Future Generation Computer Systems 91 (2019) 450–464

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Local rollback for resilient MPI applications with application-level checkpointing and message logging

Nuria Losada [a,*], George Bosilca [b], Aurélien Bouteiller [b], Patricia González [a], María J. Martín [a]

[a] Computer Architecture Group, Universidade da Coruña, Spain
[b] Innovative Computing Laboratory, The University of Tennessee, Knoxville, USA

HIGHLIGHTS

- A local rollback solution for MPI resilient programs preventing survivors rollback.
- Integration of ULFM, application-level checkpointing, and message logging.
- Split message logging between library-level and application-level.
- Application-level collective operations logging improves portability and log size.

ARTICLE INFO

Article history:
Received 18 February 2018
Received in revised form 10 July 2018
Accepted 17 September 2018
Available online xxxx

Keywords:
MPI
Resilience
Message logging
Application-level checkpointing
Local rollback

ABSTRACT

The resilience approach generally used in high-performance computing (HPC) relies on coordinated checkpoint/restart, a global rollback of all the processes that are running the application. However, in many instances, the failure has a more localized scope and its impact is usually restricted to a subset of the resources being used. Thus, a global rollback would result in unnecessary overhead and energy consumption, since all processes, including those unaffected by the failure, discard their state and roll back to the last checkpoint to repeat computations that were already done. The User Level Failure Mitigation (ULFM) interface – the last proposal for the inclusion of resilience features in the Message Passing Interface (MPI) standard – enables the deployment of more flexible recovery strategies, including localized recovery. This work proposes a local rollback approach that can be generally applied to Single Program, Multiple Data (SPMD) applications by combining ULFM, the ComPiler for Portable Checkpointing (CPPC) tool, and the Open MPI VProtocol system-level message logging component. Only failed processes are recovered from the last checkpoint, while consistency before further progress in the execution is achieved through a two-level message logging process. To further optimize this approach point-to-point communications are logged by the Open MPI VProtocol component, while collective communications are optimally logged at the application level—thereby decoupling the logging protocol from the particular collective implementation. This spatially coordinated protocol applied by CPPC reduces the log size, the log memory requirements and overall the resilience impact on the applications.

1. Introduction

Today, high-performance computing (HPC) plays an integral role in the advancement of many science and engineering disciplines. However, recent studies show that, as HPC systems continue to grow in numbers and in heterogeneity, the Mean Time To Failure (MTTF) for a given application shrinks, resulting in a high failure rate overall. Di Martino et al. [1] studied the US National Center for Supercomputing Applications' (NCSA's) "Blue Waters" supercomputer for 261 days and found that 1.53% of applications running on the machine failed because of system-related issues. The electricity cost of not using any protective mechanisms in the failed applications was estimated at almost $500,000 during the period of time studied. Future exascale systems will employ several million compute cores, many more than Blue Waters, and will accordingly be hit by errors and faults more frequently due to their scale and complexity. Therefore, long-running applications will need to rely on fault tolerance techniques not only to ensure the timely completion of their execution in these systems but also to minimize the running costs.

* Corresponding author.
E-mail addresses: nuria.losada@udc.es (N. Losada), bosilca@icl.utk.edu (G. Bosilca), bouteill@icl.utk.edu (A. Bouteiller), patricia.gonzalez@udc.es (P. González), mariam@udc.es (M.J. Martín).

https://doi.org/10.1016/j.future.2018.09.041
0167-739X/© 2018 Elsevier B.V. All rights reserved.

The Message Passing Interface (MPI) standard remains the most popular parallel programming model in HPC systems. All versions of the MPI standard, including the current version, 3.1, lack any fault tolerance support. By default, the entire MPI application is aborted upon a single process failure. Besides, even when set to return errors, the state of MPI will be undefined upon failure, and, thus, there are no guarantees that an MPI program can successfully continue its execution. For this reason, traditional fault-tolerant solutions for MPI applications rely on stop-and-restart checkpointing, where, upon a fault, and disregarding their statuses, all MPI processes are aborted and then restarted from the last checkpoint. The simplicity of this approach makes it palatable to application developers for as long as the resulting overheads, in terms of execution time and additional hardware costs, remain contained to the allotted application budget. However, this approach presents several serious disadvantages. First, in these solutions the entire application is aborted in the event of a failure, and a new MPI job needs to be relaunched. In some systems this implies the re-queueing of a new job to the scheduling system, which introduces an overhead—the size of which depends on the availability of platform resources. Second, in the general case, the re-queueing will result in the assignment of a different set of resources, forcing the movement of all checkpoint data across the cluster in order to restart the computation, which usually causes significant network contention and high overheads. Third, all processes – including the ones not affected by the failure – roll back to the last recovery line and repeat a computation already done, which introduces unnecessary overheads and energy consumption. However, in many instances a complete restart is unnecessary, since most of the computation nodes used by a parallel job will still be alive. Finally, MPI was never meant to exists only for parallel applications that target exascale platforms, but to provide a generic and portable programming paradigm that supports all types of applications including those that target smaller platforms built with different sets of requirements, where resource volatility might be a usual occurrence. Thus, more efficient solutions to allow parallel applications to cope with faults need to be explored.

The User Level Failure Mitigation (ULFM) interface [2], under discussion in the MPI Forum, proposes to extend the MPI standard with resilience capabilities to make MPI more suitable for fault-prone environments (e.g., future exascale systems). Resilient MPI programs are able to detect and react to failures without stopping their execution, thus avoiding re-spawning the entire application. ULFM includes new semantics for process failure detection, communicator revocation, and reconfiguration, but it does not include any specialized mechanism to recover the application state at failed processes. This leaves the flexibility to the application developers to implement the most optimal methodology, taking in account the properties of the target application.

The ComPiler for Portable Checkpointing (CPPC) [3] is an application-level open-source checkpointing tool for MPI applications. CPPC appears to the user as a compiler tool and a runtime library. The compiler automatically instruments the application code adding calls to the library for fault tolerance support. It locates checkpoint calls at the appropriate points of the application code, and marks the relevant variables for their inclusion in the checkpoint files by using a "liveness" analysis. CPPC applies a spatially coordinated checkpointing protocol [4] which enables processes to checkpoint independently, avoiding inter-process communications or runtime synchronization.

In this work, we propose a fault tolerance solution that combines ULFM, the CPPC checkpointing tool, and a two-level message logging protocol to transparently add resilient support to generic SPMD MPI applications. Our approach relies on a local rollback protocol where only the failed processes are recovered from the last checkpoint, while consistency and further progress of the computation is enabled using ULFM, and a split message logging protocol that operates partly at the library level and partly at the application level. A major difference with existing message logging protocols is that the collective communications are logged at the application level, which reduces the impact of collective communication on the size of logged data, while at the same time enabling the use of architecture-aware collective communications on the recovered application, speeding up the recovery process. Additionally, the spatially coordinated protocol used for checkpointing in CPPC further contributes to this memory footprint reduction, as checkpoints provide locations in which the log can be cleared.

This paper is structured as follows. Section 2 covers the related work. Section 3 introduces CPPC. Section 4 gives a global overview of the local rollback protocol, while Section 5 explains the message logging strategy. Section 6 presents the management of the communications interrupted by the failure, and Section 7 describes the tracking protocol developed to ensure the consistency of the replay process. The experimental results are presented in Section 8. Finally, Section 9 concludes this paper.

2. Related work

Though there is work in the literature that explores how to provide a fault-tolerant MPI [5–7], the ULFM interface [2] is the latest, and possibly the most conclusive, effort to include resilience capabilities in the MPI standard. ULFM is a low-level API that provides resilience constructs to support a variety of fault tolerance models and allows/requires the user to design the recovery strategy. In the literature, as well as in practice, there are a few different proposals for implementing resilient applications using ULFM [8–16], many of them specific to one or a set of applications [8,9,11–14]

In a previous work [17,18], we extended the CPPC [3] application-level checkpointing tool to use the new functionalities provided by ULFM to transparently obtain resilient MPI applications from generic MPI single program, multiple data (SPMD) programs. In this solution, all application processes roll back to the last valid recovery line; thus, all processes re-execute the computation from the checkpoint until the point where the failure occurred. In contrast, we propose here a fault tolerance solution that combines ULFM, CPPC, and a message logging protocol to avoid the rolling back of survivor processes.

Relaxing the synchronization constraint in coordinated checkpoint/restart requires replaying messages between restarted and non-restarted processes, which can be achieved by adding a message logging protocol. Message logging protocols have two fundamental parts: (1) the logging of all non-deterministic events and (2) the logging of the content of the messages. Event logging must be done reliably, and different techniques (pessimistic, optimistic, and causal), providing different levels of synchronicity and reliability, have been studied [19]. Different protocols for the payload logging, which is the logging of the contents of the messages, have also been proposed. Receiver-based approaches [20] perform the local copy of the message contents in the receiver side. The main advantage here is that the log can be locally available upon recovery; however, messages need to be committed to stable storage or to a remote repository to make them available after the process fails. On the other hand, in sender-based strategies [21–25], the logging is performed on the sender process. This is the most-used approach, as it provides better performance [26] than other approaches. In a sender-based approach, the local copy can be made in parallel with the network transfer, and processes can keep the log in their memory; if the process fails, the log is lost, but it will be regenerated during the recovery. As a drawback, during the recovery, failed process need to request the replay of messages by the survivor processes. Hybrid solutions have also been studied [27] to obtain better performance than receiver-based approaches during failure-free executions and to reduce the restart overhead of send-based approaches.

Although message logging provides a more flexible restart, the memory requirements to maintain the log represent a limiting factor. Strategies to reduce the memory consumption of the payload logging have been studied and are available in the literature. For instance, hierarchical checkpointing reduces the memory cost of logging by combining it with coordinated checkpointing [22–24]. Coordination is applied within a group of processes, while only messages between different groups are logged. When a process fails, all processes within its group have to roll back. This idea is also combined in [24] with "send-determinism" [28], to also reduce the number of logged events by assuming that processes send the same sequence of messages in any correct execution. In [29], dedicated resources (logger nodes) cooperate with the compute nodes by storing part of their message logs in the memory. Other strategies offer special consideration when logging collective communications – in order to decrease the memory footprint – by reducing the number of internal point-to-point messages of the logged collective [25] or by logging their result on a remote logger [30].

Our approach implements a local rollback protocol where only the failed processes are recovered from the last checkpoint, while consistency and further progress of the computation are enabled using ULFM and the message logging capabilities. A two-level message logging protocol is implemented: (1) at the MPI-level the VProtocol [21] message logging component is used for sender-based message logging and pessimist event logging of point-to-point communications, while (2) collective communications are optimally logged at the application level by CPPC. This strategy dissociates the collective communications from their underlying point-to-point expression, allowing the use of architecture-aware collective communications and reducing the memory footprint for their message logging. Additionally, the spatially coordinated protocol used for checkpointing by CPPC further contributes to this reduction, as checkpoints provide locations in which the log can be cleared. The proposal solves the issues that arise when a hard failure terminates one or several processes in the application by dealing with the communications that were interrupted by the failure and ensuring a consistent ordered replay so that the application can successfully resume the execution.

3. CPPC overview

The CPPC compiler automatically instruments the application code to produce an equivalent, fault-tolerant version where supplementary calls to the CPPC library add protection and recovery points. The instrumentation code includes registration and checkpoint calls. The registration calls explicitly mark the relevant variables for their inclusion in the checkpoint files—the compiler uses a "liveness" analysis to identify the variables to be registered. Each process generates a checkpoint every N calls to the checkpoint function, where N is the user-defined checkpointing frequency.

To enable users to specify an adequate checkpointing frequency, the compiler uses a heuristic evaluation of computational cost to place the checkpoint calls in the most expensive loops of the application. Checkpoint consistency is guaranteed by locating the checkpoint calls in the first "safe point" in these loops. The CPPC compiler performs a static analysis of inter-process communications and identifies safe points as code locations where it is guaranteed that there are no in-transit or inconsistent messages. Safe points allow CPPC to apply a "spatial coordination protocol" [4]. Here, processes checkpoint independently without the need of inter-process communications or runtime synchronization. Instead, processes are implicitly coordinated: they checkpoint at the same selected safe locations (checkpoint calls) and at the same relative moments according to the checkpointing frequency. Fig. 1 shows an example for a checkpointing frequency of $N = 2$. All processes checkpoint at the second, fourth, and sixth checkpoint calls, which are invoked by each process at different instants in time. The recovery line is formed by the checkpoint files generated by all of the processes at the same safe location and at the same relative moment; thus, no communications can cross the recovery line, and no communications need to be replayed during the recovery when using a global rollback with CPPC.

In the original CPPC approach, upon detection of a failure, the application is relaunched, and the restart process takes place. First, the application processes perform a negotiation to identify the most recent valid recovery line, which is formed by the newest checkpoint file simultaneously available to all processes. The restart phase has two parts: (1) reading the checkpoint data into memory and (2) reconstructing the application state. The reconstruction of the application state is achieved through the ordered execution of certain blocks of the application's code. These blocks of code move the recovered data to its proper memory location, re-create the non-portable state (e.g., the creation of communicators), and position the application control flow at the point where the checkpoint files were generated so that the execution can resume. For this purpose, the compiler inserts control flow code (labels and conditional jumps) to ensure an ordered re-execution.

4. Local rollback protocol outline

In most instances the recovery approach where all processes are restarted from the most recent valid recovery line is highly inefficient. To address this drawback, we expand upon CPPC to enable local rollback recovery. The goal of the local rollback protocol is to reach, in the event of a failure, a consistent global state from which the application can resume the execution by rolling back only the failed processes. Fig. 2 shows a global overview of the operation. In the left part of the figure, the application is executed normally until a failure occurs. The point of the execution where the failure takes place, from the survivors' perspective, is called the "failure line". The right part of the figure shows the recovery using the local rollback protocol, which is split into two phases: (1) the "process recovery" phase detects the failure and re-spawns failed processes, and (2) the "consistency recovery" phase leads the application to a consistent state from which the execution can resume.

During the processes recovery, CPPC exploits the ULFM functionalities to avoid terminating the application in the event of a failure. The default MPI error handler is replaced with a custom one, which is invoked upon process failure. Within the error handler, survivors revoke all their communicators to ensure global knowledge of the failure, agree about the failed processes, and then re-spawn them. Together, all processes reconstruct the global communicator of the application (conceptually similar to MPI_COMM_WORLD), and then rebuild all revoked communicators. The fact that all communicators are revoked to ensure failure detection implies that all communicators need to be reconstructed, by substituting all failed processes with their new replacement processes. CPPC tracks all communicators used by the application at compile time, and the CPPC compiler replaces the communicators in the application with a custom CPPC communicator that contains a pointer to the underlying MPI communicator actually used by MPI. With this approach, all communicators used by the application are known to CPPC, and they can be revoked and transparently substituted with their repaired replacement.

In the consistency recovery phase, the state of a failed process is recovered using the checkpoint file from the last valid recovery line. Then, to reach a consistent global state, failed processes need to progress between that recovery line and the failure line.

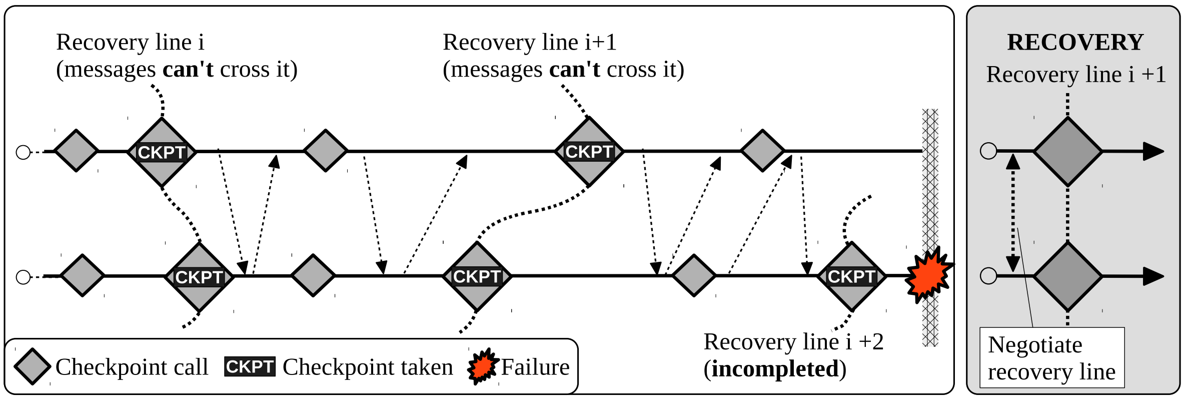


Fig. 1. Spatial coordination protocol.

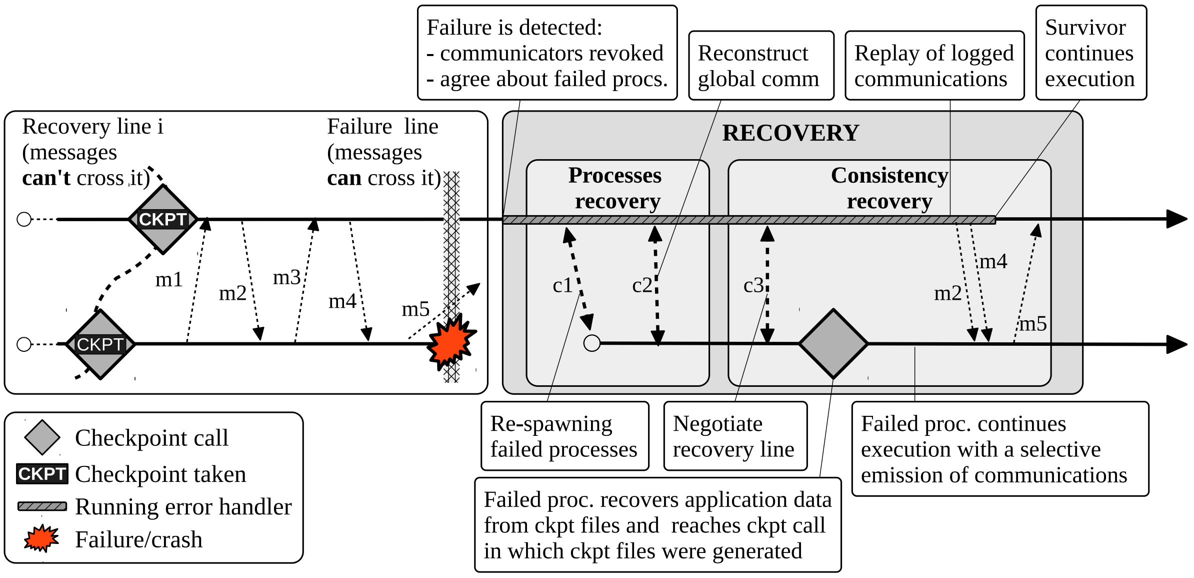

Failure is detected:
- communicators revoked
- agree about failed procs.
Reconstruct global comm
Replay of logged communications
Survivor continues execution
Recovery line i (messages can't cross it)
Failure line (messages can cross it)
RECOVERY
Processes recovery
Consistency recovery
CKPT
m1
m2
m3
m4
m5
c1
c2
c3
Re-spawning failed processes
Negotiate recovery line
Failed proc. continues execution with a selective emission of communications
Failed proc. recovers application data from ckpt files and reaches ckpt call in which ckpt files were generated
Checkpoint call
Checkpoint taken
Running error handler
Failure/crash

Fig. 2. Local rollback.

In order to reach the same state as before the failure (a known consistent state), this progress needs to occur exactly as in the original execution. A particular piece of execution is a sequence of process states and events. An event corresponds with a computational or communication step of a process that, given a preceding state, leads the process to a new state. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes; however, events are partially ordered by the Lamport "happened before" relationship [31]. Events can be deterministic or non-deterministic, depending on whether or not, from a given state, the same outcome state would always be obtained. Deterministic events follow the code flow (e.g., message emission or computations), while non-deterministic events, such as message receptions, depend on the time constraints of message deliveries. Processes are considered "piecewise deterministic:" only sparse non-deterministic events occur, separating large parts of deterministic computation. To progress the failed processes from the recovery line to the failure line and reach the same state, all events in that part of the execution need to be replayed in the exact same order as the initial execution. Deterministic events will be naturally replayed as the process executes the application code. However, the same outcome must be ensured for non-deterministic events, and thus they must be logged in the original execution. In addition, failed processes replay any message reception that impacted their state in the original execution, and thus the content of the messages needs to be available without restarting the sender process. For these purposes, the proposal applies a message logging protocol, detailed in Section 5, that keeps a copy of messages payload and logs the non-deterministic events.

As illustrated in Fig. 2, some communications need to be replayed during the consistency recovery phase: those to be received by a failed process to enable its progress (messages m_2 and m_4 in the figure) and those that were interrupted by the failure (message m_5). Other communications need to be skipped during the recovery: those that were successfully received by a survivor process (messages m_1 and m_3 in the figure). The success of the recovery is predicated on correctly identifying the communications belonging to each subset. Section 7 explains how this identification is performed and how the replay process takes place.

In addition, the reconstruction of communicators has an important implication for replaying communication messages interrupted by failures: all communications initiated but not completed before the failure are lost. We assume this includes the communication call in which the failure is detected, because – as stated in the ULFM specification – there are no guarantees regarding the outcome of that call. The management of communications interrupted by failure(s) is explained in Section 6.

5. Message logging

This section describes the message logging protocol that combines system-level logging and application-level logging. Point-to-point communications are logged using the Open MPI VProtocol component. Collective communications are logged by CPPC, at the API level, at the application level. The spatial coordination protocol used by CPPC contributes to a reduction of the log size by enabling processes to identify when a log will never be used for future replays.

5.1. *Logging point-to-point communications*

Point-to-point communications are logged using a pessimistic, sender-based message logging, which saves each outgoing message in the senders' volatile memory. Sender-based logging enables copying the messages, in parallel, while the network transfers the data. Pessimistic event logging ensures that all previous non-deterministic events of a process are logged before a process is allowed to impact the rest of the system. In MPI, non-deterministic events to be logged correspond to any-source receptions and non-deterministic deliveries (i.e., `iProbe`, `WaitSome`, `WaitAny`, `Test`, `TestAll`, `TestSome`, and `TestAny` functions). Because MPI communication channels are first in, first out (FIFO), replaying message emissions in order – and guaranteeing the same outcome for any-source receptions and non-deterministic deliveries – will lead to a consistent global execution state.

The method proposed here uses the VProtocol [21] message logging component, which provides sender-based message logging and pessimist event logging. The sender-based logging is integrated into the data-type engine of Open MPI and copies the data in a `mmap`-ed memory segment as it is packed [32] and, thus, moves the memory copies out of the critical path of the application. While log of the content of the messages is kept in the memory of the sender processes, for event logging, the outcome of non-deterministic events is stored on a stable remote server.

After a failure, communications must be replayed using the appropriate communicator. Because our approach is a hybrid between application-level and library-level recovery, failures cannot be masked completely at the application level, as is customary in pure, system-level message logging. Instead, the communication capability is restored at the application level. The ULFM communicator reconstruction implies new MPI communicators, and – in our work – the VProtocol has been extended to use translation tables between the old and new communicators, which are identified by their internal communicator ID (CID). The CID is included in the log, and – during the recovery – hash tables are built with the correspondence between old and new CIDs. This approach ensures the replay through the correct communicator and a consistent log in the event of additional failures.

5.2. *Logging collective communications*

The original VProtocol component, due to its location in the Open MPI software stack, sees only point-to-point communications. Instead of noticing a collective communication as such, it sees them unfolding as a set of point-to-point communications according to the collective algorithm implementation. It therefore logs collective operations by logging all of the corresponding point-to-point communications. This has two detrimental effects: (1) prevents operating with hardware-based or architecture-aware collective communications, and (2) results in an increased log volume that is not semantically necessary by storing identical messages at all intermediary processes along the overlay communication network used by the collective algorithm. To overcome these limitations, the method proposed here logs collective

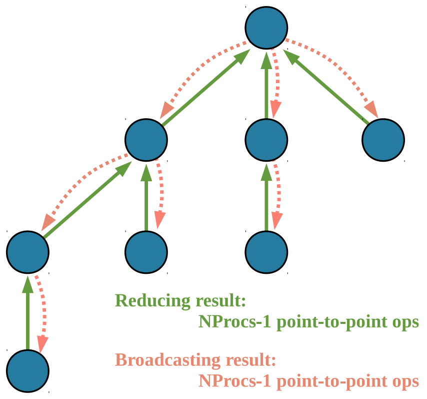


Fig. 3. Binomial-tree staging of `AllReduce` collective operation.

communications at the application level, thereby enabling the use of different collective communication implementations and potentially reducing the total log size, as the buffers sent in the intermediate steps of the collective are not logged. Conversely, when logging at the application level, the memory copies of the logged data are in the critical path of the application.

The benefits of this technique are tied to the implementation of the collective operations, and they will appear when intermediary copies are performed. For instance, Fig. 3 presents a common binomial-tree staging of the AllReduce collective operation. A first wave of point-to-point communications is performed to reduce the result, and a second one broadcasts this result to all processes. Therefore, the internal point-to-point implementation of this collective operation performs $2\times(\texttt{NProcs}-1)$ send operations, which implies logging $2 \times (\texttt{NProcs} - 1) \times \texttt{BuffSize}$ bytes of data. On the other hand, the application-level logging of the `AllReduce` collective operation logs the contribution of each process involved in the collective call, i.e., $\texttt{NProcs} \times \texttt{BuffSize}$ bytes of data. Therefore, both the number of entries that are appended to the log and the total logged data are divided by a $\frac{2\times(\texttt{NProcs}-1)}{\texttt{NProcs}}$ factor when logging at the application level.

An MPI wrapper implemented on top of CPPC performs the application-level logging. Instead of using the traditional MPI profiling API (PMPI), we decided to implement our own wrappers around MPI functions, leaving the PMPI layer available for other usages, and therefore allowing CPPC-transformed applications to benefit from any PMPI-enabled tools. The CPPC wrappers around MPI function calls perform the logging of the pertinent data and then calls the actual MPI routine. Each process logs the minimum data necessary to be able to re-execute the collective after a failure in a `mmap`-ed memory segment.

During the recovery, collective operations will be re-executed by all processes in the communicator, including survivors of the previous faults and replacement processes. Although survivor processes do not roll back, they will re-execute the collective communications during the recovery procedure, taking their inputs directly from their log. To ensure consistency, point-to-point and collective calls must be replayed in the same order as in the original execution. Thus, when logging a collective, VProtocol is notified, and it introduces a mark within its log. During the recovery, when a survivor process encounters a collective mark, the VProtocol component transfers control to CPPC to insert the collective re-execution call.

5.3. *Implications for the log size*

Traditional message logging, used in combination with uncoordinated checkpointing, treats all messages in the application as possible in-transit or orphan messages, which can be requested at any time by a failed process. In contrast, in the method proposed here – thanks to the spatially coordinated checkpoints provided by CPPC – the recovery lines cannot be crossed by any communication. Thus, only the messages from the last recovery line need to be available. Recovery lines, therefore, correspond with safe locations in which obsolete logs can be cleared, which means we can avoid keeping the entire log of the application or including it in the checkpoint files. However, with CPPC, processes checkpoint independently; the only way to ensure that a recovery line is completed would be to introduce a global acknowledgment between processes, which would add a synchronization point (and corresponding overhead) during the checkpoint phase. Instead, the logs from the l latest recovery lines are kept in the memory of the processes, l being a user-defined parameter. After a failure, the appropriate log will be chosen depending on which line is the selected recovery line. In the improbable case where an even older recovery line needs to be used, the log is not available. However, in this case, a global rollback is always possible. This approach reduces the overhead and the memory usage introduced by the message logging. Furthermore, in most application patterns, safe points are separated by semantically synchronizing collective communications that prevent a rollback going further than the last recovery line.

Note that communicator creation calls correspond with a particular type of collective operation. In many cases, derived communicators are created at the beginning of the application code, and they are used during the whole execution. Thus, these log entries are not cleared when checkpointing, as they will always be necessary for the failure recovery procedure to recreate the necessary communicators.

6. Reposting communications interrupted by a failure

Revoking and reconstructing communicators implies that all ongoing communications at the time of the failure are purged at all survivor processes. The incomplete communications correspond to the communication call in which the failure was detected and to all non-blocking communication calls that were not completed when the failure hit. A communication crossing the failure line (including between survivor processes) would then be lost and need to be reposted to ensure that the execution resumes successfully. Note that this implies not only replaying emissions (as in traditional system-level message logging) but also reposting, at the application level, those receptions and collective communications that were interrupted by the failure.

There are no guarantees regarding the outcome of the data transfer related to an MPI call in which a failure is detected (i.e., output buffers may contain garbage data). Therefore, to ensure consistency, the MPI calls mentioned above have to be re-executed. As commented earlier, all MPI calls in the application are performed through the MPI wrapper implemented on top of CPPC. Thus, within the MPI funâĂćtion wrappers, the code returned by the MPI call is checked for errors, and corrective actions are initiated when necessary. When an error is returned, the call is re-executed with its original arguments. Note, however, that some of those arguments are updated, such as the reconstructed communicators or – in the case of non-blocking communications – the requests that were reposted during the recovery replay. For non-blocking communication calls, as stated in the MPI standard, a non-blocking send start call (e.g., `MPI_Isend`) initiates the send operation but does not finish it. A separate completion call (e.g., `MPI_Wait`) is needed to finish the communication (i.e., to verify that the data has been copied out of the send buffer). Similarly, a non-blocking receive start call (e.g., `MPI_Irecv`) initiates the receive operation, but a separate completion call (e.g., `MPI_Wait`) is needed to finish the operation and to verify that the data has been received into the reception buffer.

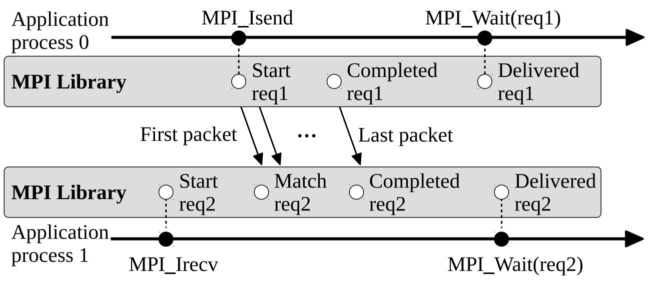


Fig. 4. States of non-blocking communications.

We consider the following states for a non-blocking communication request, illustrated in Fig. 4. When the completion call is invoked over a request, the request is considered delivered to the application. However, at any point between the initiation of the request to its delivery to the application, the request might be completed internally by CPPC; that is, the data has been received into the reception buffer even though the application has not yet acknowledged it.

When a failure strikes, there could be a number of pending non-blocking send and receive communications whose completion calls have not been invoked yet. Their requests, even when internally completed (i.e. the correct result is available in the output buffers), will not be delivered to the application after the recovery. The reason is that these requests will be associated with an old communicator, and a later invocation of a completion call will generate an error. On the other hand, those that did not complete internally will be lost upon failure, and they will need to be re-posted before the execution continues.

CPPC maintains a temporary log for non-blocking communication calls for both emissions and receptions, which is discarded upon delivery of a request. Note that a correctly designed MPI application will not modify the send buffers, nor will it use the receive buffers until a completion routine (e.g., `MPI_Wait`) has been invoked over the associated requests. Thus, this temporary log avoids doing memory copies of the send buffers and instead keeps a reference to them. For each non-blocking call, CPPC creates a log entry that permits the re-execution of the call to again initiate the send/receive operation and a reference to the associated request it generated. Due to the temporary characteristics of this log, a pool of log entries is used to avoid the overhead from allocating and freeing these items.

The consistency of the temporary non-blocking log is maintained as follows. First, when a request is internally completed by CPPC the associated log entry is removed. However, the request becomes a non-delivered request until the application layer is informed, and CPPC maintains a reference to it. Eventually, when a completion routine (e.g., `MPI_Wait`) is invoked and the request is delivered, the CPPC reference is finally removed. The management described here applies for all types of non-blocking requests and is purely local, we will detail in Section 7 the distributed management including the ordering of communication reposts.

After a failure, the first step consists of freeing the resources associated with the incomplete and non-delivered requests. The incomplete request will be reposted within the survivors' replay, as explained in Section 7. Note that when reposting a non-blocking communication call, its associated request in the application needs to be updated. This is solved using the same approach as is used with the communicators: a custom CPPC request is used by the application, which actually contains a pointer to the real MPI request, thereby enabling the MPI request to be updated transparently after it has been reposted.

7. Tracking messages and emission replay

In MPI, observing a communication completion at one peer does not imply that it has also completed at other peers. For example, an `MPI_Send` can complete as soon as the send buffer can be reused at the local process. Meanwhile, the message may be buffered, and the corresponding receive may still be pending. During the replay, survivors have to resend the message log to the restarting processes (as in traditional message logging), but they also have to send the message log to other survivor processes for which receptions have been interrupted by the failure (i.e., those receptions that have been reposted by the protocol described in Section 6). Said another way, the success of the replay relies on the receivers' capability to inform the senders which communications have been received and on the ability of the senders to distinguish which communications need to be replayed. Since this list of completed or incomplete communication depends on post order at the receiver, and not post order at the sender, the messages that need to be replayed are not necessarily contiguous in the sender-based log. Thus, it is critical that a given communication can be unequivocally identified by both peers involved.

Our strategy for identifying messages relies on sequence numbers. Every time a message is sent over a particular communicator to a particular receiver, a per-channel counter is increased, and its value is piggybacked in the message as the sender sequence ID (SSID). This SSID is then used to implement a tracking protocol for point-to-point communications to identify the messages that are expected by other peers and need to be replayed and to identify those that were completed and need to be skipped.

7.1. Tracking protocol

During the execution, processes track the SSIDs for each send and receive operation they have completed over each communication channel. For the emissions on a given channel, SSIDs grow sequentially. Thus, the sender only needs to keep the most recent SSID for that communication channel. However, receiving a message with a particular SSID does not ensure that all previous messages in that channel have been received, because a receiver can enforce a different reception order (e.g., by using different tags). Thus, processes maintain the highest SSID (h_{ssid}) they have received and a list of pending SSIDs ranges. Each range is represented with a pair $[a_{ssid}, b_{ssid}]$, meaning that messages with SSIDs in that range are pending. When, in a channel, a non-consecutive SSID is received:

- If it is larger than the highest SSID received, a new range $[h_{ssid} + 1, current_{ssid} - 1]$ is added to the pending reception list.
- Otherwise, it is a pending SSID, and it must be removed from the pending list. Note that the removal can imply splitting a pending range in two.

Each checkpoint file includes the latest sent and the highest-received SSIDs. Note that, when using CPPC, messages cannot cross the recovery line, and therefore there are no pending ranges when checkpointing.

SSIDs tracking is only used for point-to-point communication replay. In Open MPI, the header of the message already contains a sequence number for MPI point-to-point ordering. To avoid the extra cost of duplicate tracking and piggybacking, we reuse that existing sequence number in the SSID tracking algorithm. To prevent the collective communications – implemented using point-to-point communications – from impacting the SSID tracking, they are run through a different communicator. Additionally, when communicators are reconstructed with ULFM, Open MPI SSIDs are reset. The tracking protocol deals with this issue by calculating the SSID offsets. During the recovery, the value of the SSID before the failure is restored, and the SSID tracking continues using the saved value as a baseline and then adding the current value of the Open MPI sequence number. This absolute indexing of SSIDs allows for tolerating future failures after the first recovery and, notably, failures hitting the same recovery line multiple times.

7.2. Ordered replay

Once the failed processes are recovered using the checkpoint files, all processes exchange the tracking information. For each pair of processes that have exchanged messages in the past, and for each particular communication channel they have used, the receiver notifies the sender of the highest SSID it has received and of all pending ranges. Using this remote information, if the sender is:

- A failed process: it knows which emissions can be skipped because they were successfully received during the previous execution, and which ones must be reemitted.
- A survivor process: it can determine which messages from its log must be replayed because other peers require them.

Then, failed processes continue their execution, skipping the communications already received by survivors, and emitting those that the peers expect to receive. Additionally, the information from the event logger is used by the failed processes to ensure that non-deterministic events are replayed exactly as they were in the original execution (e.g., an any-source and/or any-tag reception will be regenerated as a named reception, with a well specified source and tag to prevent any potential communication mismatch and deliver a deterministic re-execution in which the data is received in exactly the same order as in the original execution). All collective communications are normally executed; the collective protocol, detailed in Section 5.2, guarantees both correctness and native collective performance.

Meanwhile, survivors start the replay of logged communications, and invoke VProtocol's replay—replaying the necessary point-to-point communications from the log. When a survivor encounters a collective log mark in its log, it transfers control to CPPC to re-execute the appropriate collective. Towards the end of the log, the survivor can encounter gaps that originated from non-blocking emissions that were interrupted by a failure, which – again – results in control transfers to CPPC to re-execute the pending emissions. As mentioned in Section 6, there can also be pending receptions interrupted by a failure that need to be re-posted. However, due to the sender-based logging protocol, there are no marks for non-blocking receptions interrupted by the failure in VProtocol's log (the only logged information for receptions relates to the ordering on non-deterministic events to the remote event logging server, when applicable). Pending receptions that were interrupted (i.e. produced an error code) are tracked by the CPPC component and are re-posted and ordered with respect to the collective communications and pending non-blocking emissions, thereby ensuring consistency. Once a survivor finishes processing its log, it continues the execution normally.

8. Experimental evaluation

The experimental evaluation was performed at the Galicia Supercomputing Center using the "FinisTerrae-II" supercomputer. Each of FinisTerrae-II's nodes is composed of two Intel Haswell E5-2680 v3 CPUs running at 2.50 GHz, with 12 cores per processor (24 cores per node), and 128 GB of RAM. The nodes are connected using an InfiniBand FDR 56 Gbps interconnect using a Fat-tree topology. The experiments spawned 24 MPI process per node (one

Table 1
Original run times of the testbed applications in minutes.

	48P	96P	192P	384P	768P
Himeno	18.48	9.22	4.76	2.45	1.56
Mocfe	20.49	8.26	4.75	2.41	1.48
SPhot	16.38	8.25	3.78	2.25	1.48
Tealeaf	17.50	9.31	4.28	2.35	1.79

per core). Checkpoint files are dumped on a remote parallel file system using Lustre over InfiniBand with a theoretical I/O bandwidth of 20 GB/s. For our testing, we used CPPC version 0.8.1, working with HDF5 version 1.8.11 and GCC version 4.4.7. The Open MPI version used corresponds with ULFM 1.1 and was modified for the integration of VProtocol and CPPC. The Portable Hardware Locality (hwloc) [33] package was used for binding the processes to the cores. Applications were compiled with optimization level O3. We report the average times of 20 executions.

We used an application testbed comprised of four domain science MPI applications with different checkpoint file sizes and communication patterns. We ran SPhot [34], a physics package that is part of Lawrence Livermore National Laboratory's Advanced Simulation and Computing (ASC) Sequoia Benchmarks assortment, at `NRUNS`=48 000. We ran the Himeno benchmark [35], a Poisson equation solver, fixing NN to 12,000 with a grid size of 1024 $\times$ 512 $\times$ 512. We also used MOCFE-Bone [36], which simulates the main procedure in a 3-D method of characteristics (MOC) code, using 4 energy groups, 8 angles, a mesh of 57^3 doing strong scaling in space, and a trajectory spacing of 0.01 cm^2. TeaLeaf [37] is a mini-app, originally part of the Mantevo project, that solves a linear heat conduction equation on a spatially decomposed regular grid using a five-point stencil with implicit solvers. We ran it with `x_cells` and `y_cells` set to 4,096 at 100 time steps. The original execution times of the applications, in minutes, are reported in Table 1. Experiments were executed doing strong scaling (i.e., maintaining the global problem size constant as the application scales out).

As reported in the next section, the settings for the experiments (i.e., checkpointing and failure frequencies) establish homogeneous parameters across the different tests, simplifying a thorough study of the performance of the local rollback protocol. In realistic scenarios, applications run times would be in the order of days, thus, multiple failures would hit the execution. Therefore, checkpoints would need to be taken to ensure the execution completion. In this scenario, the local rollback would provide more efficient recoveries each time a failure strikes, thus, improving the overall execution time under those conditions. Even though a logging overhead is introduced during the execution, the fact that the log can be cleared upon checkpointing provides an important advantage over traditional message logging techniques. More precisely, being able to clean the log on checkpoints is a critical property of our solution as it allows it to behave nicely on communication-bound applications that otherwise would have exceed the memory limitations of the running environment.

8.1. Performance evaluation of the local rollback protocol

In order to measure the benefits of the proposed solution in recovery scenarios, the experiments compare the local rollback with an equivalent global rollback strategy. In both cases, the checkpointing frequency (N) is fixed so that two checkpoint files are generated during the execution of the applications—the first one at 40% of the execution progress and the second one at 80%. The N value for each application is shown in Table 2. In all experiments, a failure is introduced by killing the last rank in the application when 75% of the computation is completed. Once the failure is detected, communicators are revoked, survivors agree about the failed process, and a replacement process is spawned. In the global rollback, all processes roll back to the checkpoint generated at 40% of the execution. In the local rollback, the replacement processes continue from the last checkpoint; while, the survivors use their logs to provide the restarting process with messages that enable the progress of the failed process until it reaches a consistent state. In the global rollback, no extra overhead besides CPPC instrumentation and checkpointing is introduced. On the other hand, the local rollback proposal introduces extra operations to maintain the logs. In these experiments, only the logs from the last recovery line are kept in memory ($l = 1$). Below, Section 8.1.1 describes the extra fault-free overhead, and Section 8.1.2 studies the benefits (upon recovery) of the local rollback proposal.

8.1.1. Logging overhead

The overhead introduced by the local rollback is influenced by the communication pattern of the applications and how the logs evolve during the execution. Table 2 characterizes the applications in terms of their log volume, and also in terms of the MPI routines that are called in the main loop of the application (where the checkpoint call is located). The table also shows the number of processes running the experiment, the total number of iterations run by the application, and the checkpointing frequency (N) indicating the number of iterations between two consecutive checkpoints. Regarding event logging, only SPhot generates non-deterministic events—precisely ($num\_procs - 1$) $\times$ 5 per iteration. Finally, the aggregated average log behavior per iteration is reported, that is, the average among all iterations, where the aggregated value for each iteration is computed as the addition of the log from all processes (i.e. the total log generated by the application in one iteration). The number of entries and the size of the log generated by the proposal are reported for both point-to-point and collective calls. For the latter, the table also shows the reduction, in percentage, that the application-level collective logging provides over the internal point-to-point logging.

In the target applications, one can see a very significant reduction in the collective communications log volume thanks to application-level logging of collective operations. This effect is significant, even though the log size of the testbed applications is dominated by point-to-point operations, other scenarios may present a larger contribution from collective communications. The results from a profiling study performed by Rabenseifner [38] show that nearly every MPI production job uses collective operations and that they consume almost 60% of the MPI time, with `AllReduce` the most called collective operation. In traditional HPC MPI applications, `AllReduce` operations frequently work with relatively small message sizes, however, emerging disciplines, such as deep learning, usually rely on medium and large messages sizes [39]. To illustrate the performance effect of this technique with different message sizes, we refine the analysis with the behavior of relevant collective communications in the Intel MPI Benchmarks [40] (IMB). Fig. 5 compares the two logging methods on 48 processes (24 per node), showing – for different message sizes – the overhead induced over a non-fault-tolerant deployment (100% means that the logging method imparts no overhead). The volume of log data and the number of log entries are reported for the collective operations `Allgather`, `AllReduce`, and `Bcast`. For these collective operations, application logging shows notable reductions in the size and number of entries in the log, which in turn translates to a notable reduction in collective communication latency. Note that sudden changes in the logged data and the number of entries in the log correspond with Open MPI choosing a different implementation of collective communications depending on message size. In the general case, the reduction in the logging overhead when logging collective communications at the application level depends on the operation's semantic requirements and on the communication pattern in the point-to-point implementation of the collective

Table 2
Benchmarks characterization by MPI calls and log behavior.

	MPI calls in main loop	#Procs	#Iters	N	Aggregated average log behavior per iteration			
					Point-to-Point		Collective Communications	
					Entries	Size	Entries (%↓)	Size (%↓)
Himeno	`MPI_Irecv`, `MPI_Isend`, `MPI_Wait`, `MPI_AllReduce`	48	12k	5k	208	28.5 MB	48(↓75.00%)	1.5 kB(↓81.82%)
		96	12k	5k	448	34.8 MB	96(↓78.57%)	3.0 kB(↓84.42%)
		192	12k	5k	944	51.5 MB	192(↓81.25%)	6.0 kB(↓86.36%)
		384	12k	5k	2.0k	68.7 MB	384(↓83.33%)	12.0 kB(↓87.88%)
		768	12k	5k	4.1k	82.2 MB	768(↓85.00%)	24.0 kB(↓89.09%)
Mocfe	`MPI_Irecv`, `MPI_Isend`, `MPI_Waitall`, `MPI_Allreduce`, `MPI_Reduce`	48	10	4	69.9k	1.7 GB	2.0k(↓75.00%)	140.4 kB(↓78.53%)
		96	10	4	150.5k	2.3 GB	3.9k(↓78.57%)	280.9 kB(↓81.59%)
		192	10	4	317.2k	3.0 GB	7.9k(↓81.25%)	561.8 kB(↓83.89%)
		384	10	4	666.6K	3.9 GB	15.7k(↓83.33%)	1.1 MB(↓85.68%)
		768	10	4	1376.3k	5.0 GB	31.5k(↓85.00%)	2.2 MB(↓87.12%)
SPhot	`MPI_Irecv`, `MPI_Send`, `MPI_Waitall`, `MPI_Barrier`	48	1k	400	235	29.3 kB	192(↓83.33%)	1.5 kB(↓96.67%)
		96	500	200	475	59.2 kB	384(↓85.71%)	3.0 kB(↓97.14%)
		192	250	100	955	118.8 kB	768(↓87.50%)	6.0 kB(↓97.50%)
		384	125	50	1.9k	238.0 kB	1.5k(↓88.89%)	12.0 kB(↓97.78%)
		768	62	25	3.8k	476.5 kB	3.1k(↓90.00%)	24.0 kB(↓98.00%)
Tealeaf	`MPI_Irecv`, `MPI_Isend`, `MPI_Waitall`, `MPI_AllReduce`	48	100	40	184.6k	1.7 GB	107.9k(↓75.00%)	3.7 MB(↓81.25%)
		96	100	40	387.4k	2.5 GB	216.0k(↓78.57%)	7.4 MB(↓83.93%)
		192	100	40	800.7k	3.7 GB	431.3k(↓81.25%)	14.8 MB(↓85.94%)
		384	100	40	1638.6k	5.4 GB	863.3k(↓83.33%)	29.6 MB(↓87.50%)
		768	100	40	3331.1k	7.8 GB	1726.6k(↓85.00%)	59.3 MB(↓88.75%)

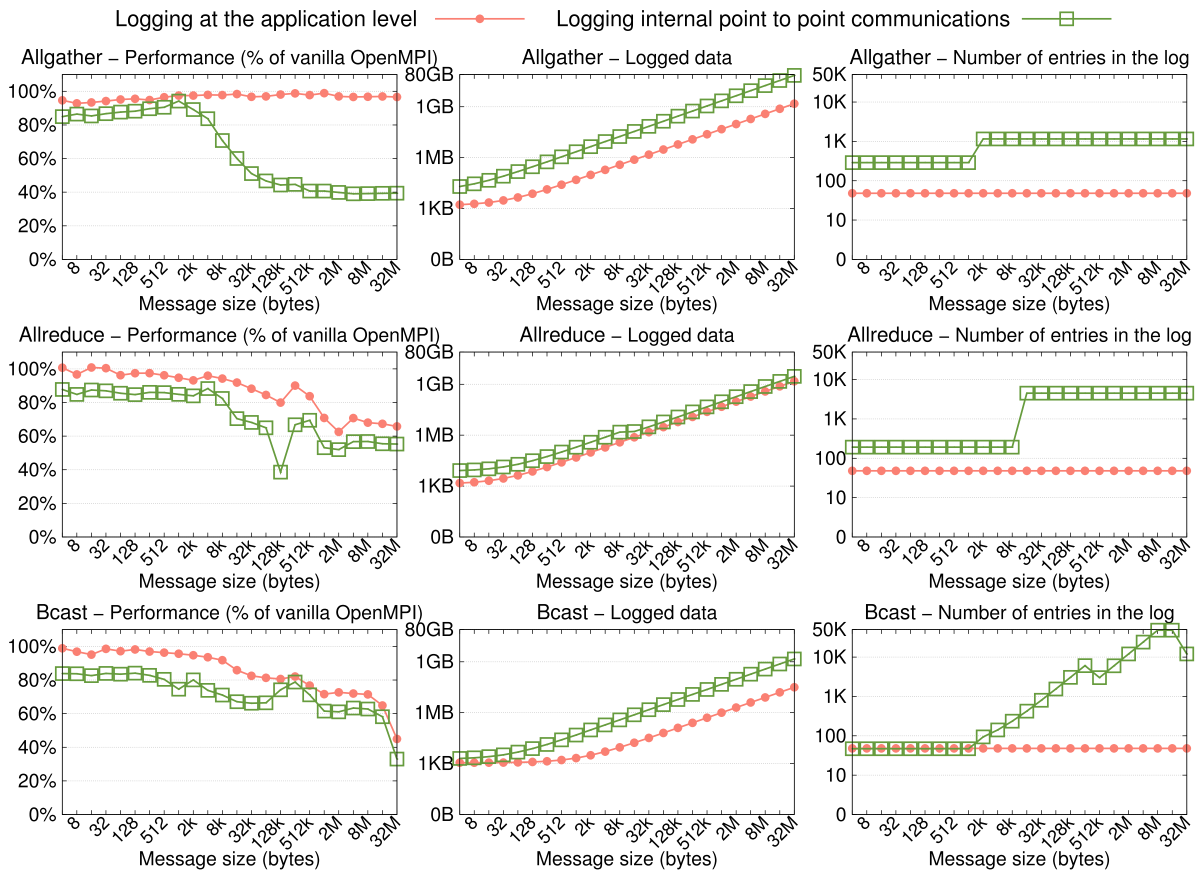


Fig. 5. Logging of collective communications: application level vs. internal point-to-point logging.

communication. In the collective operations not presented here, application logging yields minor advantages over point-to-point logging in terms of log volume. This, in turn, translates to smaller performance differences between the two approaches. Note that, in any case, point-to-point logging is not compatible with the use of hardware-accelerated collective communication, which is expected to impart a significant overhead from which application logging is immune, independent of the log volume.

For the applications, Fig. 6 shows the memory consumption overhead introduced by the log. To provide a better overview of the impact of this memory consumption, the maximum total size of the log has been normalized with respect to the available memory (number of nodes × 128 GB per node). Using CPPC and its spatial coordination protocol allows the log to be cleared upon checkpointing; thus, the maximum log size corresponds to the product of the number of iterations between checkpoints (N) and

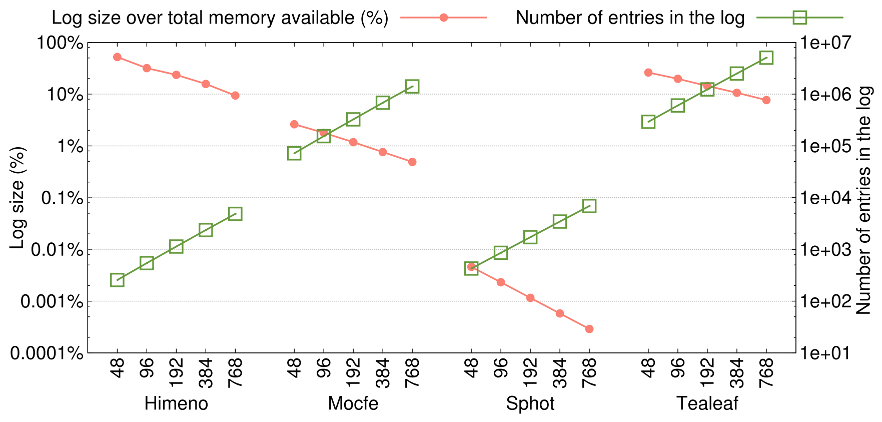


Fig. 6. Log parameters when checkpointing: maximum log's sizes expressed as the percentage of the total memory available and number of entries during the execution.

the addition of both point-to-point and collective communications log sizes per iteration (values shown in Table 2). Equivalently, the maximum number of entries in the log corresponds to the product of N and the total number of entries in the log. Fig. 6 also presents this maximum. Both the number of entries in the log and the log size are represented in log scale. In all applications tested here, we see that the number of log entries increases as more processes run the applications, while the percentage of the total memory occupied by the log decreases. The Himeno application has the largest log size, ranging from 52% of the available memory when using 48 processes to 9% of the available memory when using 768 processes. Tealeaf has the second largest log, with a maximum log size ranging from 26.2% to 7.7% of the available memory when scaling up the application. In these applications, the bulk of the communication employs point-to-point calls, and the collective operations do not account for a significant portion of the logged data.

In fault-free executions, the local rollback protocol also introduces overhead in the form of communication latency and checkpoint volume. Fig. 7 compares the overhead of global rollback and local rollback resilience in the absence of failures. It reports the absolute overhead in seconds, with respect to the original runtime (shown in Table 1), and reports the aggregated checkpoint file size (i.e., the total checkpoint volume from all processes). First, the amount of log management data that needs to be added to the checkpoint data is negligible. Therefore, checkpoint file sizes do not present a relevant difference (i.e., advantage or disadvantage) between the local and global rollback solutions. Second, the overhead introduced by the local rollback is close to the overhead introduced by the global rollback solution. Most of the logging latency overhead is hidden, and its contribution to the total overhead of the local rollback approach is small compared to the contribution of the checkpoint cost. The overhead grows with the size of the checkpoint file. The SPhot and Tealeaf applications present the smallest checkpoint file sizes, with SPhot having checkpoint file sizes of 46–742 MB and TeaLeaf having checkpoint file sizes of 261–302 MB, with the upper ranges representing an increased number of processes. Given the very small overhead imparted by checkpointing on Tealeaf, for some experiments the overhead for the global rollback for Tealeaf is slightly negative (less than 0.5% of the original runtime), presumably because of the optimizations applied by the compiler when the code is instrumented with CPPC routines. The overhead of the logging latency, while minimal in the overall runtime of the application, comes to dominate the cost of checkpointing in the failure-free overhead breakdown.

8.1.2. Recovery benefits

The local rollback solution reduces the time required for recovering the applications when a failure occurs. The application is considered to have fully recovered when all processes (failed and survivors) have reached the execution step at which the failure originally interrupted the computation. In both the global and local recovery approaches, for a failed process, this point is attained when it has finished re-executing the lost work. A survivor process is considered fully recovered once it has either re-executed all lost work in the global recovery scheme, or when it has served all necessary parts of the log to the failed (restarting) processes, and to other survivors that require it.

Fig. 8 presents the reduction percentage of the local rollback recovery time over the global rollback recovery for both the survivor and failed processes. The improvement in the recovery times is very similar for all processes: these applications perform collective operations in the main loop, and all survivor processes that originally participated in the collective communications are also involved in their replay during recovery.

Fig. 9 shows the times (in seconds) of the different operations performed during the recovery. The ULFM recovery times include the failure detection, the re-spawning of the replacement processes and the entire reconstruction of the MPI environment, including communicators' revocation, shrinking and reconfiguration. The CPPC reading times measure the time spent during the negotiation of all processes about the recovery line to be used, and the reading of the selected checkpoint files by the failed ones. The CPPC positioning times include the time to recover the application's state of the rolled back processes, including the reconstruction of the application data (moving it to the proper memory location, i.e., the application variables), and the re-execution of non-portable recovery blocks (such as the creation of communicators). The CPPC positioning finishes when the failed processes reach the checkpoint call in which the checkpoint file was originally generated. Finally, we included the failed processes' re-computation time which corresponds with the time spent from the checkpoint call in which the recovery files where generated until the failed processes have reached the execution step at which the failure originally interrupted the computation. The results in Fig. 9 are summarized in Fig. 10, which represents the percentage of the reduction in the recovery times that is due to each recovery operation in the failed processes.

Both approaches, local and global, perform the same ULFM operations during the recovery, thus, no relevant differences arise. In the general case, the CPPC reading and positioning operations benefit from the local rollback strategy, as the number of processes reading checkpoint files and moving data in memory decreases. As can be observed in both figures, the re-computation of failed processes is the recovery operation with the largest weight in the reduction of the failed processes' recovery times. This happens because the failed processes perform a more efficient execution of the computation: no communications waits are introduced, received

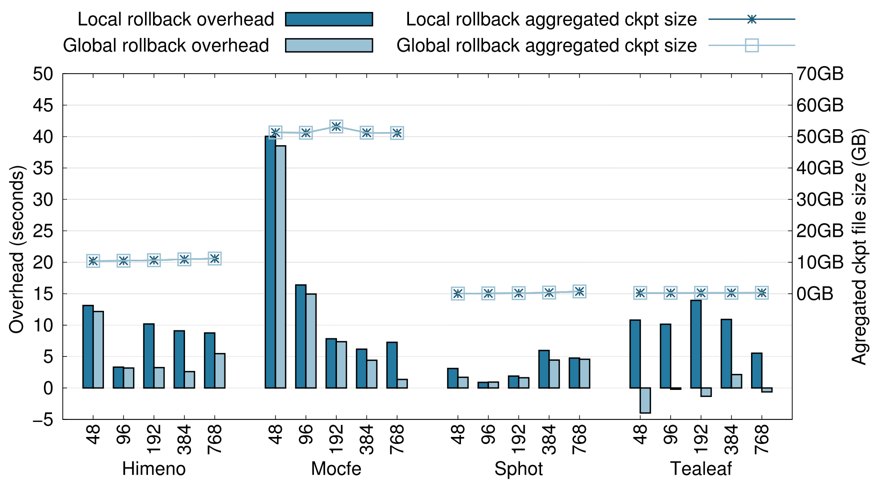


Fig. 7. Checkpoint file sizes and absolute overhead in the absence of failures with respect to the non fault-tolerant version.

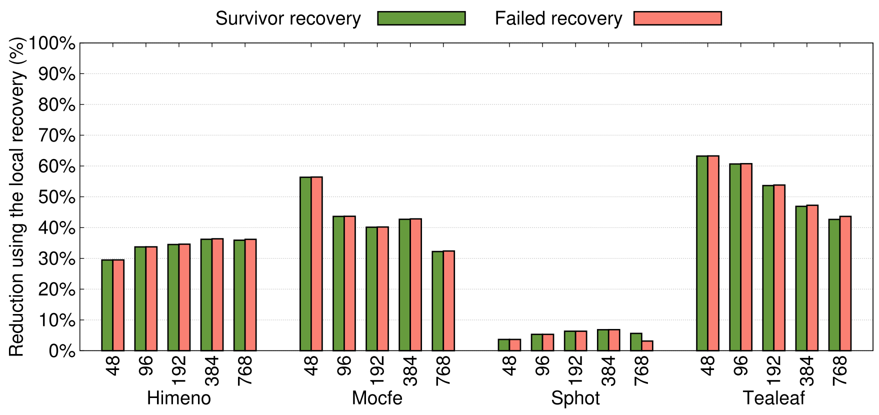


Fig. 8. Reduction of the recovery times of survivor and failed processes with the local rollback (the higher, the better).

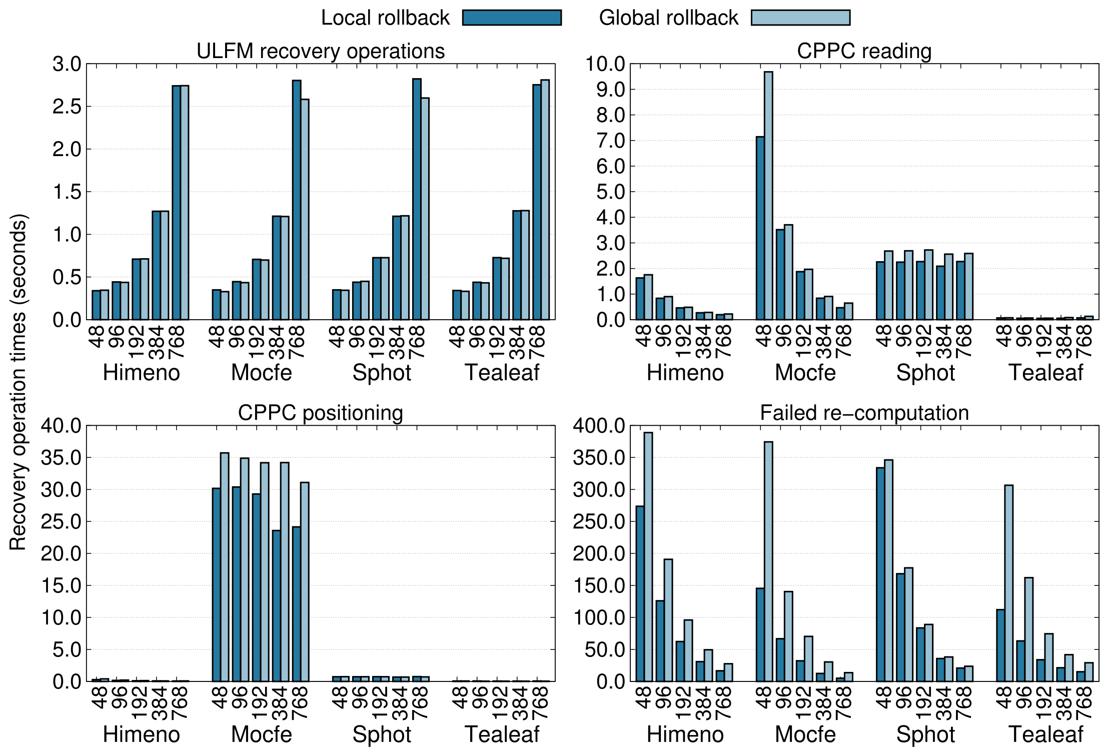


Fig. 9. Times (in seconds) of the different operations performed during the recovery.

messages are rapidly available, and unnecessary message emissions from past states of the computation are skipped (although failed processes log them). Note that, in the testbed applications, the collective operations synchronize the failed and survivor processes recovery, thus, the time spent by the survivors in the replay of communications is almost the same as the failed processes' recomputation times. In applications that do not present collective communications in the main loop, and where the applications are less tightly coupled, i.e the communications patterns between survivor and failed processes are less synchronizing, a larger improvement is expected.

The reduction in recovery time has a direct impact on the improvement of the overall execution time when a failure occurs. Fig. 11 shows the reduction in run time and energy consumption achieved when using local rollback instead of global rollback when faced with a failure. The extra runtime and the energy consumption are calculated as the difference between the original runtime/energy use and the runtime/energy use when a failure occurs. Energy consumption data is obtained using the `sacct` Slurm command. As an example, in the 48 processes execution, the overhead added (upon failure) to MOCFE's run time is reduced by 53.09% when using the local rollback instead of the global rollback, and the extra energy consumed by failure management is reduced by 74%.

The SPhot application shows the lowest performance benefit. In this application, all point-to-point communications consist of *sends* from non-root processes to the root process. Thus, during the recovery, survivor processes do not replay any point-to-point communications to the failed rank, although they do participate in the re-execution of the collective communications. For this reason, the performance benefit for SPhot during the recovery is mainly due to the execution of the failed process computation without synchronization with other ranks. The *receives* are served from the message log, and most of its emissions have already been delivered at the root process and are avoided. Even though these messages are not sent, they are still being logged to enable the recovery from future failures. In the other applications, where the communication pattern is more favorable, the local recovery permits a significant reduction in the extra time when compared to the global restart strategy.

8.2. Weak scaling experiments

In addition to the previous experiments, this section reports the results when doing weak scaling on the Himeno application, i.e, maintaining the problem size by process constant as the application scales out. These experiments compare the local rollback and global rollback under the same conditions as in the previous experiments (checkpointing frequency, failure introduction, etc.). Table 3 reports the configuration parameters and the original run times of the application, in minutes.

Table 4 (equivalent to Table 2 in the strong scaling experiments) characterizes these tests in terms of their log volume, reporting the MPI routines that are called in the main loop of the application (where the checkpoint call is located), the number of processes running the experiment, the total number of iterations run by the application, and the checkpointing frequency (N) indicating the number of iterations between two consecutive checkpoints. Table 4 also presents the aggregated average log behavior per iteration for the point-to-point and collective calls in terms of number of entries and size of the log. In comparison with the strong scaling experiments, there are no changes in the number of collective communications performed during the execution. On the other hand, even though the same number of point-to-point communications are performed in one iteration, the data transmitted presents a more abrupt increase when scaling out. Fig. 12a shows, in log scale, the memory consumption overhead introduced by the log (equivalent to Fig. 6 in the strong scaling experiments). As can be observed, in these experiments the increase in the data transmitted by the point-to-point communications results in a more steady maximum percentage of the total memory that is occupied by the log (between 31%–42% of the available memory) when scaling out.

Fig. 12b reports the absolute overheads introduce by the local and global rollback in the absence of failures, and aggregated checkpoint file size (equivalent to Fig. 7). The overhead introduced by both proposals is low, and the aggregated checkpoint file size increases when scaling out, as the contribution from each process remains constant.

Fig. 12c reports the reduction in the failed and survivors processes' recovery times, while Fig. 12d shows the reduction in run time and energy consumption achieved when using local rollback instead of global rollback when faced with a failure (equivalent to Figs. 8 and 11). On average, the reduction on the total run time is 42.9%, while the reduction in the energy consumption corresponds with a 49.7% on average. As in Himeno strong scaling experiments, most of the improvement is due to a more efficient execution of the failed processes.

9. Concluding remarks

This work proposes a novel local rollback solution for SPMD MPI applications that combines several methods to provide efficient resilience support to applications. ULFM fault mitigation constructs, together with a compiler-driven application-level checkpointing tool, CPPC, and supported by the message logging capabilities of the Open MPI library-level VProtocol pessimist message logging, are combined to significantly reduce the resilience overhead of checkpoint and recovery. The ULFM resilience features are used to detect failures of one or several processes, maintain communication capabilities among survivor processes, re-spawn replacement processes for the failed processes, and reconstruct the communicators. Failed processes are recovered from the last checkpoint, while global consistency and further progress of the computation is enabled by means of message logging capabilities. Fine tracking of message sequences and partial message completion after failures permit the deployment of message logging over ULFM, and an original two-level logging protocol permits alternating the recovery level from library-level message logging to application-directed, semantic-aware replay. Collective communications are logged at the application level, thereby reducing the log size and enabling the use with architecture-aware collective communications even after faults. The resultant local rollback protocol avoids the unnecessary re-execution overheads and energy consumption introduced by a global rollback, as survivor processes do not roll back to repeat computation already done; yet, the spatially coordinated protocol used by CPPC helps reduce the volume of stall logs carried from past checkpoints. This combination of protocols proves singularly symbiotic in alleviating the shortcomings of each individual strategy.

The experimental evaluation was carried out using four real MPI applications with different checkpoint file sizes and communication patterns. The performance of the local rollback protocol has been compared with an equivalent global rollback solution. While in a failure-free execution, the required operations to maintain the logs imply a small increase in the overhead compared to the global rollback solution, in the presence of failures, the recovery times of both failed and survivor processes are noticeably improved. This improvement translates to a considerable reduction in the additional execution time and energy consumption introduced by a failure when using local rollback instead of global rollback.

To further reduce the cost of tolerating failures, future work will study optimizations to perform a more efficient replay of the

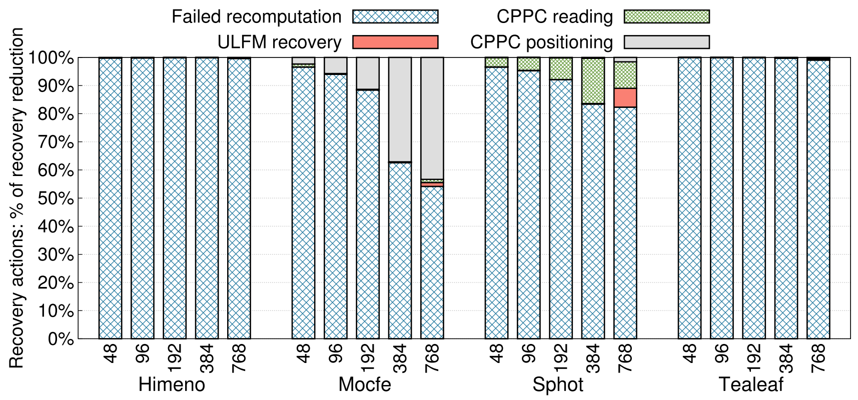


Fig. 10. Percentage that each recovery operation represents over the reduction in the failed processes' recovery times.

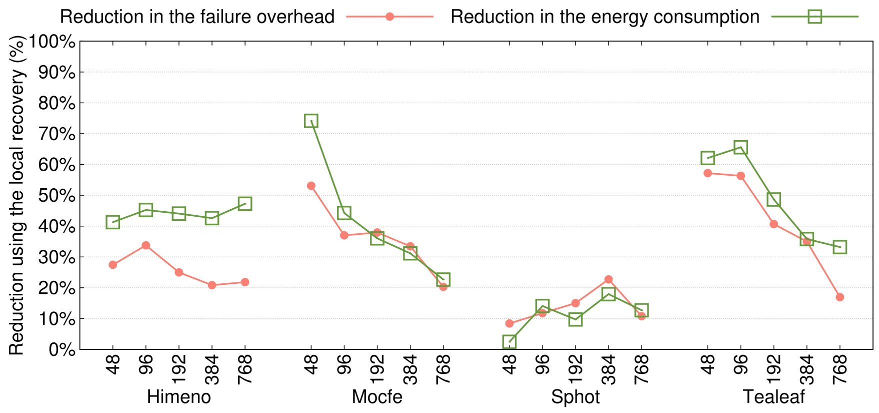


Fig. 11. Reduction in the extra run time and energy consumption when introducing a failure and using local rollback instead of global rollback (higher is better).

Table 3
Original run times (in minutes) and configuration parameters of the weak scaling experiments.

Himeno — Weak Scaling		
	Configuration parameters	Run time (min)
48 processes	Gridsize: 512 × 512 × 512, NN = 12 000	9.16
96 processes	Gridsize: 1024 × 512 × 512, NN = 12 000	9.14
192 processes	Gridsize: 1024 × 1024 × 512, NN = 12 000	9.72
384 processes	Gridsize: 1024 × 1024 × 1024, NN = 12 000	10.11
768 processes	Gridsize: 2048 × 1024 × 1024, NN = 12 000	10.24

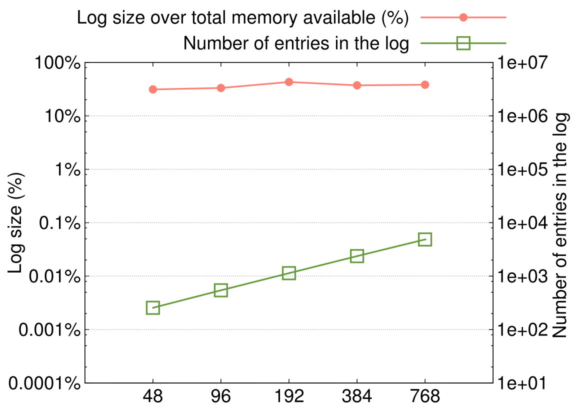
Table 4
Benchmarks characterization by MPI calls and log behavior.

	MPI calls in main loop	#Procs	#Iters	N	Aggregated average log behavior per iteration			
					Point-to-Point		Collective communications	
					Entries	Size	Entries (%↓)	Size (%↓)
Himeno	`MPI_Irecv`, `MPI_Isend`, `MPI_Wait`, `MPI_AllReduce`	48	12k	5k	208	16.3 MB	48(↓75.00%)	1.5 kB(↓81.82%)
		96	12k	5k	448	34.8 MB	96(↓78.57%)	3.0 kB(↓84.42%)
		192	12k	5k	944	90.0 MB	192(↓81.25%)	6.0 kB(↓86.36%)
		384	12k	5k	2.0k	155.8 MB	384(↓83.33%)	12.0 kB(↓87.88%)
		768	12k	5k	4.1k	320.1 MB	768(↓85.00%)	24.0 kB(↓89.09%)

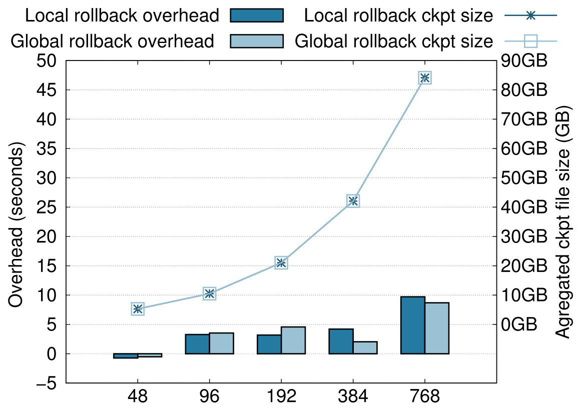
communications needed for the progress of the failed processes. The proposed strategy replays the collective operations identically (i.e., all processes involved in the original execution are also involved in the replay). A custom replay of collective communications (e.g., replacing the collective operation by one or a set of point-to-point operations) has the potential to further improve the performance of the recovery and reduce its synchronicity. To improve upon the cost of point-to-point communications, using MPI remote memory access operations can also enable us to obtain the message log from survivor processes without their active involvement in the recovery procedure.

Acknowledgments

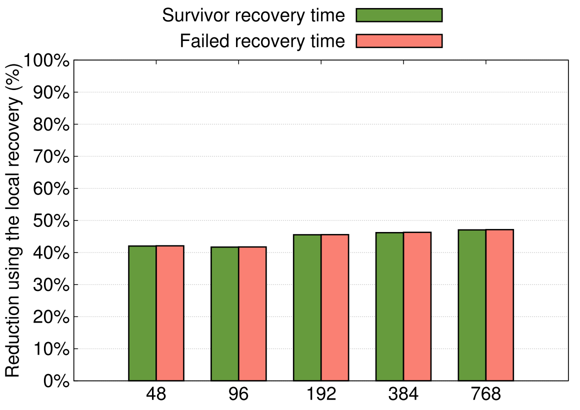
This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Projects TIN2016-75845-P and the predoctoral grants of Nuria Losada ref. BES-2014-068066 and ref. EEBB-I-17-12005); by EU under the COST Program Action IC1305 Network for Sustainable Ultrascale Computing (NESUS) and a HiPEAC Collaboration Grant and by the Galician Government (Xunta de Galicia) under the Consolidation



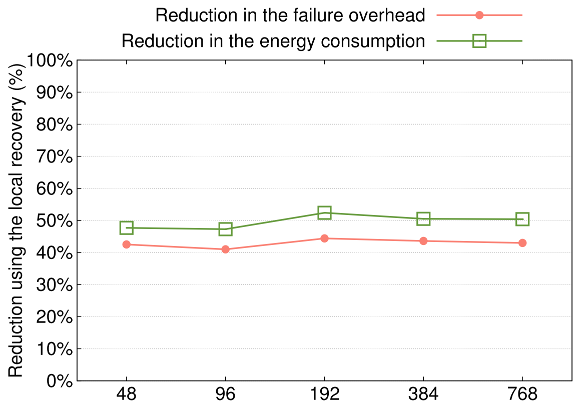
(a) Log parameters when checkpointing: maximum log's sizes expressed as the percentage of the total memory available and number of entries during the execution.



(b) Checkpoint file sizes and absolute overhead in the absence of failures with respect to the non fault-tolerant version.



(c) Reduction of the recovery times of survivor and failed processes with respect to the global rollback recovery.



(d) Reduction of the overall failure overhad and energy consumption with the local rollback with respect to the global rollback recovery.

Fig. 12. Results for the Himeno benchmark doing weak scaling (keeping the problem size by process constant).

Program of Competitive Research (ref. ED431C 2017/04). We gratefully thank Galicia Supercomputing Center for providing access to the FinisTerrae-II supercomputer.

This material is also based upon work supported by the US National Science Foundation, Office of Advanced Cyberinfrastructure, under Grants No. #1664142 and #1339763.

Authors would like to thank Thomas Hérault for the interesting discussions during the development of this work.

References

[1] C. Di Martino, Z. Kalbarczyk, R. Iyer, Measuring the resiliency of extreme-scale computing environments, in: Principles of Performance and Reliability Modeling and Evaluation, Springer, 2016, pp. 609–655.

[2] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra, Post-failure recovery of mpi communication capability: design and rationale, Int. J. High Perform. Comput. Appl. 27 (3) (2013) 244–254.

[3] G. Rodríguez, M.J. Martín, P. González, J. Tourino, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 749–766.

[4] G. Rodrıguez, M.J. Martın, P. González, J. Tourino, A heuristic approach for the automatic insertion of checkpoints in message-passing codes, J. Universal Comput. Sci. 15 (14) (2009) 2894–2911.

[5] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, M.A. Taylor, T.S. Woodall, M.W. Sukalski, Architecture of LA-MPI, a network-fault-tolerant MPI, in: International Parallel and Distributed Processing Symposium, IEEE, 2004, p. 15.

[6] G. Fagg, J. Dongarra, FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world, in: ReCent Advances in Parallel Virtual Machine and Message Passing Interface, 2000, pp. 346–353.

[7] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, D. Solt, Run-through stabilization: An MPI proposal for process fault tolerance, in: Recent Advances in the Message Passing Interface, 2011, pp. 329–332.

[8] M.M. Ali, P.E. Strazdins, B. Harding, M. Hegland, Complex scientific applications made fault-tolerant with the sparse grid combination technique, Int. J. High Perform. Comput. Appl. 30 (3) (2016) 335–359.

[9] W. Bland, K. Raffenetti, P. Balaji, Simplifying the recovery model of user-level failure mitigation, in: Workshop on Exascale MPI at Supercomputing Conference, IEEE, 2014, pp. 20–25.

[10] M. Gamell, D.S. Katz, K. Teranishi, M.A. Heroux, R.F. Van der Wijngaart, T.G. Mattson, M. Parashar, Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques, in: 45th International Conference on Parallel Processing Workshops (ICPPW), IEEE, 2016, pp. 346–355.

[11] M. Gamell, K. Teranishi, J. Mayo, H. Kolla, M. Heroux, J. Chen, M. Parashar, Modeling and simulating multiple failure masking enabled by local recovery for stencil-based applications at extreme scales, Trans. Parallel Distrib. Syst. (2017).

[12] I. Laguna, D.F. Richards, T. Gamblin, M. Schulz, B.R. de Supinski, Evaluating user-level fault tolerance for MPI applications, in: European MPI Users' Group Meeting, ACM, 2014, p. 57.
[13] S. Pauli, M. Kohler, P. Arbenz, A fault tolerant implementation of multi-level Monte Carlo methods, Parallel Comput.: Accelerating Comput. Sci. Eng. 25 (2014) 471.
[14] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B. Debusschere, O. LeMaitre, O. Knio, ULFM-MPI implementation of a resilient task-based partial differential equations preconditioner, in: Workshop on Fault-Tolerance for HPC at Extreme Scale, ACM, 2016, pp. 19–26.
[15] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, G. Wellein, CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. CoRR abs/1708.02030.
[16] K. Teranishi, M.A. Heroux, Toward local failure local recovery resilience model using MPI-ULFM, in: European MPI Users' Group Meeting, ACM, 2014, p. 51.
[17] N. Losada, I. Cores, M.J. Martín, P. González, Resilient MPI applications using an application-level checkpointing framework and ULFM, J. Supercomput. 73 (1) (2017) 100–113.
[18] N. Losada, M.J. Martín, P. González, Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications, J. Supercomput. 73 (1) (2017) 316–329.
[19] L. Alvisi, K. Marzullo, Message logging: pessimistic, optimistic, and causal, in: International Conference on Distributed Computing Systems, IEEE, 1995, pp. 229–236.
[20] H. Meyer, D. Rexachs, E. Luque, RADIC: A faulttolerant middleware with automatic management of spare nodes, in: International Conference on Parallel and Distributed Processing Techniques and Applications (2012), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 1.
[21] A. Bouteiller, G. Bosilca, J. Dongarra, Redesigning the message logging model for high performance, Concurr. Comput.: Pract. Exper. 22 (16) (2010) 2196–2211.
[22] A. Bouteiller, T. Herault, G. Bosilca, J.J. Dongarra, Correlated set coordination in fault tolerant message logging protocols, in: European Conference on Parallel Processing, Springer, 2011, pp. 51–64.
[23] E. Meneses, C.L. Mendes, L.V. Kalé, Team-based message logging: preliminary results, in: International Conference on Cluster, Cloud and Grid Computing, IEEE, 2010, pp. 697–702.
[24] T. Ropars, T.V. Martsinkevich, A. Guermouche, A. Schiper, F. Cappello, SPBC: Leveraging the characteristics of MPI HPC applications for scalable checkpointing, in: International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 8.
[25] E. Meneses, L.V. Kalé, Camel: collective-aware message logging, J. Supercomput. 71 (7) (2015) 2516–2538.
[26] S. Rao, L. Alvisi, H.M. Vin, The cost of recovery in message logging protocols, Trans. Knowl. Data Eng. 12 (2) (2000) 160–173.
[27] H. Meyer, R. Muresano, M. Castro-León, D. Rexachs, E. Luque, Hybrid Message Pessimistic Logging. Improving current pessimistic message logging protocols, J. Parallel Distrib. Comput. 104 (2017) 206–222.
[28] F. Cappello, A. Guermouche, M. Snir, On communication determinism in parallel HPC applications, in: ICCCN, 2010, pp. 1–8.
[29] T. Martsinkevich, T. Ropars, F. Cappello, Addressing the last roadblock for message logging in HPC: Alleviating the memory requirement using dedicated resources, in: European Conference on Parallel Processing, Springer, 2015, pp. 644–655.
[30] X. Liu, X. Xu, X. Ren, Y. Tang, Z. Dai, A message logging protocol based on user level failure mitigation, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2013, pp. 312–323.
[31] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[32] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, J.J. Dongarra, Dodging the cost of unavoidable memory copies in message logging protocols, in: R. Keller, E. Gabriel, M. Resch, J. Dongarra (Eds.), Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 189–197.
[33] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in HPC applications, in: International Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2010, pp. 180–186.
[34] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/. Last accessed: 2018.
[35] Himeno Benchmark. http://accc.riken.jp/en/supercom/himenobmt/. Last accessed: 2018.
[36] E. Wolters, M. Smith, MOCFE-Bone: the 3D MOC mini-application for exascale research. Tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2013.
[37] TeaLeaf website. https://github.com/UoB-HPC/TeaLeaf. Last accessed: 2018.
[38] R. Rabenseifner, Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512, in: Message Passing Interface Developers and Users Conference (MPIDC99), 1999.
[39] A.A. Awan, K. Hamidouche, A. Venkatesh, D.K. Panda, Efficient large message broadcast using nccl and cuda-aware mpi for deep learning, in: 23rd European MPI Users' Group Meeting, ACM, 2016, pp. 15–22.
[40] Intel MPI Benchmarks. https://software.intel.com/en-us/imb-user-guide. Last accessed: 2018.

Nuria Losada: She received the B.S. (2013) and M.S. (2014) degrees in Computer Science from the Universidade da Coruña, Spain. Currently, she is a Ph.D. student in the Department of Computer Engineering at the Universidade da Coruña. Her research interests include fault-tolerance, parallel computing and malleability.

George Bosilca, Ph.D.: George Bosilca is a Research Assistant Professor at the Innovative Computing Laboratory at University of Tennessee, Knoxville. His research interests revolve around distributed algorithms, communication libraries and programming paradigms and practical constructs for parallel applications to maximize their efficiency, reliability, scalability and heterogeneity at any scale and in any settings.

Aurelien Bouteiller, Ph.D.: Aurelien Bouteiller received is Ph.D. from the University of Paris in 2006. He is currently a Research Director at the Innovative Computing Laboratory, the University of Tennessee, Knoxville. He focuses on improving performance and reliability of distributed systems with research in communication and scheduling for many-core, accelerated clusters, and in containing the influence of failures on scientific computation.

Patricia González, Ph.D.: She received the B.S. (1996), M.S. (1996) and Ph.D. (2001) degrees in physics from the University of Santiago de Compostela. Currently, she is an Associate Professor in the Department of Computer Engineering at the Universidade da Coruña. Her main research interests are in the area of High Performance Computing (HPC), focused on parallel and distributed computing and fault tolerance for parallel applications.

María J. Martín, Ph.D.: She received the B.S. (1993), M.S. (1994) and Ph.D. (1999) degrees in physics from the University of Santiago de Compostela, Spain. Since 1997, she has been on the faculty of the Department of Computer Engineering at the Universidade da Coruña, where she is currently an Associate Professor of Computer Engineering. Her major research interests include parallel algorithms and applications and fault tolerance for parallel applications.