Asynchronous Receiver-Driven Replay for Local Rollback of MPI Applications

Nuria Losada, Aurelien Bouteiller, George Bosilca
Innovative Computing Laboratory
The University of Tennessee, USA

***Abstract*—With the increase in scale and architectural complexity of supercomputers, the management of failures has become integral to successfully executing a long-running high-performance computing application. In many instances, failures have a localized scope, usually impacting a subset of the resources being used, yet widely used failure recovery strategies (like checkpoint/restart) fail to take advantage and rely on global, synchronous recovery actions. Even with local rollback recovery, in which only the fault impacted processes are restarted from a checkpoint, the consistency of further progress in the execution is achieved through the replay of communication from a message log. This theoretically sound approach encounters some practical limitations: the presence of collective operations forces a synchronous recovery that prevents survivor processes from continuing their execution, removing any possibility for overlapping further computation with the recovery; and the amount of resources required at recovering peers can be untenable. In this work, we solved both problems by implementing an asynchronous, receiver-driven replay of point-to-point and collective communications, and by exploiting remote-memory access capabilities to access the message logs. This new protocol is evaluated in an implementation of local rollback over the User Level Failure Mitigation fault tolerant Message Passing Interface (MPI). It reduces the recovery times of the failed processes by an average of 59%, while the time spent in the recovery by the survivor processes is reduced by 95% when compared to an equivalent global rollback protocol, thus living to the promise of a truly localized impact of recovery actions.**

***Index Terms*—fault tolerance, MPI, User Level Fault Mitigation, ULFM, message logging, checkpoint/restart**

I. Introduction

The rapid growth of computational science has led to a growing demand for faster and more capable hardware in high-performance computing (HPC) systems. These new systems are growing ever larger and more complex—using thousands of nodes with heterogeneous hardware configurations—and therefore have an increased opportunity for hardware faults, higher failure frequency, and lower mean time to failure (MTTF). 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. Looking ahead, 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 to not only ensure the timely completion of their execution in these systems but to also minimize the running costs.

Even though the Message Passing Interface (MPI) standard remains the most popular parallel programming model in HPC systems, it lacks 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. Its implicit simplicity makes this approach widely adopted. However, it is not the most efficient, as it implies aborting and re-spawning the entire application plus recovering all processes—including the ones not affected by the failure—from a previous point of the execution to repeat a computation that has, at least partially, already been done. At large scale, with failures impacting a small subset of the resources being used, it introduces large overheads that can be avoided.

The User Level Failure Mitigation (ULFM) interface [2], under discussion in the MPI forum, proposes the inclusion of resilient capabilities in the MPI standard. ULFM includes new semantics for process failure detection, communicator revocation, and reconfiguration—that is, a minimum set needed to repair the communication capabilities. In a previous work, we combined the ComPiler for Portable Checkpointing (CPPC) [3]—an application-level, open-source, checkpointing tool for MPI applications—and ULFM to implement a local rollback protocol for single program, multiple data (SPMD) applications [4]. Using this solution, only the failed processes are recovered from the last checkpoint, while consistency and further progress of the computation is enabled using ULFM and a message logging protocol. This solution introduces notable performance benefits with respect to a global rollback; however, we have identified limitations.

The current strategy for the replay of communications, notably collective operations, results in a synchronous recovery that prevents survivor processes from continuing their execution, thereby removing any possibility of overlapping the cost of recovery with computation. Although this overlap is tightly conditioned to the communication pattern of the application (i.e., the execution of a survivor process can only continue until the next communication that involves a recovering peer), recent trends aim at improving the strong-scaling capability of HPC applications by refactoring them in

terms of asynchronous execution [5]–[7] In addition to this, even in tightly coupled communication patterns, when a large number of point-to-point communications need to be replayed, we have identified that the efficiency of local recovery can be impacted by the large amount of resources that are required to carry out the re-execution of the communication at the recovering processes.

To solve both issues, in this work we implement an asynchronous, receiver-driven local recovery. First, we modify the replay of collective operations by replacing them with point-to-point communications to heavily reduce the number of processes that need to participate in the replay. We focused on `MPI_Allreduce`, which, as reported by different profiling studies [8], is by far the most commonly used MPI collective operation, both in terms of the number of times the reduction function is called and the total amount of time spent in it. We also discuss how this strategy can be applied to other collective operations. Secondly, we implement a receiver-driven replay of communications exploiting the remote-memory access (RMA) operations provided by MPI. This RMA-based replay enables the recovering processes to fetch the message content from the memory of their passive peers, which remain free to continue their own computation. In addition, recovering processes obtain the messages at a pace that is commensurate with the speed of their progress. The combination of both techniques disengages the survivor processes from the replay of communications, while it prevents the performance penalties introduced by traditional message logging replay, in which message bursts can cause resource exhaustion and expensive out-of-order message management. To the best of our knowledge, no other message logging protocol implements a completely receiver-driven replay for point-to-point or collective communications.

The rest of this paper is structured as follows. Section II provides an overview of local rollback recovery protocols. Section III describes the optimization for the replay of collectives operations, while the receiver-driven replay is presented in Section IV. The experimental evaluation is reported in Section V, and Section VI concludes this paper.

II. Message Logging Background

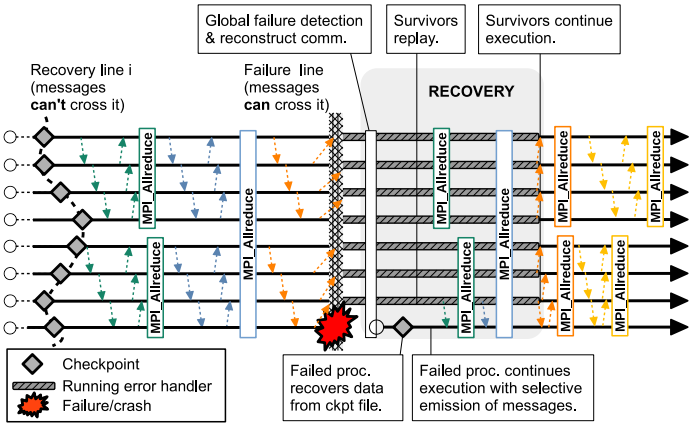
Message logging has been an active research topic in the recent decades [9]–[16]. This technique relaxes the synchronization constraint in coordinated checkpoint/restart, and enables rolling back only the processes affected by a failure. Message logging describes the execution of a process as a sequence of process states and events. An event corresponds with a computational step or a 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 [17].

a) Event Logging: Events can be deterministic or non-deterministic, depending if —from a given state—the event always result in an identical outcome state. Processes are considered "piecewise deterministic:" few non-deterministic events occur, separated by large sections of deterministic computation. This is a reasonable assumption in HPC workloads where the main source of non-deterministic events is the relative order of message receptions. After a failure, a replacement process can be driven to reach the same state as the original execution by repeating the all events in the 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 forced for non-deterministic events, and thus, they must be logged during the original execution. Different techniques (pessimistic, optimistic, and causal) provide different levels of synchronicity and reliability for the logging of events [9], which can help reduce the incurred latency overhead. Nonetheless, the dominating costs in message logging techniques are related to the time to log the message payload (i.e., the bytes transferred over the network) and the associated memory requirements.

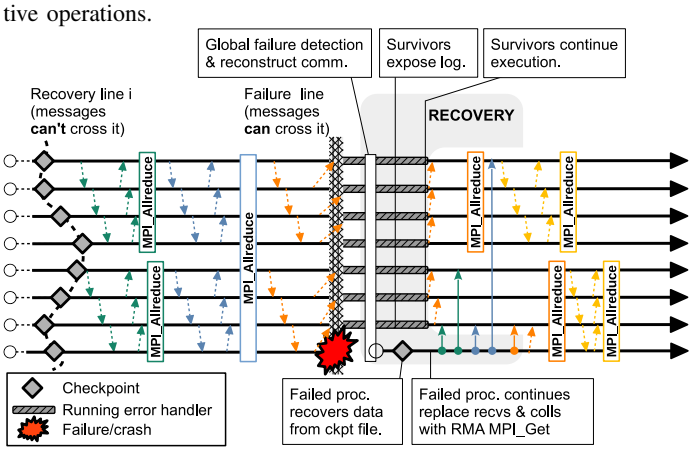
b) Payload Logging Techniques: In addition, to replaying non-deterministic events, recovering processes need to replay any message reception that impacted their state in the original execution (deterministic or not). Therefore, message logging protocols have two fundamental parts: (1) the aforementioned logging of non-deterministic events and (2) the logging of the payload of the messages that permits replaying past messages without restarting sender processes. Receiver-based approaches [15] perform a local copy of the message contents at the receiver. The main disadvantage is that the messages need to be committed to stable storage or to a remote repository to make them available after the process fails, which can be costly [16]. On the other hand, in sender-based strategies [10]–[14], the logging is performed on the sender process. The logging consists in a local copy that can be made in parallel with the network transfer, and kept in local memory; if the process fails, the log is lost, but it will be regenerated during the recovery. As a drawback, during the recovery, failed processes need to request the messages from survivor processes. Hybrid solutions have also been studied [18] that mix receiver- and sender-based logging. In this work we focus on enabling an asynchronous replay of the sender-based log.

An orthogonal strategy to reduce the log volume consists in forming groups of processes that coordinate their checkpoints [11]–[13]. In this method, processes within a group spare the expense of logging of message payload for any communication within the group. As a consequence, when a process fails, the entire group has to roll back. While effective at reducing the total log volume, these strategies still rely on a traditional sender-based approach for inter-group messages.

c) Logging Collective Communications: In traditional message logging, collective communications are logged by logging the internal Point-to-point messages that compose them. Recent advances consider the semantic aspects of the collective operation to optimize the log volume. In [14] the authors log only a subset of the point-to-point messages sufficient to reconstruct the outcome when needed, while in [19], the result (i.e., the receive buffers) is logged rather than



(a) Original replay: re-sending point-to-point and re-executing collective operations.



(b) Asynchronous receiver-driven replay.

Fig. 1: Local rollback protocol

the internal intermediate messages. Similarly, our approach does not log the full contingent of Point-to-point messages in collective operations, but in contrast, the main goal of our novel collective logging strategy is to permit a passive replay from the perspective of survivor processes.

In previous works [4], we designed an hybrid local recovery protocol that seats partially within the MPI implementation and at the user level. On one hand, Point-to-point communications and non-deterministic events are logged by the Open MPI VProtocol [10] component at the MPI library level. On the other hand, the CPPC tool instruments the code to log collective communication at the application level. This has the effect of decoupling the logging protocol from the particular collective implementation and avoids the logging of the individual point-to-point messages. During the recovery, ULFM is used to restore communication capabilities before the survivor processes resend the necessary point-to-point communications to the recovering ones using a traditional sender-based approach, while collective communications are re-executed as originally (i.e., by reposting collective calls involving all processes in the communicator). Figure 1a illustrates the operation of the protocol after a failure.

d) Identified Issues and Goals: Although message logging strives to localize the recovery of failures, we can see that existing techniques do not completely achieve that goal. Despite the fact that only failed processes restart from a checkpoint, surviving processes (i.e., processes not directly affected by the failure) play an integral, active part in helping the recovery progress, and are in essence busy serving messages and replaying synchronizing communication for the entire period. Thus, in this work, we explore how introducing an asynchronous, receiver-driven, local recovery helps minimize the involvement of survivor processes. To this end, by changing the way we log and replay the collective payload, we relax the synchronization constraint imposed by most collective communications by converting their collective behavior into a simpler, more peer-wise behavior, and thus we avoid the re-execution of collective communications by all processes. In addition, we have identified (and will characterize in the experiments) that rendering the recovery asynchronous can cause burst of network activity when survivor processes send the message log to restarting processes—at the extreme, the entire communication volume since the previous checkpoint. To alleviate this problem, we propose exploiting the MPI RMA capabilities driven by restarting processes themselves. As illustrated in Figure 1b, recovering processes fetch message content directly from their peer memory, while processes not affected by the failure are freed from any replay commitments, and can—potentially—continue their execution if the communication pattern if the application allows it.

III. Reducing the Synchronicity in the Replay of Collective Operations

Re-executing the collective operations during the replay provides a straightforward and always-correct strategy for the deployment of the message logging protocol. However, owing to the synchronizing behavior of most MPI collective communications, this strategy—in most cases—is not the most appropriate. First of all, it introduces the unnecessary communication of data that is going to be discarded by all surviving processes (i.e., processes that did not need the old data). For example, re-executing a `MPI_Allreduce` operation implies transmitting data to all peers in the communicator, but this data is going to be discarded by all survivor processes. Secondly, this strategy forces a synchronous recovery: all survivor processes in the communicator are blocked in the replay—even if they do not need to resend any other messages to recovering peers—and all the processes need to replay all collectives in the original order. Replacing the re-execution of collective operations by one or a set of point-to-point communications could simultaneously achieve two goals: (1) avoid sending unnecessary messages and (2) reduce the synchronicity of the recovery. In many cases, optimizing the replay of a collective operation requires changing the way it is logged. All in all, changes in the logging/replay are performed at the CPPC or Vprotocol level, and thus, are oblivious to the application code.

In this work, we apply this strategy for the optimized replay of the `MPI_Allreduce` operation to demonstrate the bene-

fits that can be introduced during the recovery. In the failure-free execution of a `MPI_Allreduce`, the result data is received by all peers in the communicator. By using a receiver-based logging for this operation, all survivor peers will have the resulting data in their log, and any of them can provide it to a recovering process using point-to-point communications. This approach compares favorably with logging individual point-to-point messages, both in terms of log volume and potential for introducing performance overhead. However, it remains similar to the synchronous replay of collective operation (with sender-based logging) both in terms of logged data volume and performance cost, since the vector that is logged at the receiver is strictly the same size as the one logged at the sender. This method is however significantly different during the recovery. After a failure, before starting the replay, all processes agree about a *root* responsible for resending the result data to the recovering processes. As collective operations involve all processes in the communicator, this agreement is performed once for each one of the communicators used by the application, thereby guaranteeing a valid root for the replay in all of cases. In this work, the agreement designates the lowest survivor rank in the communicator as a *root* process. However, the *root* agreement could also be performed by taking into account the processes that failed, choosing a survivor in the same or nearby node for the replay, or even having several *root* processes in charge of the replay for a subset of nearby failed processes in order to distribute the communication load.

This strategy is not restricted to the `MPI_Allreduce` collective and can easily be generalized to all collective operations in MPI [8]. In the case of `MPI_Bcast`, no changes need to be applied during the logging operation. The root process can send the data exclusively to the recovering peers—whether a survivor process's log contains the data or whether it has failed and the data is regenerated as it progresses through the execution. Similarly, all-to-all collectives can be replaced by point-to-point communications for the recovering processes without modifying how they are logged, as in the sender-based logging, each peer saves its own contribution. Even though all processes still need to participate in the replay, the unnecessary transmission of data to survivor peers is avoided.

For `MPI_Barrier`, the synchronizations must be maintained between recovering processes; however, survivor processes do not need to be involved, since by definition they have already participated in the barrier. Therefore, an optimized replay can be implemented by running the barrier on a smaller communicator that includes only the subset of recovering processes. This strategy would involve creating new derived communicators containing only the recovering peers at the beginning of the replay process, and this will not impact the logging overhead. Note that MPI operations like `MPI_Comm_create_group` permit creating small piecemeal communicators without involving all processes.

The case of the `MPI_Reduce` operation is more complex, as the result is only received by the root process, and that peer is the only one that would need it in case it is hit by a failure. Thus, performing an optimized replay in this case requires the result data to be logged on a stable remote server, or to multiple peers, which adds logging overhead. All in all, the replay of the reduction operation can be partially optimized by avoiding the replay when the root is not one of the recovering processes and, otherwise, re-executing it.

IV. Enabling a Receiver-Driven Replay

After applying the aforementioned optimization, the replay procedure now consists of a subset of survivor processes explicitly resending a subset of the logged communications as point-to-point messages. The subset of replaying survivor processes includes those that have sent messages to a process that has failed, as well as those survivors designated as a *root* for serving collective communication results. Most certainly, survivor processes can resend messages at a faster pace than a recovering peer can consume them. Indeed, the latter does not only replay communication, but also repeats the computation to recover its state, hence, it posts message receptions at a much slower rate.

A survivor process is thus facing multiple axis of tradeof when deciding at what rate the emission of message log should proceed. At an extreme, the survivor processes could be allowed to replace all collective operations with a series of non-blocking point-to-point communications and replay the message log as fast as possible, injecting the entire volume immediately. That approach, which represents the state of the practice, can lead to multiple issues. First, issuing such a large number of non-blocking can exhaust software and hardware resources alike. For example, both the available pool of MPI requests and congestion credits at the network interface level are limited. Conversely, the recovering process will receive a massive amount of concurrent, unexpected messages from multiple peers, causing a resource exhaustion and congestion issue at the receiver. Last, despite the communication being pending, if survivor processes resume computation immediately and stop monitoring the completion of these posted communication, they may stop progressing. The MPI standard does not mandate that point-to-point communication shall progress if no call to MPI routines are made, hence after an initial burst, communication may actually stop and delay the progression of the recovery. At the other extreme, this scenario can certainly be avoided by implementing a synchronizing protocol to slow down the replay by the survivor processes, thereby forcing them to wait for acknowledgments from recovering processes before resending more messages. However, this defeats the goal of achieving an asynchronous recovery, as it keeps survivor processes involved in these synchronization, thereby eliminating the opportunity to overlap the recovery with further computation.

The alternative solution we propose is to exploit the RMA capabilities provided by MPI to enable the recovery process to control when (and at what pace) it obtains the necessary logged messages by directly accessing the message log in the passive survivors' memory. The one-sided features provided by MPI [20] enable the creation of RMA *windows* in a given communicator, to which memory regions can be

subsequently attached. These memory regions are exposed to the rest of the processes in the communicator (i.e., remote peers can both read from and write to them). RMA communication calls (e.g., `MPI_Get`) must be issued within an access epoch. Access epochs are started and completed using synchronization calls; In this work, we use the passive target RMA `MPI_Win_lock(MPI_LOCK_SHARED,...)` and `MPI_Win_unlock` to start and complete an access epoch from recovering processes, and `MPI_Win_flush` to complete RMA operations.

a) Exposing Message Logs: First, we restrict the number of processes participating in the RMA window by creating a communicator that includes only the recovering processes and surviving processes with logs containing messages to be replayed. Restricting the number of processes participating in the window can reduce the cost of some RMA operations, especially at larger scales, where—potentially—a very small subset of the application processes need to be involved in the recovery. On this dynamic window, the survivor processes attach (using `MPI_WIN_ATTACH`) the memory segments that contain their log to make it accessible to the recovering peers. To fetch the messages from a remote peer, the recovering processes need to know the exact address in the remote memory where the data is stored. Therefore, survivor processes notify the recovering ones of the base address in the log for the relevant communication channels (i.e., for each pair recovering the destination process and communicator). Note that after this stage, survivor processes have no further participation and are free to continue executing the remainder of the application, as they only appear as passive targets in RMA operations.

b) RMA Replay of Communications: The first key aspect of the replay corresponds to how the recovering peers access the remote survivors' logs. We employ sender-based message logging, and thus the messages to be replayed are available from the survivors' logs. However, for implementation reasons, messages to the same destination process over the same communicator are not necessarily stored contiguously in the log. Instead, messages are saved in their post order from the sender perspective (i.e., the order of calls to `MPI_Send/Isend`), regardless of the destination process or communicator. This strategy enables the allocation of a memory chunk in which messages are logged as they are being sent, and additional memory only needs to be allocated when the previous segment is completely full. Alternatives, such as trying to group messages by receiver processes and/or by communicator, complicate the logging operation and increase the logging overhead, thereby introducing problems such as predicting the amount of data that will be sent to each peer and fragmentation in the log. It is entirely possible to reorganize the log content upon certain conditions (e.g., long intervals between checkpoints, memory constraints, remote checkpoint), but this is not part of the study presented here. All in all, the most efficient way for recovering processes to iterate through the sender's log is by accessing only those messages sent to them in the past. Thus, minimizing the number of remote accesses that are performed to obtain the non-contiguous log becomes an important issue. To enable the iteration over all—and only—those messages, we include additional information alongside the message log. The sender-based log structure metadata contains the information that enables the replay (e.g., for a point-to-point communication: size of the data, communicator ID, destination process, and tag). We extend it to also include the offset in the log of the next message in the same communication channel (same peer and communicator). The cost of maintaining this information during the failure-free operation is negligible compared to the cost of the logging, but it enables the recovering peers to be completely independent and to locally compute the remote addresses of the messages it needs to replay—all without sequentially iterating over each message in the remote log.

As the recovering processes progress in the execution, they internally translate the receptions or collective operations into `MPI_Get` operations. Note that, in a naive implementation, two `MPI_Get` operations are necessary for each message: one to fetch the metadata and another one to get the actual contents of the message using the appropriate buffer size indicated in the metadata. These two operations cannot be overlapped as the first get operation must complete for the recovering process to obtain the displacement of the payload in the sender's log as well as the size of the payload. Once this information is locally available, the second get obtains the content of the message in the reception buffer. In addition, for each get operation a new access epoch has to be completed in order to ensure the visibility of the data in the reception buffers. However, RMA operations are more efficient when accessing contiguous data (one `MPI_Get` operation gets a large chunk of data) or in aggregated modes (performing N remote operations within the same access epoch). We implement two optimizations to get closer to such a contiguous access pattern. First, we delegate part of the metadata assembly to the survivor processes involved in the recovery. They copy the metadata of the messages to be replayed for each recovering peer in a continuous buffer. This enables the recovering process to get the metadata of N_{mt} messages using only one get operation, which can be overlap with other operations such as attaching the actual memory segment containing the log. Using this optimization, we move from performing N get operations each one in a different access epoch (as each operation needs to be completed to calculate the address of the metadata of next message), to a single get operation on a contiguous memory region in a single access epoch (at the expense of N local memory copies at the survivor). It should also be noted that as each survivor process compute the aggregated metadata independently, the aggregations happen in parallel and we remove the sequential behavior imposed when executing the aggregation operation on the recovering processes. Second, we prefetch the messages payload (i.e. the message contents) of N_{pf} messages. Upon the first reception in a communication channel, the next N_{pf} messages are also fetched in the same access epoch and kept in a local buffer in packed form. The next receptions in the communication channel will consume the messages in the local buffer until it is

TABLE I: Original run times of the testbed applications in minutes.

	256P/216P	512P	1024P/1000P
Himeno	8.88	4.46	2.40
Mocfe	3.03	1.56	0.86
Tealeaf	4.57	2.37	1.88
Lulesh	9.06	3.26	1.58
Stencil	1.47	0.99	0.52

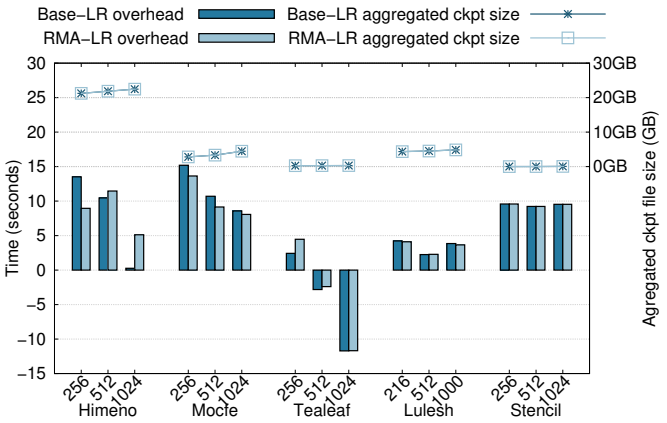


Fig. 2: Absolute overheads (in seconds) and checkpoint file sizes in the absence of failures.

empty. Both strategies reduce the cost of the RMA operations while they maintain the memory consumption at the recovering process under a configurable threshold.

V. Experimental Evaluation

The experimental evaluation was performed at the NERSC Cray XC40 Cori supercomputer. The nodes in the machine are composed of two Intel Haswell E5-2698 v3 processors running at 2.30 GHz, with 16 cores per processor (32 cores per node), and 128 GB of RAM. The nodes are connected using a Cray Aries with Dragonfly topology (5.625 TB/s global bandwidth (Phase I), 45.0 TB/s global peak bidirectional bandwidth (Phase II)). The experiments spawned 32 MPI process per node (one per core). For our testing, we used CPPC version 0.8.1, working with HDF5 version 1.8.11 and GCC version 7.3.0. The Open MPI version used corresponds with ULFM 2.0 and was modified for the integration of VProtocol and CPPC. The Portable Hardware Locality (hwloc) [21] package was used for binding the processes to the cores. Applications were compiled with optimization level O3. Checkpoint files are dumped to the local disk of the nodes. We report the average times of 10 executions.

We used an application testbed comprised of five domain science MPI applications with different checkpoint file sizes and communication patterns. We ran the Himeno benchmark [22], a Poisson equation solver, fixing NN to 12,000 with a grid size of $1024 \times 1024 \times 512$. MOCFE-Bone [23], which simulates the main procedure in a 3-D method of characteristics (MOC) code, using 4 energy groups, 8 angles, a mesh of 16×24^3 doing strong scaling in space, and a trajectory spacing of $0.01cm^2$. TeaLeaf [24] 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 200 time steps. Lulesh [25], [26] represents a typical hydrocode; it approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It was run with fixing the number of cycles to 500 and the number of elements to 110 592 000, by setting the length of cube mesh along side (-s parameter) to 80, 60, and 48 as the application scale out. Finally, the Stencil benchmark from the Parallel Research Kernels [27] applies a data-parallel stencil operation to a two-dimensional array. It was run with radius 10, gridsize 320, and 650k iterations. This benchmark is used to exemplify the behavior in communications patterns containing a large number of point-to-point communications but no collective operations. The original execution times of the applications, in minutes, are reported in Table I. Experiments were executed doing strong scaling (i.e., maintaining the global problem size constant as the application scales out). The block size used to get the metadata (N_{mt}) was set to 512 and the number of message prefetched (N_{pf}) in blocks of 32.

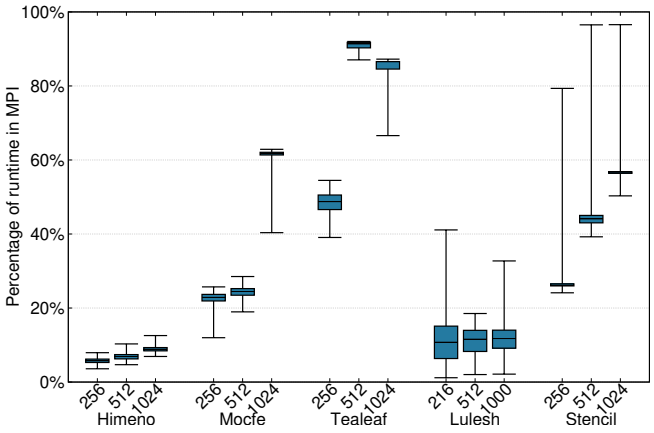
In all experiments, the checkpointing frequency is fixed so that two checkpoint files are generated during the execution of the applications—the first is generated at 40% of the execution progress and the second is generated at 80%.

Figure 2 shows: (1) the overhead when logging to enable the original replay protocol with point-to-point resending and collective re-execution (denoted as base-LR) and (2) the RMA version described in this work (denoted as RMA-LR). The overhead and checkpoint file sizes show no significant differences between the two approaches. The modifications introduced by the RMA-LR present little impact on the overhead, and the overhead remains low. In the case of Tealeaf, the fault-tolerant execution is slightly faster than the original run, conceivably because of changes in the optimization applied by the compiler in the CPPC-instrumented code.

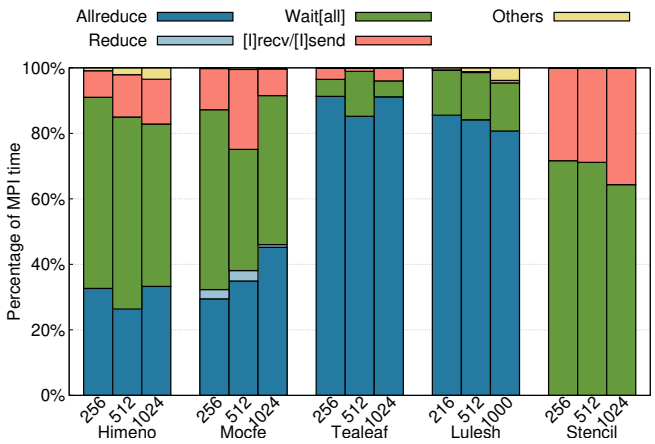
A. Characterization of the Communication Pattern

We use mpiP [28] to analyze the MPI usage of the original applications.Figure 3a reports the percentage of the application runtime that is spent in MPI using a box plot to represent the variability across processes, and Figure 3b breaks it down into the operations that consume the most time. Mocfe, Tealeaf, and Stencil are the applications that spent the largest percentages of their execution in the communication library, specially when scaling out. Lulesh and Stencil are the applications with the largest variability across different processes. Regarding the break down of MPI time, Allreduce collective operations account for most of the time for Tealeaf and Mocfe, while completion calls of point-to-point communications dominate the MPI time for the rest of applications.

We also analyze the data being logged for each application and characterize the applications in terms of their log volume



(a) Percentage of the application's runtime spent in MPI and variability across processes.



(b) MPI time broken down by most consuming communication calls.

Fig. 3: Benchmarks profiling information reported by mpiP.

TABLE II: Benchmark characterization: MPI calls and log behavior per iteration.

	MPI calls	#Procs	#Iters	Aggregated average log behavior per iteration			
				Point-to-Point		Collective communications	
	Main loop			#	Size	#	Size
Himeno	`Irecv, Isend, Wait, Allreduce`	256	12K	1.3K	98.6MB	256	7.0KB
		512	12K	2.6K	165.2MB	512	14.0KB
		1024	12K	5.4K	201.5MB	1.0K	28.0KB
Mocfe	`Irecv, Isend, Waitall, Allreduce, Reduce`	256	120	2068.5K	1.4GB	51.5K	11.2MB
		512	120	4343.8K	1.8GB	102.9K	22.5MB
		1024	120	8894.5K	2.8GB	205.8K	45.0MB
Tealeaf	`Irecv, Isend, Waitall, AllReduce`	256	200	864.8K	3.4GB	460.5K	14.1MB
		512	200	1757.1K	5.3GB	920.4K	28.1MB
		1024	200	3578.1K	7.3GB	1844.1K	56.3MB
Lulesh	`Irecv, Isend, Wait, Waitall, AllReduce`	216	500	6.9K	489.6MB	216	6.7KB
		512	500	17.9K	693.5MB	511	16.0KB
		1000	500	36.8K	902.1MB	998	31.2KB
Stencil	`Irecv, Isend, Wait,`	256	650K	960	1.48MB	0	0
		512	650K	2K	2.27MB	0	0
		1024	650K	4K	3.07MB	0	0

in Table II. We report the MPI routines that are called in the main loop of the application (where the checkpoint call is located). For each experiment, the table also shows the number of processes running the experiment, the total number of iterations run by the application, and the aggregated average log behavior per iteration. The aggregated average log behavior per iteration is computed as the sum of the logs from all processes (i.e., the total log generated by the application in one iteration) of the point-to-point and collective logs in terms of number of calls and size of the data that is logged. Note that, even though Mocfe runs `MPI_Reduce` operations in the main loop of the application—with the reported problem configuration—it does so in a communicator containing only one process; thus, those communications are neither logged nor replayed, as they will be naturally regenerated if that process fails. Therefore, in all test-bed applications, the logged communications correspond with point-to-point and `MPI_Allreduce` operations, and none of them generate non-deterministic events during their whole execution. Additionally, Figure 4 reports the aggregated log parameters—the addition of the values from all processes—when checkpointing, that is, the maximum size during the execution, as the logs are cleared after checkpointing. For all applications, point-to-point communications account for most of the log. Even in those cases in which most of the MPI time is spent in collective communications, the amount of data transmitted is very low, and the bulk of communications—both in terms of number of calls and transmitted data—correspond to point-to-point communications. Himeno has the largest maximum point-to-point log size, ranging from 462 GB to 945 GB when scaling out, followed by Stencil (from 375 GB to 781 GB), Tealeaf (from 273 GB to 583 GB), while for Mocfe and Lulesh, it remains under 176 GB. Regarding the collective log size, Mocfe (2 GB maximum) and Tealeaf (4 GB maximum) have the largest ones. On the other hand, Himeno's collective log size remains under 131 MB, and it is particularly low for Lulesh (6 MB maximum).

B. Recovery after a Failure

In this section, we evaluate the performance of the recovery when introducing a failure by killing the last rank in the application when 75% of the computation is completed. We compared three recovery procedures: global rollback, local rollback using the original replay of communications, and local rollback using the RMA replay with optimized collectives. In all of them, the ULFM capabilities are used to detect the failure and restore the communication capabilities. Due to practical constraints with allocating supplementary processes at runtime in production systems, a set of oversubscribed spare replacement processes are spawned at the beginning of

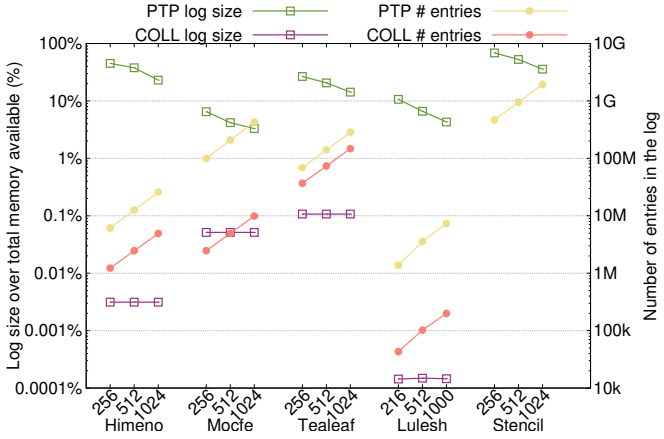
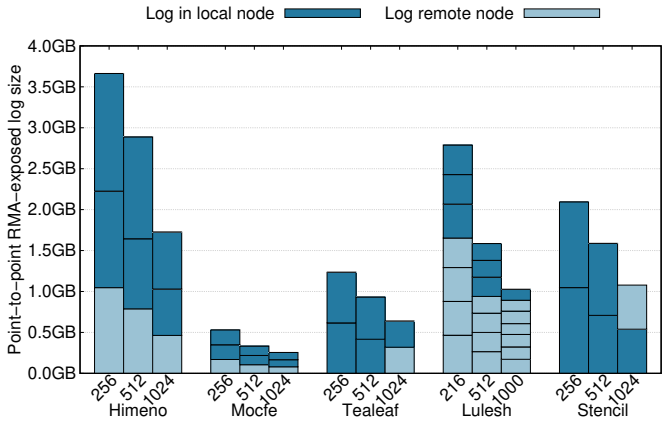
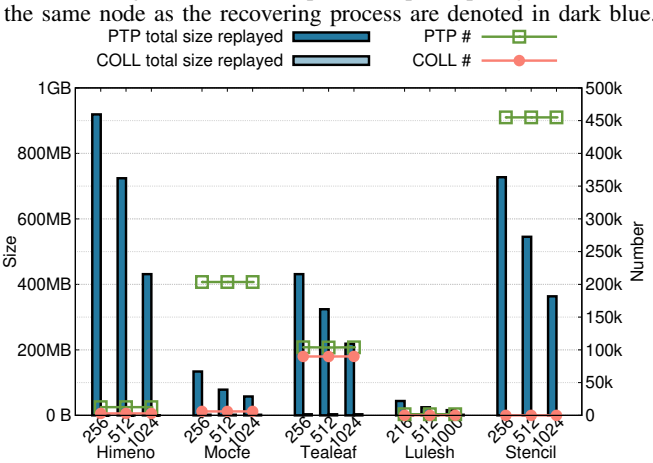


Fig. 4: Log parameters when checkpointing: maximum log sizes expressed as the percentage of the total memory available and number of entries during the execution.



(a) Size of log for the survivor processes participating. Survivors on the same node as the recovering process are denoted in dark blue.

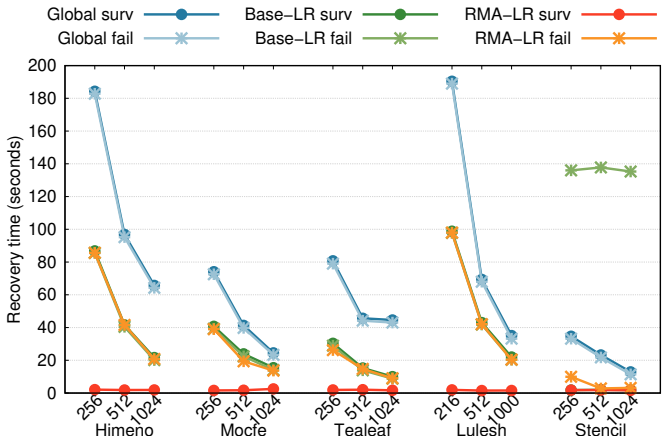


(b) Communications replayed during the recovery.

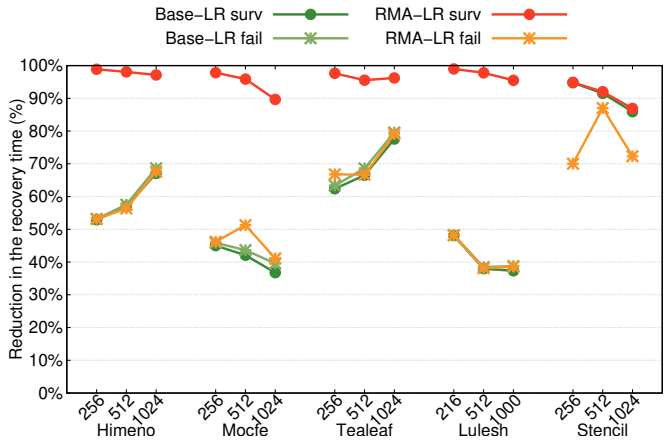
Fig. 5: Analysis of the log size and the replayed communications during the recovery.

the execution and remain inactive (nanosleep) until they need to replace a failed rank. The recovering process restores the application data from the checkpoint file, and the replay of communications takes place. When using base-LR replay, the necessary point-to-point communications are re-sent by the survivor peers, and collective operations are re-executed. On the other hand, using the RMA strategy, the recovering peer directly obtains the message's contents from the remote survivors' logs as it advances in the execution. In all experiments, failures are introduced at the same point of the execution of the last rank, and, therefore, the logs of the survivors processes and the communications that need to be replayed present almost no differences between the two local rollback protocols. Figure 5a shows the total size of the log at each one of the survivor processes that have point-to-point messages that need to be replayed to the failed peer, indicating with different colors if the survivor process is in the local node or in a remote node with respect to the recovering process. In the RMA protocol, the point-to-point logs account for the majority of the memory that is attached and exposed in the one-sided windows. Figure 5b reports the communications that are actually replayed (those routed to a failed process) in each experiment, representing the amount of data and the number of operations for point-to-point and collective communications. In all applications, the bulk of the replayed data corresponds with point-to-point communications, from a small number of neighboring processes to the recovering peer. For Tealeaf and Stencil, around 65% of the point-to-point logs exposed in the window are actually being replayed, while 23% are replayed for Himeno and Mocfe, and 1.45% are replayed for Lulesh. On the other hand, the amount of replayed data related to collective operations is always less than 3.2 MB.

Figure 6a reports the time spent in the recovery (in seconds) for the replacement and the survivor processes. For three testbed strategies, the recovery times include the time spent once the communication capabilities had been restored and until the state prior to the failure had been reached. Figure 6b reports the percentage reduction in the recovery times that the local rollback strategies achieve with respect to the global rollback. In the case of applications replaying collectives communications (all but Stencil), the base-LR replay reduces the recovery times of both failed and survivor processes, on average, by 53% compared to global rollback. The RMA-LR protocol maintains the same performance benefits in the recovering processes, while the survivor recovery times are reduced to a minimum (less than 3 seconds), which means a reduction of 95% compared to a global rollback. The case of the Stencil benchmark presents a different scenario when using the base-LR protocol: the absence of collective communications in this application allows survivor processes to replay their communications unhindered by the progress of the recovering processes. This results in the survivor processes posting all possible sends for the recovering processes and then continuing their execution. All of these early sends translate on the receiver side in unexpected messages, leading to catastrophically slower recovery. The recovering processes are overwhelmed by the sheer amount of communications being replayed, and their recovery times are heavily increased, being between $4\times$ and $11\times$ the cost of a global rollback. The main difference between this benchmark and the other applications is the number

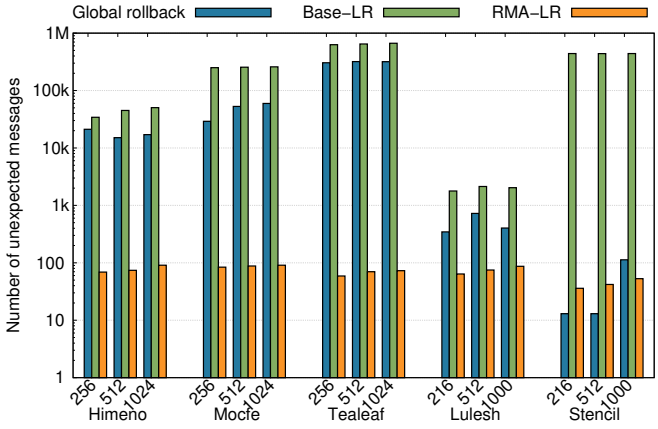


(a) Recovery times (in seconds) for the survivor and failed processes for the different resilience strategies: global rollback, local rollback with point-to-point replay, and local rollback with RMA and optimized collective replay.

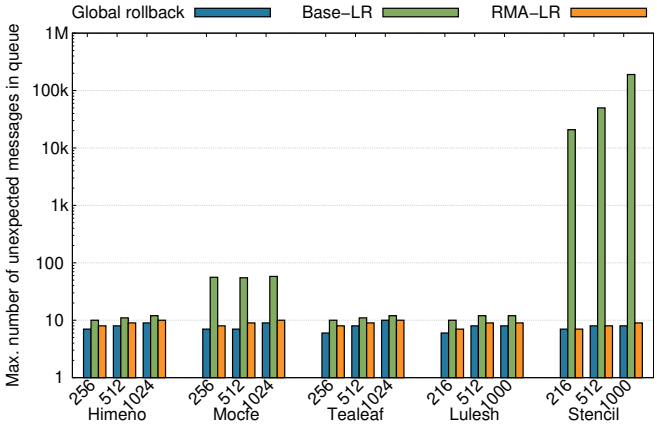


(b) Percentage reduction in the recovery times for the survivor and failed processes when using a local rollback (with point-to-point or RMA replay) with respect to the global rollback recovery times.

Fig. 6: Recovery times using the different protocols.



(a) Total number of unexpected messages.



(b) Max number of unexpected messages in the queue (watermark).

Fig. 7: Unexpected messages generated during the recovery.

of point-to-point communications that are replayed without the need for a synchronizing collective and the number of unexpected messages that are introduced. Using the SPC counters capability [29] included in Open MPI, we measured the impact of the unexpected messages generated during the recovery for each of the resilience strategies. Figure 7 reports the different events related to unexpected messages, including the total number of unexpected messages that are generated and the maximum number of messages in the queue at a given time during the recovery. The unexpected messages generated during the global rollback recovery serve as a baseline and correspond with the unexpected messages that were originally generated in that part of the execution. For all the applications, the base-LR protocol worsens all of the events related to the unexpected messages, as all replayed point-to-point messages are unexpected at the recovering process. However, the re-execution of collective operations keeps the maximum number of unexpected messages in the queue at the same time and keeps their matching time low. Re-executing collective operations introduces synchronization points that slow down the replay carried out by survivor processes, thereby providing the recovering peers a chance to catch up. In the case of Stencil, the replay without pace performed by survivors heavily increases the number of unexpected messages. Note, however, that the mere presence of collective operations does not guarantee that the number of unexpected messages will be kept low. Communication patterns where large numbers of point-to-point communications need to be replayed between two consecutive collective operations will present performance penalties similar to those of the Stencil benchmark.

All in all, the RMA-LR protocol solves the problems introduced as a side effect by the base-LR with unexpected messages, and the recovery times of the failed processes are reduced by 59% across all applications, while the time spent by survivor processes in the recovery procedure is reduced to a minimum (less than 3 seconds), which means a reduction of 95% compared to a global rollback. The overall failure overhead presents the same reduction trend than the

recovery times of the failed processes. In less tightly coupled communication patterns, like those present in asynchronous algorithms, the RMA protocol will not only eliminate the problems introduced by unexpected messages, but also benefit from overlapping the recovery of the failed processes with further computation in the execution of the survivor processes.

VI. Concluding Remarks

We extended our previous work on a local rollback protocol [4] by exploiting MPI one-sided communications and implementing optimizations to cope with the replay process limitations. With the RMA approach, the recovering peers obtain the message contents at the exact point of the execution when they need it—thereby limiting the amount of resources necessary for the recovery—while the prefetching of metadata and aggregation of messages amortize the setup cost of the RMA operations. A first optimization relaxes the synchronicity imposed by the complete replay of collective operations that prevents survivor processes from continuing their execution, increasing the opportunity to overlap the progress of the recovering processes with further computation on survivor peers. We also modify the way collective operations are logged, with `MPI_Allreduce` in particular, to enable their replay as point-to-point communications—a strategy that reduces the amount of data transmitted and limits the participation in the recovery to the strictly necessary set of survivor processes. A second optimization streamlines the uncontrolled replay of point-to-point communications (or collective communication replayed as point-to-point) improving the efficiency of the local recovery and drastically decreasing the amount of resources required at the recovering processes. The resulting asynchronous, receiver-driven replay of communications reduces the recovery times of the failed processes by 59% on average compared to a global rollback, while the participation of survivor processes is minimal (reduced by 95%).

Acknowledgements

This research was supported by the National Science Foundation of the United States under award #1664142 and #1725692 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy's Office of Science and the National Nuclear Security Administration under UT Battelle subaward 4000153505.

References

[1] C. Di Martino, Z. Kalbarczyk, and 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, and J. Dongarra, "Post-Failure Recovery of MPI Communication Capability: Design and Rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[3] G. Rodríguez, M. J. Martín, P. González, J. Tourino, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[4] N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín, "Local Rollback for Resilient MPI Applications with Application-Level Checkpointing and Message Logging," *Future Generation Computer Systems*, vol. 91, pp. 450–464, 2019.

[5] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *The International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 97–111, 2014.

[6] H. Avron, A. Druinsky, and A. Gupta, "Revisiting asynchronous linear solvers: Provable convergence rate through randomization," *Journal of the ACM (JACM)*, vol. 62, no. 6, p. 51, 2015.

[7] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, "Performance of asynchronous optimized Schwarz with one-sided communication," *Parallel Computing*, vol. 86, pp. 66–81, 2019.

[8] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 30.

[9] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, and causal," in *International Conference on Distributed Computing Systems*. IEEE, 1995, pp. 229–236.

[10] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the Message Logging Model for High Performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010.

[11] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Correlated set coordination in fault tolerant message logging protocols," in *European Conference on Parallel Processing*. Springer, 2011, pp. 51–64.

[12] E. Meneses, C. L. Mendes, and L. V. Kalé, "Team-based message logging: Preliminary results," in *International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 697–702.

[13] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello, "SPBC: Leveraging the characteristics of MPI HPC applications for scalable checkpointing," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[14] E. Meneses and L. V. Kalé, "Camel: collective-aware message logging," *The Journal of Supercomputing*, vol. 71, no. 7, pp. 2516–2538, 2015.

[15] H. Meyer, D. Rexachs, and E. Luque, "RADIC: A Fault Tolerant Middleware with Automatic Management of Spare Nodes," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2012, p. 1.

[16] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," *Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, 2000.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[18] H. Meyer, R. Muresano, M. Castro-León, D. Rexachs, and E. Luque, "Hybrid Message Pessimistic Logging. Improving current pessimistic message logging protocols," *Journal of Parallel and Distributed Computing*, vol. 104, pp. 206–222, 2017.

[19] X. Liu, X. Xu, X. Ren, Y. Tang, and 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.

[20] M. Forum, "MPI: A message-passing interface standard (V3.0)," 2012.

[21] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and 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.

[22] Himeno Benchmark, http://accc.riken.jp/en/supercom/himenobmt/, last accessed: January 2018.

[23] E. Wolters and M. Smith, "MOCFE-Bone: the 3D MOC Mini-Application for Exascale Research," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2013.

[24] TeaLeaf website, https://github.com/UoB-HPC/TeaLeaf.

[25] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[26] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[27] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[28] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable MPI profiling," 2005.

[29] D. Eberius, T. Patinyasakdikul, and G. Bosilca, "Using software-based performance counters to expose low-level open MPI performance information," in *Proceedings of the 24th European MPI Users' Group Meeting*. ACM, 2017, p. 7.