Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications

Amina Guermouche [‡][*], Thomas Ropars [*], Elisabeth Brunet [*], Marc Snir[†], Franck Cappello [*][†]

[*] INRIA Saclay-Île de France, F-91893 Orsay, France
[‡] Université Paris Sud, F-91405 Orsay, France
[†] University of Illinois at Urbana-Champaign - College of Engineering, Urbana, IL, USA
guermou@lri.fr, tropars@inria.fr, brunet@it-sudparis.eu, snir@illinois.edu, fci@lri.fr

Abstract—As reported by many recent studies, the mean time between failures of future post-petascale supercomputers is likely to reduce, compared to the current situation. The most popular fault tolerance approach for MPI applications on HPC Platforms relies on coordinated checkpointing which raises two major issues: a) global restart wastes energy since all processes are forced to rollback even in the case of a single failure; b) checkpoint coordination may slow down the application execution because of congestions on I/O resources. Alternative approaches based on uncoordinated checkpointing and message logging require logging all messages, imposing a high memory/storage occupation and a significant overhead on communications. It has recently been observed that many MPI HPC applications are *send-deterministic*, allowing to design new fault tolerance protocols. In this paper, we propose an uncoordinated checkpointing protocol for send-deterministic MPI HPC applications that (i) logs only a subset of the application messages and (ii) does not require to restart systematically all processes when a failure occurs. We first describe our protocol and prove its correctness. Through experimental evaluations, we show that its implementation in MPICH2 has a negligible overhead on application performance. Then we perform a quantitative evaluation of the properties of our protocol using the NAS Benchmarks. Using a clustering approach, we demonstrate that this protocol actually succeeds to combine the two expected properties: a) it logs only a small fraction of the messages and b) it reduces by a factor approaching 2 the average number of processes to rollback compared to coordinated checkpointing.

I. INTRODUCTION

Fault tolerance is becoming a major issue in very large scale HPC systems [18], [7]. At exascale, some projections estimate a mean time between failures (MTBF) between 1 day and a few hours [7]. This paper focuses on fault tolerance for large scale HPC applications. We consider MPI (Message Passing Interface) applications since MPI is widely used in the HPC community. Fault tolerance for MPI HPC applications relies on rollback-recovery checkpointing protocols to ensure correct ex-

ecution termination despite failures. To design checkpointing protocols adapted to the MTBF of very large scale platforms, the main issues that have to be addressed are: i) the cost of checkpointing because the checkpointing frequency should be high to ensure that applications will progress, increasing the impact of checkpointing on application performance; ii) the cost of recovery because rolling back millions of cores after a failure would lead to a massive waste of energy.

The two main families of checkpointing protocols are coordinated and uncoordinated checkpointing. Coordinated checkpointing is the most popular in HPC production centers. Coordinating processes at checkpoint time ensures that the saved global state is consistent. It has two main advantages: i) recovery is simple since after a failure every process is restarted from its last checkpoint; ii) garbage collection is efficient since only the last checkpoint of every process is needed. However coordinated checkpointing also has major drawbacks. First, coordinated checkpointing is expensive regarding energy consumption because a single failure makes all processes rollback to their last checkpoint [10]. Second, having all processes writing their checkpoints at the same time creates burst accesses to the I/O system that may slow down the application execution [15].

Uncoordinated checkpointing does not require any synchronization between the processes at checkpoint time. Thus, it can be used to address the problem of burst accesses to the I/O system by allowing to better schedule checkpoints. However uncoordinated checkpointing has a major drawback called the *domino effect*: if no set of checkpoints form a consistent global state, the application has to be restarted from the beginning in the event of a failure. It makes recovery cost unacceptable and garbage collection complex to implement [10].

By reducing the set of applications considered to *piecewise deterministic* applications, other protocols improve uncoordinated checkpointing. For these applications, uncoordinated checkpointing can be combined

with message logging to avoid the *domino effect* and limit the number of processes to rollback in the event of a failure [1]. However, this comes at the expense of logging all application messages, which consumes storage space and adds a significant overhead on communication bandwidth.

A new property, called *send-determinism*, has recently been shown to be common to many MPI HPC applications [8]. In a send-deterministic application, given a set of input parameters, the sequence of message emissions, for any process, is the same in any correct execution. Reducing the set of applications to send-deterministic applications opens the opportunity to design new rollback-recovery protocols.

In this paper, we present a new uncoordinated checkpointing protocol for send-deterministic applications. This protocol only requires to log a small subset of application messages to avoid the domino effect. Furthermore, as in message logging protocols, it does not require all processes to rollback in the event of a failure. Since it does not suffer from the domino effect, simple and efficient garbage collection can be done. Finally, by allowing checkpoint scheduling, this protocol appears to be a good candidate to address the problem of burst accesses to the I/O system.

We present the following contributions: in section III, we introduce our protocol and describes its algorithms. In section IV, we prove that these algorithms always lead to a correct execution, despite the presence of failures. Section V demonstrates experimentally that this protocol has a low overhead on failure free execution. As a last contribution, we show on the NAS benchmarks that combining our protocol with process clustering provides a unique set of properties: 1) it does not rely on checkpoint coordination, 2) it does not suffer from the domino effect, 3) on the considered applications, the average number of processes to rollback in case of failure is close to 50% and 4) it requires to log only a small percentage of executions messages.

II. Context

In this section, we first define the system and application models considered in this paper. Then we recall the basic issues that any rollback-recovery protocol has to solve and explain how *send-determinism* can help.

A. System and Application Model

We consider an asynchronous distributed system. We model a parallel computation as consisting of a finite set of processes and a finite set of channels connecting any (ordered) pair of processes. Reliable FIFO channels are assumed. There is no bound on message transmission delay and no order between messages sent on different channels. Message exchanges create dependencies between processes. Sending and receiving event ($send()$ and $receive()$) of application processes are partially ordered by Lamport's *happened-before* relation denoted "$\rightarrow$" [13]. In this paper, "$m1 \rightarrow m2$" means that $receive(m1) \rightarrow send(m2)$. Regarding failures, we consider a fail-stop failure model for the processes and assume that multiple concurrent failures can occur.

We consider send-deterministic applications. In a send-deterministic application, each process sends the same sequence of messages for a given set of input parameters in any valid execution. Deterministic applications are a subset of the send-deterministic applications for which, per process message receptions sequence is also always the same. The protocol proposed in this paper targets send-deterministic applications and so, also applies to deterministic applications. Thus, it covers a large fraction of the MPI HPC applications [8].

B. Reaching a Consistent Global State after a Failure

To deal with failures in message passing applications, a rollback-recovery protocol is needed to manage causal dependencies between processes and ensure that a consistent global state will eventually be reached.

In a non send-deterministic application, when the sender of a delivered message rolls back due to a failure and not the receiver, the global state becomes inconsistent because there is no guaranty that the same message will be sent again during recovery. The receiver becomes *orphan* because its state depends on a message that is seen as not sent. By extension, we call this message an *orphan message*. To deal with this problem, protocols based on checkpointing force the orphan process to rollback in a state preceding the orphan message reception. If processes checkpoints are not coordinated, this may lead to the *domino effect*.

Protocols combining checkpointing and message logging limit the number of rolled back processes by logging all the information that are needed to be able to replay the sequence of messages that led to send the orphan messages. These protocols consider that processes are piecewise deterministic and so, a total order in message delivery is required during recovery.

In a send-deterministic application, sequence of message sendings are always the same in any correct execution, given a set of input parameters. It implies that ensuring causal delivery order during recovery is enough to ensure that orphan messages will be sent again. Thus send-determinism allows us to design a protocol that avoids the domino effect in uncoordinated checkpoint

while logging less information than existing message logging protocols.

III. AN UNCOORDINATED CHECKPOINTING PROTOCOL FOR SEND-DETERMINISTIC APPLICATIONS

We propose an uncoordinated checkpointing protocol for send-deterministic applications that only logs a subset of the application messages to avoid the domino effect. Thanks to send-determinism, our protocol is able to recover the execution from an inconsistent global state and thus does not require every process to rollback after a failure. Recovery is based on partial message logging and partial re-execution. In this section, we first present the main principles of our protocol and then provide a complete description including pseudo-code.

A. Protocol Principles

1) Taking Advantage of Send-Determinism: Send-deterministic processes send the same sequence of messages in any correct execution: the delivery order of non causally dependent messages has no impact on the processes execution. It allows us to design a protocol that is able to replay orphan messages after a failure without logging all messages during failure free execution.

To replay orphan messages, our protocol re-executes the failed processes from their most recent checkpoint to send the messages again. To provide the messages needed by a failed process for its re-execution, processes that sent it a message before the failure also have to rollback to send these messages again.

On Figure 1, if process P_1 fails, it rolls back to its last checkpoint H_1^2 (2 being the checkpoint number). Processes P_0 and P_2 have to rollback to replay messages m_8 and m_9. We call m_8 and m_9, *rolled back* messages. When P_1 rolls back, message m_{10} becomes orphan. However, process p_3 does not need to rollback because m_{10} will always be replayed regardless of the reception order of m_8 and m_9 during re-execution.

2) Avoiding the Domino Effect: Existing uncoordinated protocols suffer from the domino effect because orphan processes need to be rolled back and a cascading effect may force some processes to restart from the beginning of the execution. Since our protocol does not rollback orphan processes, it does not suffer from this domino effect.

However, since our protocol rolls back processes that sent rolled back messages, it creates another risk of domino effect: a rolled back message sent before a checkpoint forces the sender to roll back to a previous checkpoint. To identify these messages, our protocol uses *epochs*. The process epoch is incremented when
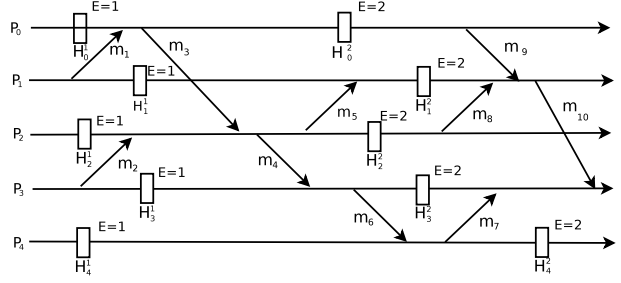

Fig. 1: Execution Scenario

a checkpoint is taken. To avoid the domino effect, messages sent at epoch Es and received at epoch Er with $Es < Er$ are logged: we log messages going from the *past* to the *future*. On Figure 1, if process P_3 fails, logging message m_7 avoids the domino effect that would have made every process rolls back to epoch 1 or even before. Since m_7 is logged, process P_4 is able to send it again without rolling back. Sender-based message logging can be used as in other message logging protocols [11].

3) Managing Causal Dependencies: During recovery, causal delivery order must be ensured to guarantee that orphan messages are re-sent. Since the protocol does not rollback orphan processes and replays logged messages, messages that are causally dependent could be sent at the same time during recovery and delivered in an order that does not respect causal precedence, leading to an incorrect state. Figure 2 illustrates the problem. In this example, process P_2 fails and rolls back to its last checkpoint. Our protocol also makes P_4 rolls back to its last checkpoint because of message m_6. Messages m_0 and m_2 are logged. We assume that m_7 is the next message that would have been sent to P_2 if the failure did not occur. When recovery starts, messages m_0, m_2, m_6 and m_7 can be sent to P_2. From P_2 point of view, it is *a priori* impossible to find the causal order between them. This is why we need additional information to deal with causal dependencies during recovery.

In our protocol, rollbacks follow causal dependency paths since rolling back the receiver of a message makes the sender roll back. Checkpoints and logged messages are used to *break* these paths and avoid restarting every process from the beginning. As a consequence, messages depending on orphan messages can be replayed during recovery. For example, it is because m_2 is logged that it could be re-sent by P_3 before m_0 is replayed, even if m_2 depends on orphan message m_1.

To deal with this problem, we introduce execution *phases*: all messages on a causality path that is not
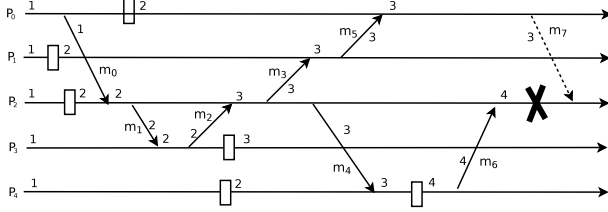
3

Fig. 2: Example of Causality Problem. Numbers show the evolution of the phases.

interrupted by a *break* are in the same phase. The phase is incremented every time the path is broken. For this, we define three rules: (i) A process increments its phase on a checkpoint; (ii) When a process receives a message, its phase becomes the maximum between its phase and the one of the message[1]; (iii) If the received message is a logged message, the process updates its phase to the maximum between the phase of the message plus one and the phase of the process to show that the causality path is broken.

During recovery, phases are used to ensure that a message is sent only when the orphan messages it depends on are replayed. A replayed message m (m_6 on Figure 2) is sent when all orphan messages in a phase $\rho < \rho'$ are replayed, since all messages in phase ρ that m depends on have been rolled back. Messages sent without re-executing the code generating the message, i.e. logged messages (m_0 and m_2 on Figure 2) and messages sent by non rolled back processes (m_7 on Figure 2), can be sent when all orphan messages in a phase $\rho' \leq \rho$ are replayed.

Figure 2 shows processes and messages phases. At the beginning only m_0 can be sent. Message m_2 can be sent when orphan message m_1 is replayed. Messages m_6 and m_7 can be replayed when message m_3 and m_4 are replayed. This is enough to ensure causal delivery order during recovery.

Note that phases are mandatory even if the application is deterministic. The deterministic message delivery order during failure free execution might be due to causal precedence. In an MPI application, an anonymous *recv()* call always matches the same message if it is the only message that can be received at that time. Since our protocol can make the application restart from an inconsistent global state, it might not be true anymore during recovery.

4) Garbage Collection: Since our protocol logs all messages going from the *past* to the *future*, a process rolling back in an epoch E cannot make another process

[1]The phase of a message is the phase of its sender.

roll back in an epoch less than E. Thus in case of multiple concurrent failures, processes roll back at most into the smallest epoch failed processes roll back to. On Figure 1, for example, the protocol ensures that for any failure, processes rollback at most in epoch 2.

This property makes garbage collection simple. If E is the smallest current epoch in the application, checkpoints in an epoch less than E can be deleted as well as logged messages received in a epoch less than E. A periodic global operation can be used to evaluate E.

B. Protocol Description

Figure 3 details the protocol executed by the application processes. Figure 4 presents the dedicated process, called *recovery* process, that we use to manage recovery. For the sake of clarity, we consider here a single failure.

During failure free execution, every message has to be acknowledged by the receiver (Line 27 of Figure 3) so that the sender can decide which messages have to be logged (Lines 36-37 of Figure 3). The process phase is increased every time a checkpoint is taken (Line 44 of Figure 3) and updated every time a message is received (Lines 21-24 of Figure 3). A $date$, increased every time a message is sent or received by a process, is used to identify duplicate messages (Line 20 of Figure 3).

To be able to deal with failures, every process saves, per communication channel, information during failure free execution. For every channel (P_i,P_j):

- A structure called SPE (SentPerEpoch) is used to save the reception epoch of last non logged message sent by P_i to P_j for every epoch of P_i. This information is used to know in which epoch P_i has to rollback if P_j rolls back (Line 14 of Figure 4).
- A structure called RPP (ReceivedPerPhase) is used to save the sending date of the last message received by P_i from P_j for every phase of P_i. This information is used to find the orphan messages between P_i and P_j if P_i rolls back.

During recovery, the dedicated *recovery* process is used to compute the recovery line, i.e. the set of processes to rollback and their corresponding epoch, and ensure the causal dependency order between replayed messages. When a process fails, it rolls back to its most recent checkpoint and broadcasts the epoch in which it restarts to all other processes (Lines 47-52 of Figure 3). When a process receives a failure notification, it suspends its execution and sends its SPE structure to the *recovery* process (Lines 54-56 of Figure 3). When the *recovery* process has received all processes SPE structure, it can start the recovery line computation. This

Local Variables:
1: $Status_i \leftarrow Running$ // Status of the process, $Running$, $Blocked$ or $RolledBack$
2: $Date_i \leftarrow 1; Epoch_i \leftarrow 1; Phase_i \leftarrow 1$ // Date of the current event, Epoch number and Phase number on process P_i
3: $NonAck_i \leftarrow \emptyset$ // list of messages sent by process P_i and not yet acknowledged
4: $Logs_i \leftarrow \emptyset$ // list of messages logged by P_i
5: $SPE_i \leftarrow [\bot, ..., \bot]$ // Data on the messages sent per epoch. $SPE_i[Epoch_{send}].date$ is P_i's date at the begining of $Epoch_{send}$.
6: $RPP_i \leftarrow [\bot, ..., \bot]$ // Data on the messages received per phase
7: $RL_i \leftarrow [\bot, ..., \bot]$ // $RL_i[j].epoch$ is the epoch in which P_j has to rollback after a failure; $RL_i[j].date$ is the corresponding date
8: $OrphCount_i \leftarrow [\bot, ..., \bot]$ // $OrphCount_i[phase]$ is the number of process that will re-send an orphan message to P_i in phase $phase$
9: $OrphPhases_i \leftarrow \emptyset$ // Phases where an orphan message has been received by P_i
10: $LogPhases_i \leftarrow \emptyset$ // Phases in which P_i has logged messages to replay
11: $ReplayLogged_i \leftarrow [\bot, ..., \bot]$ // $ReplayLogged_i[phase]$ is the list messages logged by P_i to be replayed in phase $phase$
12:
13: **Upon sending message** msg **to** P_j
14:      **wait until** $Status_i = Running$
15:      $Date_i \leftarrow Date_i + 1$
16:      $NonAck_i \leftarrow NonAck_i \cup (P_j, Epoch_i, Date_i, msg)$
17:      Send $(msg, Date_i, Epoch_i, Phase_i)$ to process P_j
18:
19: **Upon receiving** $(msg, Date_{send}, Epoch_{send}, Phase_{send})$ **from** P_j
20:      **if** $Date_{send} > RPP_i[Phase_i][j].date$ **then**   // msg is received for the first time
21:          **if** $Epoch_{send} < Epoch_i$ **then**   // logged
22:              $Phase_i \leftarrow Max(Phase_i, Phase_{send} + 1)$
23:          **else**
24:              $Phase_i \leftarrow Max(Phase_i, Phase_{send})$
25:          $Date_i \leftarrow Date_i + 1$
26:          $RPP_i[Phase_i][j].date \leftarrow Date_{send}$
27:          Send $(Ack, Epoch_i, Date_{send})$ to P_j
28:          Deliver msg to the application
29:      **else if** $\exists phase$ such that $Date_{send} = RPP_i[phase][j].date$ **then**   // it is the last orphan msg of one phase
30:          $OrphCount_i[phase] \leftarrow OrphCount_i[phase] - 1$
31:          **if** $OrphCount_i[phase] = 0$ **then**   // The process received all duplicated messages for this phase
32:              Send $(NoOrphanPhase, phase)$ to the recovery process
33:
34: **Upon receiving** $(Ack, Epoch_{recv}, Date_{send})$ **from** P_j
35:      Remove $(P_j, Epoch_{send}, Date_{send}, msg)$ from $NonAck_i$
36:      **if** $Epoch_{send} < Epoch_{recv}$ **then**   // logging
37:          $Logs_i \leftarrow Logs_i \cup (P_j, Epoch_{send}, Date_{send}, Phase_{send}, Epoch_{recv}, msg)$
38:      **else**
39:          $SPE_i[Epoch_{send}][P_j].epoch_{recv} \leftarrow Epoch_{recv}$
40:
41: **Upon checkpoint**
42:      Save $(Epoch_i, ImagePs_i, ReceivedPerPhase_i, SentPerEpoch_i, Logs_i, Phase_i, Date_i)$ on stable storage
43:      $Epoch_i \leftarrow Epoch_i + 1$
44:      $Phase_i \leftarrow Phase_i + 1$
45:      $SPE_i[Epoch_i].date \leftarrow Date_i$
46:
47: **Upon failure of process** P_i
48:      Get last $(Epoch_i, ImagePs_i, RPP_i, SPE_i, Logs_i, Phase_i, Date_i)$ from stable storage
49:      Restart from $ImagePs_i$
50:      $Status_i \leftarrow RolledBack$
51:      Send $(Rollback, Epoch_i, Date_i)$ to all application processes and to the recovery process
52:      Send SPE_i to the recovery process
53:
54: **Upon receiving** $(Rollback, Epoch_{rb}, Date_{rb})$ **from** P_j
55:      $Status_i \leftarrow Blocked$
56:      Send SPE_i to the recovery process
57:
58: **Upon receiving** (RL_{recov}) **from the recovery process**
59:      **if** $RL_{recov}[i].epoch < Epoch_i$ **then**   // we need to rollback
60:          Get $(RL_{recov}[i].epoch, ImagePs_i, RPP_i, SPP_i, Logs_i, Phase_i, Date_i)$ from stable storage
61:          $Status_i \leftarrow RolledBack$
62:      **for all** $phase$ such that $RPP_i[phase][j].date > RL_{recov}[j].date$ **do**   // Looking for phases with an orphan message
63:          $OrphPhases_i \leftarrow OrphPhases_i \cup phase$
64:          $OrphCount_i[phase] \leftarrow OrphCount_i[phase] + 1$
65:      **for all** $(P_j, Epoch_{send}, Date_{send}, Phase_{send}, Epoch_{recv}, msg)$ in $Logs_i$ such that $Epoch_{recv} \geq RL_{recov}[j].epoch$ **do**   // Looking for logged messages to be replayed
66:          $LogPhases_i \leftarrow LogPhases_i \cup Phase_{send}$
67:          $ReplayLogged_i[Phase_{send}] \leftarrow ReplayLogged_i[Phase_{send}] \cup (P_j, Epoch_{send}, Date_{send}, Phase_{send}, Epoch_{recv}, msg)$
68:      Send $(Orphan, Status_i, Phase_i, OrphPhases_i, LogPhases_i)$ to the recovery process
69:
70: **Upon receiving** $(ReadyPhase, Phase)$ **from the recovery process**
71:      **if** $ReplayLogged_i[Phase] \neq \emptyset$ **then**   // Send the logged messages if any
72:          Replay $msgs \in ReplayLogged_i[Phase]$
73:      **if** $(Status_i = RolledBack \wedge Phase_i = Phase + 1) \vee (Status_i = Blocked \wedge Phase_i = Phase)$ **then**
74:          $Status_i = Running$

Fig. 3: Protocol Algorithm for Application Processes

```
Local Variables:
1:  DependencyTable ← [⊥, ..., ⊥]                              // DependencyTable[j][Epoch_send][k].epoch_recv is the SPE of process j
2:  RolledBackPhase ← ∅                    // RolledBackPhase[phase] contains the list of rolled-back processes blocked in phase phase
3:  BlockedPhase ← ∅      // BlockedPhase[phase] contains the list of non rolled-back processes and logged messages blocked in phase phase
4:  NbOrphanPhase ← ∅          // NbOrphanPhase[phase] is the number of processes having at least one orphan message in phase phase
5:
6:  Upon receiving (Rollback, Epoch_rb, Date_rb) from P_j
7:      RL_recov[j].epoch ← Epoch_rb
8:      RL_recov[j].date ← Date_rb
9:      wait until DependencyTable is complete
10:     RL_tmp ← [⊥, ..., ⊥]
11:     repeat
12:         for all P_j such that RL_tmp[j] ≠ RL_recov[j] do
13:             RL_tmp[j] ← RL_recov[j]
14:             for all P_k such that DependencyTable_i[k][Epoch_send][j].Epoch_recv ≥ RL_recov[j].epoch do
15:                 RL_recov[k].epoch ← min(RL_recov[k].epoch, Epoch_send)        // If Epoch_send is taken, RL_recov[k].date is udpated too
16:     until RL_recov = RL_tmp
17:     Send RL_recov to all application processes
18:
19: Upon receiving SPE_j from P_j
20:     DependencyTable[j] ← SPE_j
21:
22: Upon receiving (OrphanNotification, Status_j ,Phase_j,OrphPhases_j, LogPhases_j) from P_j
23:     if Status_j = Rolled − back then
24:         RolledBackPhase[Phase_j] ← RolledBackPhase[Phase_j] ∪ P_j
25:     else
26:         BlockedPhase[Phase_j] ← BlockedPhase[Phase_j] ∪ P_j
27:     for all phase ∈ LogPhases_j do
28:         BlockedPhase[phase] ← BlockedPhase[phase] ∪ P_j
29:     for all phase ∈ OrphPhases_j do
30:         NbOrphanPhase[phase] ← NbOrphanPhase[phase] + 1
31:     if OrphanNotification has been received from all application processes then
32:         Start NotifyPhases
33:
34: Upon receiving (NoOrphanPhase, Phase) from P_j
35:     NbOrphanPhase[Phase] ← NbOrphanPhase[Phase] − 1
36:     Start NotifyPhases
37:
38: NotifyPhases
39:     for all phase such that ∄phase' ≤ phase ∧ NbOrphanPhase[phase'] > 0 do  // Notification for phases that do not depend on orhpan
        messages
40:         Send (ReadyPhase, phase) to all processes in BlockedPhase[phase]
41:         Send (ReadyPhase, phase) to all processes in RolledBackPhase[phase + 1]
```

Fig. 4: Algorithm for the Recovery Process

computation finishes when the application state does not include anymore non-logged messages that are seen as sent and not received (Lines 10-16 of Figure 4). For very large scale applications, computing the recovery line could be expensive because it requires to scan the table again every time a rollback is found. However the table scanning could be easily parallelized for a better scalability by dividing the table into sub-tables.

The recovery line, including the epoch in which every process starts recovery and the date at the beginning of this epoch, is then sent to all application processes (Line 17 of Figure 4). When a process receives the recovery line from the *recovery* process, it computes its list of phases containing an orphan message. For each phase, the process calculates the number of processes that will send him an orphan message, using the RPP structure (Lines 62-64 of Figure 3). Then it computes the logged messages it has to replay (Lines 65-67 of Figure 3). Finally, every process sends to the *recovery* process its

actual phase, its status (rolled back or not), the phases in which it has orphan messages and the phases of the logged messages it has to send (Line 68 of Figure 3). Then, the recovery starts.

When a process receives all orphan messages for one phase, it notifies the *recovery* process (Lines 29-32 of Figure 3). When all orphan messages in a phase less than or equal to phase ρ are replayed, the *recovery* process notifies that all non rolled back processes and all logged messages blocked in a phase less than or equal to ρ, as well as all rolled back processes blocked in a phase less than ρ can be unblocked (Lines 38-41 of Figure 4).

IV. FORMAL PROOF

In this section, we prove that our protocol ensures a valid application execution despite failures.

Definition 1: A valid execution under send deterministic assumption is an execution where:
- Each process sends its valid sequence of messages.

- Causality is respected in message delivery.

To prove this, we consider the state an application would restart from after a failure, i.e. the recovery line computed by our algorithm described on Figure 4.

We first give some definitions we use in the proof. A process is defined by a set of states, an initial state (from this set), and an ordered set of events V_p. An event can be a message sending, a message reception or an internal computation. The events $send(m) \in V_p$ and $receive(m) \in V_p$ respectively add and delete message m from channel (p, q). Each process p takes a sequence of checkpoints H_p^i, where i is the checkpoint number.

Definition 2 (Checkpoint interval and Epoch): A checkpoint interval I_p^i is the set of event between H_p^i and H_p^{i+1}. i is the epoch number of the checkpoint interval.

Let's consider a process p that rolls back to checkpoint H_p^x.

Definition 3 (Rolled back message): Let $\mathscr{R}$ be the set of rolled back messages of process p. $m \in \mathscr{R}$ if and only if $receive(m) \in V_p$ and $receive(m) \in I_p^y$ with $y \geq x$. We denote by $\mathscr{R}|(q, p)$ the subset of messages in $\mathscr{R}$ received on channel (q, p).

Definition 4 (Logged messages): Let $\mathscr{L}$ be the set of logged messages of a process q. $m \in \mathscr{L}$ if and only if for a process p, $Send(m) \in I_q^x$ and $Receive(m) \in I_p^y$ with $y \geq x$. We denote by $\mathscr{L}|(q, p)$ the subset of messages in $\mathscr{L}$ sent on channel (q, p).

Definition 5 (Replayed messages): Let $\mathscr{S}$ be the set of replayed messages of process p. $m \in \mathscr{S}$ if and only if $send(m) \in V_p$ and $send(m) \in I_p^y$ with $y \geq x$. We denote by $\mathscr{S}|(p, q)$ the subset of messages in $\mathscr{S}$ sent on channel (p, q). Note that $\mathscr{L} \cap \mathscr{S} = \emptyset$

The current date on a process p is noted $date_p(p)$. The date of an event $e \in V_p$ is $date_p(p)$ when e occurs.

Definition 6 (Orphan message): Let $\mathscr{O}$ be the set of orphan messages, and m a message on channel (p, q). $m \in \mathscr{O}$ if and only: $date(send(m)) > date_p(p)$ and $date(receive(m)) < date_p(q)$. We denote by $\mathscr{O}|(p, q)$ the subset of messages in $\mathscr{O}$ received on channel (p, q).

Note that $\mathscr{O} \cap \mathscr{R} = \emptyset$ and $\mathscr{O} \subset \mathscr{S}$. Now we introduce definitions related to phases.

Definition 7 (Set of events in a phase): Let $Ph(p)$ be the phase of a process p. Let V_p^i be the set of events on a process p in phase i. An event $e \in V_p^i$ if and only if $Ph(p) = i$ when e occurs.

Definition 8 (Phase of a message): The phase of a message $Ph_m(m) = k$ where $send(m) \in V_p^k$.

Definition 9 (Set of orphan messages in a phase): Let $\mathscr{O}_{=k}$ be the set of orphan messages in a phase k. A message $m \in \mathscr{O}_{=k}$ if and only if $Ph_m(m) = k$. We

define $\mathscr{O}_{\leq k}$ (or any other operator) as the set of orphan processes in a phase lower than or equal to k.

Definition 10 (Set of rolled back messages in a phase): Let $\mathscr{R}_{=k}$ be the set of rolled back messages in a phase k. $m \in \mathscr{R}_{=k}$ if and only if $receive(m) \in V_p^k$.

Note that sets of orphan messages are built according to message sender's phase and set of rolled back messages according to message receiver's phase.

To prove that the valid sequence of messages is sent by each process, we first prove that no message is lost in a rollback. Then we prove that all rolled back messages are eventually re-sent.

Lemma 1: Considering 2 processes p and q, p sending message on channel (p, q). If process q rolls back, $\mathscr{R}|(p, q) \setminus \mathscr{S}|(p, q) \in \mathscr{L}|(p, q)$.

Proof: Let's consider $m \in \mathscr{R}|(p, q)$ and H_q^i the checkpoint q rolls back to. As $m \in \mathscr{R}|(p, q)$ then $receive(m) \in I_q^k$ where $k \geq i$. If $m \notin \mathscr{S}|(p, q)$ then $send(m) \in I_p^j$ where $j < k$. Then $m \in \mathscr{L}$ by definition of a logged message. ∎

We state now some properties of our algorithm.

Proposition 1: Let m_i and m_j be two messages. If $m_i \rightarrow m_j$ then $Ph_m(m_i) \leq Ph_m(m_j)$. (lines 21-24 and line 44 of Figure 3)

Proposition 2 (Send replayed message condition): Let $m_j \in \mathscr{S}$. m_j is sent if and only if $\mathscr{O}_{<Ph_m(m_j)} = \emptyset$. (line 40 of Figure 4)

Proposition 3 (Send non replayed message condition): Let $m_j \notin \mathscr{S}$. m_j is sent if and only if $\mathscr{O}_{\leq Ph_m(m_j)} = \emptyset$. (line 41 of Figure 4)

We give now the condition to replay the rolled back messages of a phase x:

Lemma 2: $\forall m_i \in \mathscr{R}_{=x}$, m_i is re-sent if and only if $\mathscr{O}_{<x} = \emptyset$.

Proof: As $m_i \in \mathscr{R}$ then $m_i \in \mathscr{S}$ or $m_i \in \mathscr{L}$ according to Lemma 1.

1) If $m_i \in \mathscr{S}$, $Ph_m(m_i) \leq x$ (Line 24 of Figure 3), m_i is sent if $\mathscr{O}_{<x} = \emptyset$ according to Proposition 2.
2) If $m_i \in \mathscr{L}$: m_i is sent if $\mathscr{O}_{\leq Ph_m(m_i)} = \emptyset$ according to Proposition 3. Since $m_i \in \mathscr{L}$ and $m_i \in \mathscr{R}_{=x}$ then $Ph_m(m_i) < x$ (as the phase of the receiver is always greater than the one of the received logged message: lines 21-22 of figure 3). Thus m_i is sent if $\mathscr{O}_{<x} = \emptyset$. ∎

Lemma 3: All rolled back messages are eventually re-sent.

Proof: Let $\mathscr{O}_{min\_phase}$ be the set of orphan messages with the lowest phase. So $\forall m_i \in \mathscr{R}_{<min\_phase}$, m_i is sent according to Lemma 2. And $\forall m_j \in \mathscr{O}_{=min\_phase}$, m_j is sent according to Proposition 2.

Then $\mathscr{O}_{min\_phase} = \emptyset$ and $\forall m_i \in \mathscr{R}_{<min\_phase+1}$, m_i is sent. And so on until $\mathscr{R} = \emptyset$ and $\mathscr{O} = \emptyset$. ∎

We prove now that the causality is respected on message delivery. Causality is broken if for two messages m_i and m_j, $m_i \to m_j$, m_j is received before m_i during recovery. If all processes roll back in a state that does not depend on m_i, the condition is trivially ensured since m_j cannot be sent before m_i is replayed. We first show that, if the application is in a state where m_i and m_j can both be sent, there is an orphan message in the causality path between m_i and m_j.

Lemma 4: Let m_i and m_{i+x}, $x \geq 1$ be two messages such that $m_i \to m_{i+1} \ldots \to m_{i+x}$. If m_i and m_{i+x} can both be sent, then $\exists m \in \mathscr{O}$ (that might be m_i) such that $m_i \to \ldots \to m \to \ldots \to m_{i+x}$.

Proof: As m_i and m_{i+x} can both be sent then $m_i \in \mathscr{R} \cup \mathscr{O}$. m_{i+x} can be sent if $m_{i+(x-1)}$ is received. Either $m_{i+(x-1)}$'s sender rolls back and then $m_{i+(x-1)} \in \mathscr{O}$ or not. If $m_{i+(x-1)} \notin \mathscr{O}$, $m_{i+(x-2)}$ is received. Then $m_{i+(x-2)}$ is either an orphan or not. And so on until: m_{i+1} can be sent if m_i is an orphan or not. As $m_i \in \mathscr{R} \cup \mathscr{O}$, and m_{i+1}'s sender does not roll back (otherwise, we would have stopped in m_{i+2}), then $m_i \in \mathscr{O}$. ∎

Let's consider the relation between the phase of an orphan message and a replayed one depending on it.

Lemma 5: Let m_i and m_j be two messages such that $m_i \to m_j$. If $m_i \in \mathscr{O}$ and $m_j \in \mathscr{S}$ then $Ph_m(m_i) < Ph_m(m_j)$.

Proof: As $m_i \to m_j$ then according to Proposition 1, $Ph_m(m_i) \leq Ph_m(m_j)$. To prove that $Ph_m(m_i) < Ph_m(m_j)$ we prove that $Ph_m(m_i) = Ph_m(m_j)$ leads to a contradiction. So let's assume that $Ph_m(m_i) = Ph_m(m_j)$. According to lines 21 and 44 of Figure 3, there is no checkpoint and no logged messages on the causality path from m_i to m_j (as these two events are the only ones that increase a phase). Then according to lines 10-16 of Figure 4, if $m_j \in \mathscr{S}$, $m_i \in \mathscr{R}$. This is impossible since $m_i \in \mathscr{O}$ and $\mathscr{O} \cap \mathscr{R} = \emptyset$. ∎

We finally prove that a message depending on an orphan message cannot be sent before the orphan message is sent.

Lemma 6: If m_i and m_j are two messages such that $m_i \to m_j$ and $m_i \in \mathscr{O}$, m_j cannot be sent until m_i is sent.

Proof: If $m_j \in \mathscr{S}$, m_j is sent if and only if $\mathscr{O}_{<Ph_m(m_j)} = \emptyset$ (Proposition 2). If $m_j \notin \mathscr{S}$, m_j is sent if and only if $\mathscr{O}_{\leq Ph_m(m_j)} = \emptyset$ (Proposition 3). $m_i \in \mathscr{O}_{=Ph_m(m_i)}$. Since $m_i \to m_j$, $Ph_m(m_j) \geq Ph_m(m_i)$ according to Lemma 1.

1) If $Ph_m(m_i) = Ph_m(m_j)$: then $\mathscr{O}_{\leq Ph_m(m_j)} \neq \emptyset$. According to Lemma 5, as $Ph_m(m_i) = Ph_m(m_j)$, $m_j \notin \mathscr{S}$.

2) If $Ph_m(m_i) < Ph_m(m_j)$: then $m \in \mathscr{O}_{<Ph_m(m_j)} \neq \emptyset$. ∎

Theorem 1: After a failure, the execution is valid under the send deterministic assumption.

Proof: Lemmas 1 and 3 prove that after a failure, all processes are able to send their valid sequence of messages. Lemma 6 shows that our protocol enforces causal order in messages delivery. ∎

The proof for multiple simultaneous failures is the same since a failed process and a rolled back one have the same behavior: all the information needed is included in the checkpoint.

V. EVALUATION

In this section, we first describe our prototype developed in MPICH2. Then we present experimental evaluations. We first evaluate the performance on failure free executions. Then we evaluate the number of processes to rollback in case of failure and propose a solution based on processes clustering to limit the number of rollbacks while logging only a small percentage of the messages.

A. Prototype Description

We implemented our protocol in MPICH2, in the Nemesis communication subsystem. The protocol can be used over TCP and MX/Myrinet. We have integrated process uncoordinated checkpointing to the Hydra process manager. Process checkpointing is based on BLCR.

We have seen in Section III-B that the protocol requires all messages to be acknowledged with the reception epoch to control the logging. But sending an additional message to acknowledge every application message would impair the communication performance, especially the latency of small messages on high performance networks: a send operation cannot terminate until the corresponding acknowledgment is received.

Our implementation relies on the FIFO channels to limit the number of acknowledgments. This is illustrated on Figure 5 and we focus on the messages sent by process P_1. Instead of waiting for an acknowledgment for every message, small messages content is copied by default. Thus a *send()* operation can return without waiting for the acknowledgment. Each message is identified by a sender sequence number (ssn). Only the first message per epoch and per channel that has to be logged (m_4) is acknowledged. For other messages, processes piggyback on the messages they send, the ssn of the last message they received. When P_2 sends message $m3$ to process P_1, it piggybacks $ssn = 2$ on it to make P_1 aware that it can delete messages m_1 and m_2 from the

8

logs. After the first logged message (m_4) has been acknowledged, we know that all messages sent to P_2 have to be logged until the epoch of P_1 changes. We mark these messages (m_5) as already logged to avoid making P_2 send additional acknowledgments. If the number of non-acknowledged messages becomes too large because P_2 never sends messages to P_1, P_1 can explicitly require an acknowledgment from P_2. For large messages, we cannot afford an extra copy of every message content, so these messages are always acknowledged except when they are marked as already logged.
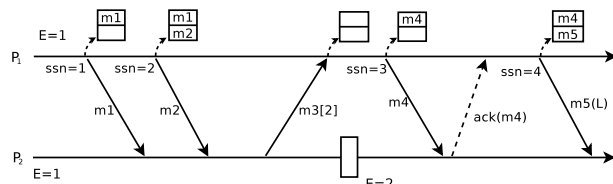

Fig. 5: Protocol Acknowledgment Management for Small Messages on a Communication Channel

B. Experimental Setup

Our experiments were run on Grid'5000. Performance evaluations were run on Lille cluster using 45 nodes equipped with with 2 Intel Xeon E5440 QC (4 cores) processors, 8 GB of memory and a 10G-PCIE-8A-C Myri-10G NIC. Experiments evaluating the number of messages logged and the number of processes to rollback were run on Orsay cluster using 128 nodes equipped with 2 AMD Opteron 246 processors. In both cases, the operating system was Linux with kernel 2.6.26.

C. Ping-Pong Performance on Myrinet

To evaluate the overhead of our protocol on communication performance, we run a ping-pong test using Netpipe [19] on 2 nodes of the Lille Grid'5000. Results are presented on Figure 6. In this experiment, the limit for small messages, i.e. messages that do not require an explicit acknowledgment, is set to 1 KB. The figure compares the native performance of MPICH2[2] to the performance of MPICH2 with our protocol when no messages are logged, and when all messages are logged. The measure of the protocol performance when no message are logged evaluates our optimized management of the acknowledgments. We observe that latency overhead of our protocol with and without message logging, compared to the native MPICH2 for small messages is around 15% ($0.5\mu s$). This overhead is due to the management of the data piggybacked on

[2]https://svn.mcs.anl.gov/repos/mpi/mpich2/trunk:r7262

messages to avoid sending acknowledgments. Regarding large messages (from 64KB), the performance of the protocol without logging show that acknowledging every message has a negligible overhead compared to native MPICH2. However, message logging has a significant impact on bandwidth for these messages because of the extra message content copy that is required.

D. NAS Parallel Benchmark Performance

We ran 3 of the NAS Parallel Benchmarks (BT, CG and SP) on Lille cluster. Figure 7 presents the overhead induced by our protocol when no messages are logged and when all messages are logged, compared to MPICH2 native performance. Results show that the protocol induces no overhead when messages are not logged and only a very small (less than 5%) overhead when all messages are logged. However, the overhead might be higher with communication-bound applications using larger messages. Even if logging all messages has little impact on applications performance, it consumes extra storage, and so should be limited.

E. Quantitative Evaluation of Protocol Properties

In this section, we evaluate quantitatively, using the NAS benchmarks, the proportion of logged messages during the execution and the number of processes to roll back when a failure occurs.

1) Methodology: To compute the number of processes to roll back, the SPE table of all processes, described in Section III-B, is saved every 30s during the execution. We analyze these data offline and run the recovery protocol: for each version of SPE, we compute the rollbacks that would be induced by the failure of each process. Then, we can compute an estimation of the average number of processes to roll back in the event of a failure during the execution.

2) Uncoordinated Checkpointing: We ran some experiments with uncoordinated checkpoints and random checkpoint time for each process and noticed that a small number of messages need to be logged. However, in all these experiments, all processes need to roll back in the event of a failure: taking checkpoints randomly does not create any *consistent* cut in causal dependency paths.

3) Process Clustering: To address this issue, message logging should be forced according to the communications patterns of the applications. The objective is to isolate, as much as possible, the processes that communicate frequently and form "clusters". Clustering will limit the rollback propagations. In our protocol, a message is logged if it is sent from an epoch to a larger one. Different epochs can then be used over clusters to log messages. This approach is not symmetrical: the
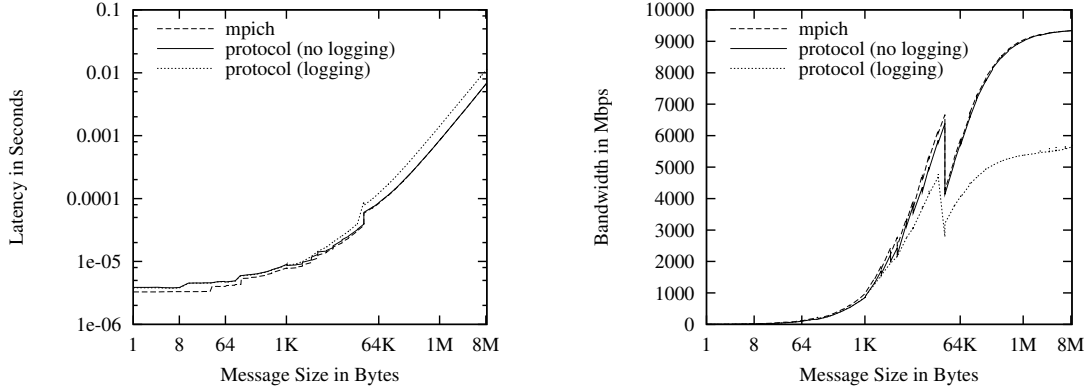
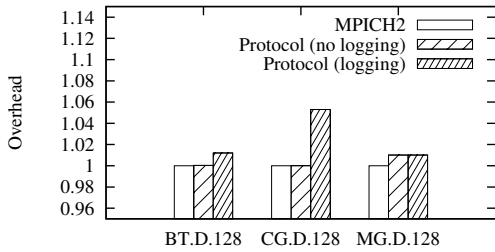Fig. 6: Myrinet 10G Ping-Pong Performance

Fig. 7: NAS Performance on Myrinet 10G

messages sent from the cluster having the largest epoch will never be logged. So processes of a cluster will always need to restart if at least one process with a smaller epoch rolls back. However, statistically, for a large number of failures, we can assume that failures will be distributed evenly between clusters.

Process clustering should minimize two factors : (i) the number of logged messages (or the volume of logged messages) and (ii) the number of processes to rollback in the event of failure. While it is out of the scope of this paper to present a detailed study on how to optimize clustering, we present an example of clustering demonstrating that our protocol can achieve our initial objectives: log only a fraction of the application messages while avoiding global restart. In the following experiment, to find a good clustering, we favor two parameters: a) locality: maximize the intra-cluster communications and b) isolation: minimize the inter-cluster communications. For every NAS benchmark, we have analyzed the communication topology to maximize locality and isolation. Figure 8 shows the communication density between all processes for CG and MG for 64 processes. It also shows the clustering, in squares, and

the epochs at the beginning of the execution. Only inter cluster messages from clusters in an epoch to clusters with larger epoch are logged. Cluster epochs are separated by a distance of 2 to ensure that when processes of one cluster are checkpointed, they do not reach the same epoch as another cluster. For the other NAS benchmarks, we used the communication patterns provided in [16]. The average number of clusters to roll back can be computed theoretically. Let us consider p clusters numbered from 1 to p and a pessimistic scenario where the failure of a process leads all processes of the same cluster to roll back. Assuming that E_i is the epoch of cluster i, thanks to logged messages, if a cluster j fails, all clusters with $E_i < E_j$ do not need to roll back. If the cluster with the smallest epoch E_s fails, p clusters will roll back. If the cluster with epoch E_{s+2} fails, only clusters with $E_i > E_{s+2}$ roll back, *i.e.* $(p-1)$ clusters. If the cluster with the largest epoch fails, only this cluster rolls back. Let us consider p executions, one failure per execution, failures being evenly distributed over clusters. The total number of clusters to roll back over the p executions is $p*(p+1)/2$, that is $(p+1)/2$ on average. When p is large, this average approaches 50%.

We ran experiments to evaluate the percentage of logged messages ($\%log$) and processes to rollback ($\%rl$), for the 5 NAS kernels in class D for 64, 128 and 256 processes, with 4, 8 and 16 clusters. The results are shown in table I. We observe that the percentage of logged messages decreases when the number of processes per cluster increases. However, given a fixed number of processes, using more and thus smaller clusters reduces the number of processes to roll back in the event of failure for all the NAS kernels. Thus there is a trade-off to consider between the size of the storage space used for message logging and the energy consumption for
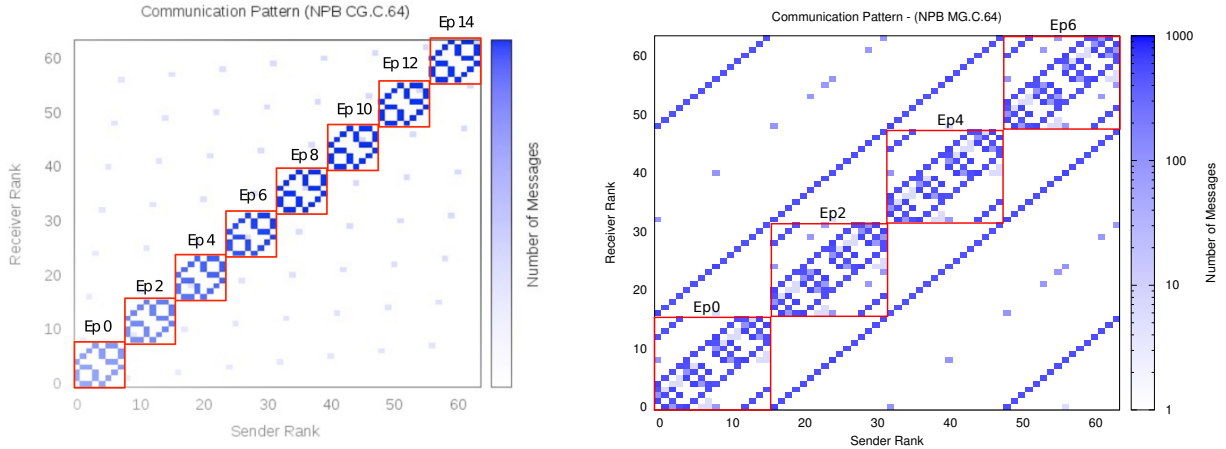
10

Fig. 8: Communication Density Graph and Clustering in CG and MG

Size	64				128						256					
Cluster	4		8		4		8		16		4		8		16	
	%log	%rl	%log	%rl	%log	%rl	%log	%rl	%log	%rl	%log	%rl	%log	%rl	%log	%rl
MG	16.59	62.5	25	46.9	9.5	62.5	17.1	56.3	25.4	42.1	9.1	62.5	16.9	55.7	24.9	41.7
LU	10.72	62.5	25	56.2	10.35	62.5	24.14	56.3	25.9	42.1	5	62.5	11.7	56.1	25.03	53.1
FT	37.2	62.4	43.6	59	37.3	62.5	43.6	56	46.8	53	37.4	62.3	43.6	55.8	46.8	52.5
CG	3.8	62.5	4.4	56.3	2.9	62.5	3.4	56.3	15	43.8	2.9	62.5	3.4	56.3	3.6	53.1
BT	16.7	62.5	33.3	56.3	13	62.6	25.2	56.4	36.7	53.3	8.3	62.5	16.7	56.25	33.3	53.12

TABLE I: Number of Logged Messages and Rolled Back Processes According to Cluster Size

restart. If energy needs to be reduced, the best option is to use many clusters. If storage space is to be minimized, large clusters is the best option. The kernel illustrating the most the effect of our protocol is CG: for 256 processes and 16 clusters, less than 4% of the messages are logged while the number of processes to rollback in the event of failure is about half of what a coordinated checkpointing protocol requires. This is mostly due to the clustered nature of the communications in CG. On the other hand, FT uses many all-to-all communications and so clustering has a limited effect on the reduction of the number of messages to log. However, the percentage of logged messages can always be limited to 50%. If we consider a clustering configuration, 3 sets of messages can be defined: A, the intra-cluster messages; B, the logged inter-cluster messages; C, the non-logged inter-cluster messages. If B includes more than 50% of the messages, a simple reconfiguration of the epochs over the clusters allows making C (less than 50%) being logged instead of B.

VI. RELATED WORK

As mentioned in the introduction, many fault tolerant protocols have been proposed in the past [10]. They are divided in 4 classes: uncoordinated checkpointing [4],

coordinated checkpointing [9], [14], message logging [1] and communication induced checkpoint [3].

In theory, coordinated checkpointing protocol allows partial restart. In [12], the authors propose a blocking coordinated checkpointing protocol that coordinates only dependent processes and thus may restart only a subset of processes. However we have observed that for most MPI HPC applications, the communication patterns and the frequency of communications between two consecutive checkpoints eventually lead to establish a global dependency between processes: the failure of any process would lead to a global restart.

Messages logging protocols need either to log reception orders on stable storage (pessimistic and optimistic) or to piggyback them on application messages (causal) to ensure a total delivery order. All these approaches have a significant impact on communication and application performance [5]. In [6], authors use determinism in MPI applications to reduce the number of determinants to log. In [17], the authors extend the optimistic assumption made in optimistic message logging to improve performance. However, all these protocols still log all message contents during the execution.

Communication Induced Checkpoint (CIC) avoids the domino effect in uncoordinated checkpointing. Processes

take independent checkpoints. In addition, some checkpoints are forced to guarantee the progress of the recovery line. However, evaluations of this protocol has shown that CIC protocols do not scale well with the number of processes. They induce more checkpoints than needed for a given meantime between failures [2].

VII. Conclusion

This paper presents a novel uncoordinated checkpointing protocol for send-deterministic HPC applications. This protocol provides an alternative to coordinated checkpointing and message logging protocols by featuring a unique set of properties: 1) it does not rely on checkpoint coordination nor synchronization, 2) it does not suffer from the domino effect, 3) it logs only a fraction of the execution messages. We provided a detailed description of the protocol and proved that it can tolerate multiple concurrent failures. Experimental results show that the performance of our optimized implementation of the protocol in the MPICH2 library is very closed to the native performance of MPICH2 on MX/Myrinet. Furthermore, we proposed a solution based on clustering to adapt our protocol to the applications communication patterns. Using 5 kernels of the NAS Benchmarks, we demonstrated experimentally that this solution logs only small percentages of the application messages and reduces the number of processes to restart in case of failure to a factor approaching 2 compared to coordinated checkpointing.

As future work, we plan to explore further the association of send-determinism and clustering to further reduce the number of processes to rollback and the number of messages to log. We also plan to study if send-determinism can be used to improve other classes of rollback-recovery protocols.

References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[2] L. Alvisi, S. Rao, S. A. Husain, d. M. Asanka, and E. Elnozahy. An Analysis of Communication-Induced Checkpointing. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 242, Washington, DC, USA, 1999. IEEE Computer Society.

[3] R. Baldoni. A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. *International Symposium on Fault-Tolerant Computing*, 0:68, 1997.

[4] B. Bhargava and S.-R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - an Optimistic Approach. pages 3–12, 1988.

[5] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI . *International Journal of High Performance Computing Applications*, 20(3):319–333, 2006.

[6] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[7] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[8] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (IC-CCN 2010)*, 2010.

[9] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[11] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[12] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[14] N. Neves and W. K. Fuchs. Using Time to Improve the Performance of Coordinated Checkpointing. In *Proceedings of the International Computer Performance & Dependability Symposium*, pages 282–291, 1996.

[15] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[16] R. Riesen. Communication Patterns. In *Workshop on Communication Architecture for Clusters CAC'06*, Rhodes Island, Greece, Apr. 2006. IEEE.

[17] T. Ropars and C. Morin. Active Optimistic Message Logging for Reliable Execution of MPI Applications. In *15th International Euro-Par Conference*, pages 615–626, Delft, The Netherlands, August 2009. Springer-Verlag.

[18] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78:012022 (11pp), 2007.

[19] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.