPI application because of only one failure, fault tolerance MPI implementation are very desirable. The typical fault tolerant technique implemented in MPI library is the coordinated checkpointing. This technique consists in regularly taking a global state of the system and, if a failure occurs, restart this application from this global state. There is two main ways to implement this technique. The first one, called the preemptive coordinated checkpointing, consists in stopping the MPI computation to takes the global state. This permit having a better control on the state of the different processes and their communication channels. The second one, called non-blocking coordinated checkpointing, does not provide this kind of control, but does not require to interrupt the MPI computation.

The blocking solution is simple to implement in high performance driver because it requires few modifications in the low communication layer. The non blocking solution even if does not stop the computation, can require modifications introducing overhead in the driver. As the number of processes regularly increases, it is
important to evaluate the impact of this kind of fault tolerant protocols on large scale MPI computation. In this paper, we compare these two protocols, the blocking and non-blocking ones, and evaluate their impact on large scale systems. We detail the implementation of the blocking protocol inside MPICH2, compare it with the our precedent non-blocking implementation MPICH-vcl [4] and evaluate its impact on overall performance.

The paper is organized as follows. The second part of the paper presents the related works highlighting the originality of this work. Section III presents the common principle of the global checkpointing protocols, then the blocking and non-blocking solutions. Section IV presents the two implementations used to compare these two fault tolerant MPI protocols in a fair way. Section V presents the experiment results in terms of application performance and fault tolerance using micro benchmarks and the NAS benchmarks. Section VI sums up what we learned from these experiments.

II. RELATED WORKS

MPI is a standard for message-passing systems widely used for parallel applications. Several implementations of this standard are available, among them are two main open sources: MPICH and OpenMPI.

Fault tolerance in MPI applications can be implemented following three strategies: explicit (managed by the programmer), semi-automatic (guided by the programmer) and automatic (transparent for the programmer/user). In this paper we focus on the last strategy, that realizes fault-tolerance without any intervention from the programmer and allows an execution that does not report the failures to the user.

Several techniques are used to implement fault tolerance in message-passing systems. Simple replication is not relevant for such systems, since if the system is designed to tolerate \( n \) faults, every component must be replicated \( n \) times and the computation resources are thus divided by \( n \). The two main techniques used are message-logging and coordinated checkpoints. A review of the different techniques can be found in [5].

Message-logging consists in saving the messages sent between the computation nodes, and replay them if a failure occurs. It is based on the piecewise deterministic assumption: the execution of a process is a sequence of deterministic events separated by non deterministic ones [6]. With this assumption, replaying the same sequence of non-deterministic events at the same moment makes possible the recovering of the state preceding a failure. Thus these protocols consist for every process to save all its non-deterministic events in a reliable manner and to checkpoint regularly. When a failure occurs, only the crashed process is restarted from its last checkpoint, and it recovers its last state after having replayed all saved events. There is no need to coordinate the checkpoints of the different processes. No orphan processes (i.e. processes that are waiting for a message that will never come, since the expected sender is more advanced into its execution) are created. The recover mechanism is more complex than with coordinated checkpoints as a process shall obtain its past events and be able to replay them. Moreover the overhead induced during failure-free execution decreases the performances in not very faulty environments, such as clusters [4]. Furthermore, it can lead to the domino effect [7]: a process that rollbacks and that need a message to be replayed, asks another process to rollback. This process does, and asks another one to do so, etc. The execution can be restarted from the beginning because of cascading rollbacks and so the benefits of fault tolerance are lost.

Message-logging protocols are classified into three categories: optimistic, pessimistic and causal protocols. Optimistic protocols assume that no failure will occur between the moment a process executes a non-deterministic event and the moment this event is saved on a reliable storage support. So when a process executes a non-deterministic event, it sends it to the reliable storage support then continues its computation without waiting any acknowledgment [6]. The induced overhead during failure-free execution is then quite small, but the optimistic hypothesis introduces the risk to get an incoherent state if it is not realized. Pessimistic protocols do not make this hypothesis, and the processes wait for an acknowledgment from the reliable storage support to continue their execution [8]. The induced overhead during failure-free execution is then quite small, but the optimistic hypothesis introduces the risk to get an incoherent state if it is not realized. Pessimistic protocols do not make this hypothesis, and the processes wait for an acknowledgment from the reliable storage support to continue their execution [8]. The induced overhead during failure-free execution is then quite small, but the optimistic hypothesis introduces the risk to get an incoherent state if it is not realized. Pessimistic protocols do not make this hypothesis, and the processes wait for an acknowledgment from the reliable storage support to continue their execution [8].

Coordinated checkpointing has been introduced by Chandy and Lamport in [12]. This technique requires
that at least one process sends a marker to notify the other ones to take a snapshot of their local state and then form a global checkpoint. The global state obtained from a coordinated checkpoint is coherent, allowing the system to recover from the last full completed checkpoint wave. It does not generate any orphan processes nor domino effect, but all the computation nodes must rollback to a previous state. The recover process is simple, and a simple garbage collection reduces the size needed to store the checkpoints.

In blocking checkpointing protocols, the processes stop their execution to perform the checkpoint, save it on a reliable storage support (that can be distant), send an acknowledgment to the checkpoint initiator and wait for its commit. They continue the execution only when they have received this commit. The initiator sends the commit only when it has received all the acknowledgments from all the computing nodes to make sure that the global state that has been saved is fully completed. As claimed in [13], blocking checkpoints induce an important latency and non-blocking checkpoints are then more efficient.

Non-blocking coordinated checkpoints with distributed snapshots consists in taking checkpoints when a marker is received. This marker can be received from a centralized entity, that initiates the checkpoint wave, or from another computation node which has itself received the marker and transmits the checkpoint signal to the other nodes. This algorithm assumes that all the communication channels comply with the FIFO property. Therefore the computation processes do not have to wait for the other ones to finish their checkpoint, and then the delay induced by the checkpoint corresponds only to the local checkpointing.

Communication-induced checkpoint protocols (CIC) perform uncoordinated checkpoints but avoid the domino effect [14]. Unlike coordinated checkpoints, it does not require additional messages for a process to know when it has to perform a local checkpoint. The information about when a local checkpoint must be performed are piggybacked in the messages exchanged between the processes. Two kinds of checkpoints are defined: local and forced. Local checkpoints are decided by the local process, forced ones are decided by the process according to the information piggybacked in the messages. The forced ones avoid the domino effect and ensure then the progress of the recovery line, i.e. the set of checkpoints of all the processes describing a coherent global state. When a failures occurs, all the processes rollback to their last stored local checkpoint and then to the last recovery line. CIC is an interesting theoretical solution but it has been shown in [15], using NPB 2.3 benchmark suite [16], that it is not relevant for typical cluster applications.

Several MPI libraries are fault tolerant. A review can be found in [17]. Coordinated checkpointing has been implemented in several MPI implementations on different levels of the application.

LAM/MPI [3], [18] implements the Chandy-Lamport algorithm for a system-initiated global checkpointing. When a checkpoint must be performed, the mpirun process receives a checkpoint request from a user or from the batch scheduler. It propagates the checkpoint request to each MPI process to initiate a checkpoint wave. As in our blocking Chandy-Lamport implementation, each MPI process then coordinates itself with all the others, flushing every communication channel, in order to reach a consistent global state. If a failure occurs, mpirun restarts all the processes from their last stored state. Finally, processes rebuild their communication channels with the other ones and resume their execution.

III. PROTOCOLS

In this paper, we compare two global checkpointing protocols. These are rollback recovery protocols. To perform this rollback recovery, they regularly take a snapshot of the local state of every process (of the system), such that when a failure occurs, all processes are rolled back to their last stored state. In order to ensure the global checkpoint coherence resulting from the collection of the different local state, these two protocols rely on the Chandy and Lamport algorithm [12]. In this algorithm, one or more process can initiate a checkpoint wave. When a process starts a checkpoint, it records its local state and sends a marker to all its neighbors. When a process receives a marker, if it has not started its checkpoint wave yet, it starts it. Every message a process receives after having started its checkpoint wave and before having received the marker of the sender is recorded in the receiver image as the channel’s state.

The first protocol we consider in this paper, called Vcl, is a direct implementation of the Chandy and Lamport algorithm for MPI computations. A MPI process consists in two unix processes: a computation process (MPI) and a communication process (daemon). The communication process is used to store transit messages and to replay these messages when a restart is performed. Moreover, we added a process, the checkpoint scheduler, which is the only one that can initiate a checkpoint wave. Furthermore, specific processes, called checkpoint servers are used to store the local images of all processes.
Finally, we define a dispatcher for launching the different processes in the system, detecting failures and restarting the failed application.

Fig. 1. A vcl execution

The protocol works as shown in the figure 1. The MPI process 1 initially receives the marker from the checkpoint scheduler (1), stores its local state (2) and sends marker to every process (2). From this point, every message, like \( m \) in the figure, received after the local checkpoint and before having received the marker of the sender, is stored by the daemon process. When the MPI process 0 receives the marker, it starts its local checkpoint and sends a marker to every other process (3). The reception of this marker by 1 concludes the local checkpoint of 1. If a failure occurs, all processes restart from their last stored checkpoint (4) and the daemon process replays the deliver of the stored messages (5). Note that the message \( m' \) may be not sent again in the new execution.

The second protocol, that we call pcl, is used in other implementations [18]. This protocol consists in synchronizing the different processes for emptying the communication layer. Thus, during the checkpoint wave, no message is still being exchanged and there is no need to store the channel state in any way. When the system is restarted after a failure, every process reloads its last local image et re initializes the communication layer for establishing connection, then the computation can continue.

The synchronization is performed by markers exchanges to flush all channels. In pcl, global checkpointing is made by following this sequence of actions: The MPI process of rank 0 regularly starts a new checkpoint wave, it changes its state to checkpointing and sends markers to every other processes (1). When a process receives a first marker, it changes its state to checkpointing and sends markers to every other process (2). After having sent a marker, a process does not send any other message through the same channel until it takes its checkpoint (blue segment 3). Such messages, still in the process memory, are automatically stored in the checkpoint. Similarly, after having received a marker, a process does not receive any other message from the same channel, message receptions are delayed up until the end of the checkpoint of the process (green segment 4). When a process has received the marker of every other process, it checkpoints and sends the resulting image to the checkpoint server (5). After having taken its checkpoint, a process can send and receive any messages. When the images is completely stored, the process sends a message to rank 0 to warn it about the end of its checkpoint and continues its execution. Finally the rank 0 MPI process acknowledges the different checkpoint servers about the coherence of the wave after having received every confirmation of the end of checkpoint from every process. If a failure occurs, all processes restart from their last stored checkpoint (6) and every message delayed in emission will be sent again after the restart (7).

IV. IMPLEMENTATION DETAILS

MPICH is an prominent project developed at the Argonne National Laboratory. It aims at providing a high performance MPI library implementation. The first major revision, called MPICH, addresses the MPI-1 specification. The nowadays major revision, called MPICH2, extends the performance of the first one and addresses the MPI-2 new specifications. In this section we present the details of the integration of the global checkpointing mechanism inside this two major versions.

A. Non-preemptive checkpointing implementation inside MPICH

MPICH implements a full MPI library from a channel. Such channel implements the basic communication routines for a specific hardware or for a new communication
protocols. We developed a generic framework, called MPICH-V, to compare different fault tolerance protocols for MPI applications. This framework implements a channel for the MPICH 1.2.7 library, based on the chp4 default channel.

MPICH-V is composed of a set of runtime components and a channel called ch_v. This channel relies on a separation between the MPI application and the actual communication system. Communication daemons (Vdaemon) provide all communication routines between the different components involved in MPICH-V. The fault tolerance is performed by implementing hooks in relevant communication routines. This set of hooks is called a V-protocol. The two main V-protocol of interest for this paper are Vcl and Vdummy. Vdummy is a minimalist implementation of a non fault tolerant protocol using the MPICH-V architecture. Vdummy is used to measure the performance of the ch_v device and its communication daemon. Vcl implements the Chandy and Lamport algorithm (c.f. section III).

a) Daemon: A daemon manages communications between nodes, namely sending, receiving, reordering and establishing connections. It opens one TCP socket per MPI process and one per server type (the dispatcher and a checkpoint server for the Vcl implementation). It is implemented as a mono-thread process that multiplexes communications through select calls. Moreover, to limit the number of system calls, all communications are packed using iovec techniques. The communication with the local MPI process is done using blocking send and receive on a Unix socket.

b) Dispatcher: The dispatcher is responsible for starting the MPI application. It starts the different processes, servers first then MPI processes, using ssh to launch remote processes. The dispatcher is also responsible for detecting failures and restarting nodes. A failure is assumed after any unexpected socket closure.

Failure detection relies on the OS TCP keep-alive parameters. Typical Linux configurations define a failure detection as a miss of 9 consecutive losses of keep-alive probes, keep-alive probes being expected every 75 seconds. These parameters can be changed through the \#tcp_keepalive_probes# and \#tcp_keepalive_intvl# system parameters to provide more reactivity to hard system crashes. In this work, we emulated failures by killing the task, not the operating system, so failure detection was immediate, the TCP connection being cut as soon as the task is killed by the operating system.

c) Checkpoint server and checkpoint mechanism: The two implementations use the same abstract checkpointing mechanism. This mechanism provide an unified API to address three system-level task checkpointing libraries, namely Condor Standalone Checkpointing Library [19], libckpt [20] and the Berkeley Linux Checkpoint/Restart [21], [18]. All these libraries allow to take a unix process image in order to store it on a disk and to restart this process on the same architecture. By default, BLCR which is the most recent library is used.

The checkpoint servers are responsible for collecting local checkpoints of all MPI processes. When a MPI process starts a checkpoint, it duplicates its state by calling the fork system call. The forked process calls the checkpoint library to create the checkpoint file while the initial MPI process can continue the computation. The daemon associated with the MPI process connects to the checkpoint server, that first creates a new process and a checkpoint server only store one complete global checkpoint. Thus, it is never interrupted during a checkpoint phase.

When a global checkpoint is complete it is not necessary to still store the past global checkpoints. Thus, checkpoint servers only store one complete global checkpoint at a time using two files alternatively to store the current global checkpoint and the last complete one.

If a failure occurs, all MPI processes restart from the local checkpoint stored on the disk if it exists, otherwise they obtain it from the checkpoint server.

d) Checkpoint Scheduler: The checkpoint scheduler manages the different checkpoint waves. It regularly sends markers to every MPI process. The checkpoint frequency is a parameter defined by the user. It then waits for an acknowledgment of the end of the checkpoint from every MPI process before asserting the end of the global checkpoint to the checkpoint servers. The checkpoint scheduler starts a new checkpoint wave only after the
end of the previous one.

B. blocking checkpointing implementation inside MPICH2

MPICH2 is a new implementation of the MPI standard which extends results obtained in MPICH and addresses the issue of the MPI2 new functionalities. MPICH2 is structured in three layers: 1) the abstract device interface (ADI3) which links MPI standard to an extend set of high level communication routines, 2) chameleon 3 (CH3) which abstracts the ADI3 routines to an API composed of a few (ten to twenty) communication routines and 3) channels which implements this CH3 API depending on the specific network hardware or communication protocol.

We introduce in this paper a new implementation of a blocking checkpointing mechanism for fault tolerance inside MPICH2 called MPICH2-pcl. This implementation consists in a new channel, called ft-sock, based on the TCP sock channel, and two components: a checkpoint server and a specific dispatcher.

e) Ft-sock channel: The ft-sock channel is a derivation of the existing sock implementation. It consists in a basic set of communication routines using poll mechanism to multiplex I/O and iovec to reduce the number of system calls. The core of the communication system is based on sequences of request to send and request to receive for each MPI peer. Sending or receiving messages consists in posting such requests to the sock channel.

To implement the blocking checkpointing mechanism, the main modifications consists in adding a hook in the request posting function for verifying and delaying these posts if a checkpoint wave is currently active. The exchange of markers used in the protocol (c.f. III) is done by using the communication primitive defined in sock and adding a new type of packet.

In Contrary to MPICH-vcl, there is no a specific checkpoint scheduler server to start checkpoint waves. This role is now dedicated to the MPI process of rank 0.

f) Checkpoint implementation details: The same checkpoint server as in MPICH-V is used to store MPICH2-pcl checkpoint images. As explain in section III, a process starts taking its image only after it receives all markers and sent all markers. At this time, the process forks to create its checkpoint file in the same way as in MPICH-vcl while the main process releases the delayed requests and continues the MPI computation. When the clone ends the checkpoint, the SIGCHLD signal is delivered to the main process that sends a message to the MPI process of rank 0 such that a new checkpoint wave can be scheduled.

g) Runtime: MPD and FTPM: MPICH2 introduces a new process management environment called MPD. It consists in having a persistent daemon on every node of the system for launching MPI jobs. All these daemons are connected in a ring topology. This avoid the use of sequential ssh commands to start a job. When launching a job on \( n \) nodes, \( n \) MPD fork to create process managers (PMs). Then the process managers fork to execute \( n \) MPI processes. The different MPI processes are not connected together at the start of the execution. Two MPI processes connect themselves only from the first communication request between them. The role of process manager is to provide informations about the different nodes locations. In the current MPICH2 implementation, the MPD is
known to be fault tolerant but the process manager is not. When a failure occurs, all the PMs and the MPI processes of the job are killed.

Our implementations of fault tolerant protocols include checkpoint servers. When a failure occurs, all the non-failed processes must be killed but not the checkpoint servers. This could be implemented by the MPD architecture using two jobs: one for the computing nodes and one for the checkpoint servers. However, computing nodes need to locate the checkpoint servers. The MPD implementation does not provide means to get information from other jobs. We propose to add a new concept in the MPD architecture: the process group. A job would consist in one or many process groups, each process being managed by a process manager.

Rather than modifying the process manager, we implement a simpler environment (FTPM) to start, manage, detect failures and restart applications. The FTPM is composed of an mpiexec and PMs. In FTPM, there is no MPD daemon and we use a modified version of PMs: checkpoint servers are now launched through them. We also modify the machinefile format in order to add the specification of the mapping between machines used as computing nodes and machines used as checkpoint server.

At run time, mpiexec computes which computing nodes are used as MPI processes and launches the corresponding checkpoint servers then the processes through PMs. Process and checkpoint server spawning is done using a ssh command. To improve the execution time, these spawns are done in parallel. To avoid throttle of the node running mpiexec, the number of concurrent ssh is bounded by a parameter. In the rest of the execution, mpiexec has two roles: to monitor the MPI processes and to maintain a distributed database. Node monitoring is done in the same way as in MPICH.

Each MPI process publishes its location to the others by associating in the distributed database its rank to a business card (composed as in MPD of the process IP address, hostname and port to connect). The database is also used to store the greatest successful checkpoint wave number and to locate which checkpoint server holds which local checkpoint. Since at restart time MPI processes may be assign to spare nodes, their last local checkpoint may be not located on the local disk or on the local server associated to the running machine.

V. EXPERIMENT RESULTS

A. Experimental conditions
B. Overhead evaluation
C. Performance of fault recovery
D. Scalability issue

VI. CONCLUSION

REFERENCES


