Plan B: Interruption of Ongoing MPI Operations to Support Failure Recovery

Aurelien Bouteiller
ICL, University of Tennessee
Knoxville, TN, USA
bouteill@icl.utk.edu

George Bosilca
ICL, University of Tennessee
Knoxville, TN, USA
bosilca@icl.utk.edu

Jack J. Dongarra
ICL, University of Tennessee
Oak Ridge National Lab.
Manchester University
Knoxville, TN, USA
dongarra@icl.utk.edu

ABSTRACT

Advanced failure recovery strategies in HPC system benefit tremendously from in-place failure recovery, in which the MPI infrastructure can survive process crashes and resume communication services. In this paper we present the rationale behind the specification, and an effective implementation of the Revoke MPI operation. The purpose of the Revoke operation is the propagation of failure knowledge, and the interruption of ongoing, pending communication, under the control of the user. We explain that the Revoke operation can be implemented with a reliable broadcast over the scalable and failure resilient Binomial Graph (BMG) overlay network. Evaluation at scale, on a Cray XC30 supercomputer, demonstrates that the Revoke operation has a small latency, and does not introduce system noise outside of failure recovery periods.

CCS Concepts

•**Computing methodologies** → *Distributed algorithms;* •**Computer systems organization** → **Reliability; Fault-tolerant network topologies;** •**Software and its engineering** → Software fault tolerance; Ultra-large-scale systems;

1. INTRODUCTION

As the number of components comprising HPC systems increases, probabilistic amplification entails that failures are becoming a common event in the lifecycle of an application. Currently deployed petascale machines, like Titan or the K-computer, experience approximately one failure every 10 hours [24], a situation which is expected to worsen with the introduction of exascale systems in the near future [2]. Coordinated Checkpoint/Restart (CR), either at the application or the system level, is currently the most commonly deployed mechanism to circumvent the disruptions caused by failures. It can be implemented without meaningful support for fault tolerance in the Message Passing Interface (MPI). However, models and analysis [16, 8] indicate that the status-quo is not sustainable, and either CR must drastically improve (with the help of in-place checkpointing [5, 22], for example), or alternative recovery strategies must be considered. The variety of prospective techniques is wide, and notably includes checkpoint-restart variations based on uncoordinated rollback recovery [9], replication [16], or algorithm based fault tolerance —where mathematical properties are leveraged to avoid checkpoints [13, 10]. A common feature required by most of these advanced failure recovery strategies is that, unlike historical rollback recovery where the entire application is interrupted and later restarted from a checkpoint, the application needs to continue operating despite processor failures, so that, whatever the recovery procedure, it can happen in-line and in-place. The User Level Failure Mitigation (ULFM) proposal [6] is an effort to define meaningful semantics for restoring MPI communication capabilities after a failure.

One of the most important features provided by the ULFM interface is the capability to interrupt ongoing MPI operations, in order for the application to stop the normal flow of processing, and regroup in a recovery code path that performs the application directed corrective actions. The ULFM API exposes that operation to the users, through a function named `MPIX_COMM_REVOKE`, so that applications can selectively trigger this interruption only when the recovery strategy is collective, and on the scope of the communication objects that need to be repaired. In this paper, we investigate an effective implementation of such a *Revoke* operation, based on the Binomial Graph topology [3].

The contribution of this paper is threefold: In Section 2, we present the rationale for non-uniform error reporting in MPI operations, and thereby infer the specification of an explicit failure propagation routine that can interrupt a failed communication plan; in Section 3 we lay down the requirements of the proposed Revoke operation in terms of a reliable broadcast with relaxed properties, expose that the BMG overlay broadcast has the desired resiliency while remaining scalable, and describe important implementation features of the BMG based Revoke operation; then in Section 4 we present the performance of the BMG based Revoke on a Cray supercomputer, which is, to our knowledge, the first practical evaluation of a reliable broadcast at such a large scale. We then discuss, in Section 5, how these contributions contrast with, and complement related works, before we conclude.

EuroMPI '15, September 21-23, 2015, Bordeaux , France
© 2015 ACM. ISBN 978-1-4503-3795-3/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2802658.2802668

2. NON-UNIFORM FAILURE KNOWLEDGE

This section discusses the rationale behind the proposed design that justifies the introduction of the Revoke operation. We take the perspective of the performance conscious MPI implementor, and analyze the unacceptable overhead resulting from requiring uniformity of failure knowledge. We then present the issues that arise when this requirement is dropped, and the modus-operandi of the Revoke interface to resolve them. The proposed design does indeed permit minimal overhead on failure free performance, as has been illustrated by the implementation presented in [7]. A more general presentation of the ULFM interface can be found in [6].

2.1 Failure Detection

Failure detection has proven to be a complex but crucial area of fault tolerance research. Although in the most adverse hypothesis of a completely asynchronous system, failures (even simple processes crash, as we consider here) are intractable in theory [17], the existence of an appropriate failure detector permits resolving most of the theoretical impossibilities [11]. However, requiring complete awareness (thus active monitoring) of failures of every process by every other process would generate an immense amount of system noise (from heartbeat messages injected into the network and the respective treatments on the computing resources to respond to them), and it is known that MPI communication performance is very sensitive to system noise [23]. Fortunately, processes that are not trying to communicate with a dead process do not need, a priori, to be aware of its failure, as their operations are with alive processors and therefore deadlock-free. As a consequence, failure detection in ULFM only requires to detect failures of processes that are direct partners in a communication operation.

2.2 Local versus Uniform Error Reporting

Another natural preconception is to consider that detection of failures at any rank results in MPI automatically altering the state of all communication objects in which the associated process appears (*i.e.* communicators, windows, etc.). In such a model, it is understood that the failure "damages" the communication object and renders it inappropriate for further communications. However, a complication is hidden in such an approach: the state of MPI communication objects is the aggregate state of individual views by each process of the distributed system. As failure awareness is not expected to be global, the implementation would then require internal and asynchronous propagation of failure detection, again, a process that is prone to introduce jitter. Furthermore, some recovery patterns (typical in PDE solvers [1], as an example) do not require advanced, nor collective, corrective actions and can continue between non-failed processes on the unmodified communication object. As a consequence, ULFM never automatically modifies the state of communication objects. Even if it contains failed processes, a communicator remains a valid communication object, until explicitly required. Therefore, error reporting is not intended to indicate that a process failed, but to indicate that an operation cannot deliver the normal semantic at the local rank: when a failure happened, but an MPI operation can proceed without disruption, it completes normally; when the failed process is supposed to participate in the result of the operation, it is obviously impossible for the operation to succeed, and an appropriate error is returned.

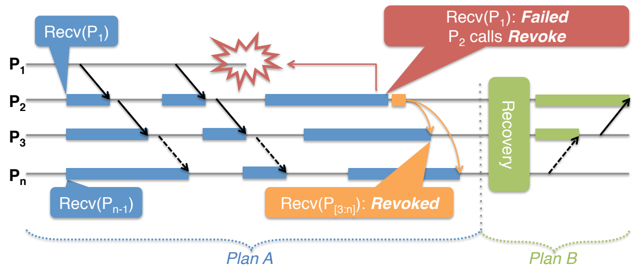


Figure 1: The transitive communication pattern in *plan A* must be interrupted before any process can switch to the recovery communication pattern *plan B*. By revoking the communication context, P_2 ensures that all possibly unmatched operations in *plan A*, which could provoke deadlocks, are interrupted.

Errors can then be captured by the application by setting the appropriate `MPI_ERRHANDLER`.

An additional criterion to consider is that some MPI operations are collective, or have a matching call at some other process (*e.g.* Send/Recv). Convenience would call for the same error be returned *uniformly* at all ranks that participated in the communication. This would easily permit tracking the global progress of the application (and then infer a consistent, synchronized recovery point). However, the performance consequences are dire, as it requires that every communication concludes with an agreement operation between its participants in order to determine the global success or failure of the communication, as viewed by each process. Such an operation cannot be possibly achieved in less than the cost of an AllReduce, even without accounting for the cost of actually tolerating failures during the operation, and would thus impose an enormous overhead on communication. In regard to the goal of maintaining an unchanged level of performance, it is clearly unacceptable to double, at best, the cost of all latency bound communication operations, especially when no failure has occurred. Furthermore, it is already customary for MPI operations to have a local only semantic, for example, when an `MPI_REDUCE` completes at a non-root process, there is no guarantee that the root has received the result of the collective operation yet. The semantic only specifies that when the operation completes, the local input buffer can be reused.

As a consequence, in ULFM, the reporting of errors has a local operation semantic: the local completion status (in error, or successfully) cannot be used to assume if the operation has failed or succeeded at other ranks. In many applications, this uncertainty is manageable, because the communication pattern is simple enough. In some cases, however, the communication pattern does not allow such flexibility, and the application thereby requires an operation to resolve that uncertainty, as described below.

2.3 Dependencies Between Processes

If the communication pattern is complex, the occurrence of failures has the potential to deeply disturb the application and prevent an effective recovery from being implemented. Consider the example in Figure 1: as long as no failure occurs, the processes are communicating in a point-to-point pattern (called *plan A*). Process P_k is waiting to receive a message from P_{k-1}, then sends a message to P_{k+1} (when such processes exist). Let's observe the effect of introducing a failure in *plan A*, and consider that P_1 has failed. As only P_2 communicates directly with P_1, other processes do not detect this condition, and only P_2 is informed of the failure of P_1. The situation at P_2 now raises a dilemma: P_3 waits on P_2, a non-failed process, therefore the operation must block until the matching send is posted at P_2; however, P_2 knows that P_1 has failed, and that the application should branch into its recovery procedure *plan B*; if P_2 were to switch abruptly to *plan B*, it would cease matching the receives P_3 posted following *plan A*. At this point, P_2 needs an effective way of interrupting operations that it does not intend to match anymore, otherwise, the application would reach a deadlock: the messages that P_3 to P_n are waiting for will never arrive. The proposed solution to resolve this scenario is that, before switching to *plan B*, the user code in P_2 calls `MPIX_COMM_REVOKE`, a new API which notifies all other processes in the communicator that a condition requiring recovery actions has been reached. Thanks to this flexibility, the cost associated with consistency in error reporting is paid only after an actual failure has happened, and only when necessary to the algorithm, and applications that do not need consistency, or in which the user can prove that the communication pattern remains safe, can enjoy better recovery performance.

3. THE REVOKE OPERATION

When a process of the application calls `MPIX_COMM_REVOKE` (similar operations exist for windows and files, we will, without loss of generality, reason in the case of communicators), all other alive processes in the communicator eventually receive a notification. The `MPIX_COMM_REVOKE` call has an effect on the entire scope of the communicator, without requiring a collective or matching call at any participant. Instead, the effect of the Revoke operation is observed at other processes during non-matching MPI communication calls: when receiving this notification, any communication on the communicator (ongoing or future) is interrupted and a special error code returned. Then, all surviving processes can safely enter the recovery procedure of the application, knowing that no alive process belonging to that communicator will deadlock as a result.

After a communicator has been revoked, its state is definitively altered and it can never be used again to communicate. This alteration is not to be seen as the (direct) consequence of a failure, but as the consequence of the user explicitly calling a specific operation on the communicator. In a sense, Revoking a communicator explicitly achieves the propagation of failure knowledge that has intentionally not been required, but is provided when the user deems necessary. Because the object is discarded definitively, any stale message matching the revoked object is appropriately ignored without modifications in the matching logic, and multiple processes may simultaneously Revoke the same communicator without fears of injecting delayed Revoke notifications, thereby interfering with post-recovery operations. In order to restore communication capacity, ULFM provides the repair function `MPIX_COMM_SHRINK`, which derives new, fresh communicators that do not risk intermixing with pre-failure operations or delayed notifications.

3.1 A Resilient, Asynchronous Broadcast

The revocation notification needs to be propagated to all alive processes in the specified communicator, even when new failures happen during the Revoke propagation. Therefore, it is in essence a *reliable broadcast*. Among the four defining qualities of a reliable broadcast usually considered in the literature (*Termination, Validity, Integrity, Agreement*) [19], the non-uniform variants of the properties are sufficient, and the integrity criteria can be relaxed in the context of the Revoke algorithm.

First, the agreement and validity properties ensure that if a process broadcasts a value v, all processes deliver v. In the uniform-agreement case, that property extends to failed processes: if a failed process had delivered the value, then it must be delivered at all correct processes. In the Revoke operation, if failures kill the initiator as well as all the already notified processes, the Revoke notification is indeed lost, and surviving processes may never receive the notification. However, either correct processes are not expecting messages from the set of dead processes, therefore no operation can deadlock, or at least a correct process is directly trying to exchange messages with a dead process and will detect its failure, which means that its blocking operations will complete in error, and leave the opportunity for the application to reissue the Revoke operation. In all cases, a non-uniform reliable broadcast is sufficient to ensure deadlock free operation. This is of practical significance, because the reliable broadcast respecting the uniform-agreement property requires that the system is free of *send-omission* failures (that is, a send has completed, but the receiver does not receive the message) [19]. In MPI, when a send operation completes, it does not mean that the receiver has delivered the message; the message may still be buffered on the sender process, and when that process is the victim of a crash failure, it may thereby simultaneously commit a send-omission failure. Ensuring that the network is free of send-omission failures requires the acknowledgement of sent messages, or additional rounds of message exchanges before delivering the reliable broadcast. As Revoke can be implemented with a non-uniformly agreeing reliable broadcast, that extra cost is spared.

Second, the integrity property states that a message is delivered once at most, and variants with additional ordering properties exist, like FIFO or causal ordering between the delivery of different broadcasts. In the case of a Revoke notification, the first Revoke message to reach the process has the effect of immutably altering the state of the communicator. Supplementary deliveries of Revoke messages for the same communicator have no effect. Similarly, if multiple initiators concurrently broadcast a Revoke notification on the same communicator, the order in which these notifications are delivered has no importance, as the final outcome is always a switch to an immutable revoked state. Therefore, we can retain a non-ordered, relaxed integrity reliable broadcast, in which we allow multiple out-of-order deliveries, but retain the reasonable assumption that Revoke messages do not appear out of "thin air". Then, as long as the algorithm still ensures the non-uniform agreement property, there are no opportunities for inconsistent views.

These simplified requirements are crucial for decreasing the cost of the Revoke operation, as the size of the messages and the number of message exchanges rounds can be drastically increased when one needs to implement an ordered, uniform reliable broadcast. Given the non-uniform agreement, the no-ordering, and loose integrity properties, in the

Revoke reliable broadcast, a process that receives its first Revoke message can perform a single round of emissions to all its neighbors, with a constant message size, and then deliver the Revoke notification immediately, without further verifications.

The last important aspect is the topology of the overlay network employed to perform the broadcast operation. In the reliable broadcast algorithm, when a process receives a broadcast message for the first time, it immediately broadcasts that same message to all its neighbors in the overlay graph. The agreement property can be guaranteed only when failures do not disconnect the overlay graph. In early prototype versions of the ULFM implementation, the reliable broadcast procedure employed a fully connected network (which guarantees that disconnected cliques never form). Obviously, this method scales poorly as, with the number or processes, the graph degree is linear, and the number of exchanged messages is quadratic. In practice, at scale, the large graph degree resulted in the application aborting due to resource exhaustion (too many open channels simultaneously, not enough memory for unexpected messages, etc.). Therefore, one needs to consider a more scalable overlay topology with a low graph degree that can yet maintain connectivity when nodes are suppressed.

3.2 Binomial Graph Overlay Topology

The Binomial Graph (BMG), introduced in [3], is a topology that features both opposing traits of a small degree, yet a strong resistance to the formation of disconnected cliques when nodes fail. A BMG is an undirected graph $G = (V, E)$, where the vertices V represent a set of processes, and the edges E are a set of links forming an overlay network between these processes. Each vertex $v \in V$ is given an unique identifier in $[0 \dots n-1]$, where $n = |V|$ (*i.e.* the rank of the process). For each vertex v, there is a link to a set of vertices $W = \{v \pm 1, v \pm 2, \dots, v \pm 2^k | 2^k \leq n\}$. Intuitively, a binomial graph can be seen as the union of all the binomial trees rooted at all vertices.

The BMG topology is proven to feature several desirable properties. It is a regular graph topology, in which all nodes have the same degree, even in graphs with unremarkable number of vertices (*e.g.* when $n \neq 2^i$, etc.). The degree, $\delta = 2 \times \lceil log_2 n \rceil$, is logarithmic with the number of nodes, therefore scalable. Meanwhile, it retains a small diameter and a small average distance (in number of hops) between any two nodes (also logarithmic). In addition, a binomial broadcast tree rooted at any node can be naturally extracted from a BMG. Such an extracted broadcast tree is symmetric in terms of performance, in the sense that the broadcast duration for a short message is $\lambda \times log_2 n$, where λ is the link latency, whatever the node selected as the root.

Last, the BMG topology has a high node-connectivity — the minimum number of nodes whose removal can result in disconnecting the network—, which is equal to the degree δ. As a consequence the BMG is $\delta - 1$ node fault tolerant in all cases (an optimal result for a graph of degree δ). The probability distribution for the formation of a disconnected graph when δ or more failures happen is very favorable (the failures have to strike a particular set of nodes, in a variant of the generalized birthday problem). Indeed, model evaluations have observed that when less than 50% of randomly distributed nodes have failed, the disconnection probability is well under 1% [4].

3.3 Implementation

The Revoke operation is implemented in the ULFM MPI library [7], which is a fork from Open MPI 1.6. Its algorithm is a non-uniform, non-ordered reliable broadcast spanning on a BMG overlay network. When a process invokes `MPIX_COMM_REVOKE`, first the communicator is locally marked as revoked, then the initiator process sends a message to all its neighbors in the BMG overlay. Revoke messages are sent using the low-level *Byte Transport Layer (BTL)* interface of Open MPI. The BTL layer is the springboard upon which the MPI communication operations are implemented, so reusing that infrastructure provides a portable access to all the high performance networks supported by Open MPI. Unlike MPI communications, a BTL communication has no matching receive. Instead, it is active message based: the sender tags outgoing messages, the reception of a message of a particular tag triggers the execution of a corresponding registered callback routine at the receiver, with the message payload as a parameter. The Revoke tag is distinct from the active message tags employed for MPI communications, which ensures a clear separation and avoids polluting the MPI matching logic with special cases for Revoke. With this low level interface, receptions do not need to be preposted and are always an unexpected event, and a message sent to a failed peer is silently dropped. As a consequence, failure detection is not forced and neighbors in the Revoke overlay topology do not need to be monitored. The message itself contains the communicator's context identifier and an epoch. Overall, a Revoke message is 24 bytes long, BTL protocol header included. When the active message is received, it triggers the Revoke forwarding callback. This callback is executed within the context of the Open MPI progress engine and has access to the internal structures of the implementation.

The callback first seeks the communicator associated with the context identifier. A technical difficulty arises here: 1) in Open MPI, context identifiers are actually indices in an array, and for performance reasons it is desirable to reuse lower indices as soon as the related communicator is freed, rather than increase the size of the communicator associative array; 2) `MPI_COMM_FREE` is not synchronizing. As a consequence, in some cases, a message may arrive after the associated communicator has been freed locally, and it is important to verify that the operation has no side effects on an unrelated communicator reusing that context identifier. In more details, imagine that, in plan A, P_r posts a reception from P_s; meanwhile, process P_k revokes the communicator, which interrupts the reception at P_r. Now, imagine that the delivery of the revoke notification is unusually slow at P_s, which then proceeds to send its message to P_r, unaware (yet) that it is following a revoked communication plan. As long as P_r does not free the communicator, the normal matching logic of MPI will correctly dispatch this stall message to the revoked communicator, where it will thereafter be appropriately discarded. However, when P_r frees the communicator, the context identifier becomes available for reuse, and if P_r creates a new communicator (as an example, a duplicate of `MPI_COMM_SELF`, which requires no communication), then that stall message could incorrectly be delivered in the newly created communicator. To avoid this caveat without impacting the MPI matching logic, yet still allow for the reuse of the communicator identifiers, the `MPI_COMM_FREE` function needs to become loosely synchronizing.

The `MPI_COMM_FREE` function is defined as a collective operation whose implementation is likely to be local, that is, it usually requires no communication. In order to minimize the performance impact, we designed a fault tolerant barrier that can progress in the background, so that it doesn't inflict a significant duration increase on the `MPI_COMM_FREE` call itself. The deallocation of the communicator then becomes lazy, when the application calls `MPI_COMM_FREE`, the communicator is marked for deallocation (and the user handle can be destroyed immediately), however, the internal representation of the communicator is deallocated only when it is safe, after the background barrier completes. Similarly to the Revoke operation, this barrier is implemented at the BTL level and essentially performs a binomial reduce-broadcast sequence. When a process receives the broadcast direction message, it can infer that every process invoked `MPI_COMM_FREE` on that communicator, hence all communication on the communicator completed[1] (either successfully, or in error when a participant died, or the revoked operation was interrupted).

However, Revoke notification messages are not posted under the control of the user, and therefore they are not completed before `MPI_COMM_FREE`. Thus, it is still possible that some continue to be delivered after the loosely synchronizing `MPI_COMM_FREE` has completed. In order to discriminate between different communicators using the same index, the Revoke message compounds the index with the epoch number, representing how many times this index has been allocated. This compound key is then used to perform the communicator lookup (in the case of Revoke messages only, normal MPI messages still employ the normal MPI matching with context identifiers only). If a communicator does not exist anymore (the message epoch is lower than the index epoch), the Revoke message is dropped; this is safe, as when the communicator doesn't exist anymore, the loosely synchronized `MPI_COMM_FREE` guarantees that it has been freed at every other process too. When the communicator with the correct epoch exists, there are two cases; 1) the communicator had already been revoked, then the callback drops the message and returns; 2) the communicator is not yet revoked, then it is revoked immediately and the Revoke message is broadcast to all neighbors.

When a communicator is revoked for the first time, the list of pending MPI requests is traversed to mark all requests on that communicator as completed in error. Their status is set to the special error code `MPIX_ERR_REVOKED`, pending RDMA operations are cancelled, and the memory registrations are withdrawn. In addition, the unexpected and matching queues of the communicator are also traversed to discard incoming message fragments.

4. EXPERIMENTAL EVALUATION

The experimental evaluation of the Revoke operation is conducted on the Darter platform, a Cray XC30 supercomputer hosted at the National Institute for Computational Science (NICS). Each of the 724 compute nodes features Two 2.6 GHz Intel 8-core XEON E5-2600 (Sandy Bridge) Series processors, and is connected via a Cray Aries router with a bandwidth of 8GB/sec. We employ the ULFM Open MPI fork, with the "`tuned`" collective communication module, the "`uGNI`" transport module between nodes, and the "`SM`" transport module for inter-core, shared-memory communication.

[1]Freeing a communicator that still has pending messages is standard compliant, but strongly discouraged: as the communicator is not available anymore, if the operation must report an error, it triggers the default `MPI_ERRORS_ABORT` error handler, which effectively makes such an application inherently non-fault tolerant.

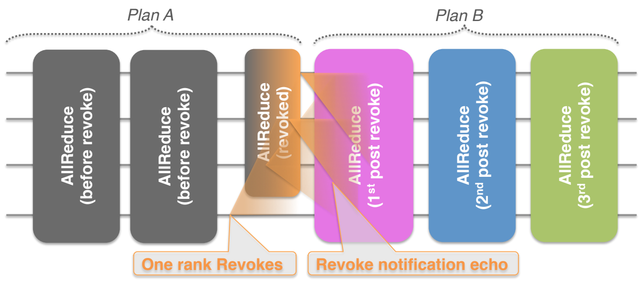

Plan A
Plan B
AllReduce (before revoke)
AllReduce (before revoke)
AllReduce (revoked)
AllReduce (1st post revoke)
AllReduce (2nd post revoke)
AllReduce (3rd post revoke)
One rank Revokes
Revoke notification echo

Figure 2: The Revoke Benchmark: a process revokes plan A during a collective communication. As soon as plan A is interrupted, every process switches to plan B, a similar communication plan, with the same collective operation, but on a distinct, duplicate communicator.

4.1 Benchmark

Because of its asymmetrical nature, the impact of the Revoke call cannot be measured directly. At the initiator, the call only starts a non-synchronizing wave of token circulation, and measuring the very short duration of the call is not representative of the actual time required for the Revoke call to operate at all target processes. Measuring the time needed for a particular operation to be interrupted gives a better estimate of the propagation time of a Revoke notification. However, the overall impact remains underestimated if one doesn't account for the fact that even after all processes have successfully delivered a Revoke notification, the reliable broadcast algorithm continues to emit and handle Revoke messages in the background for some time.

The benchmark we designed measures both the duration and the perturbation generated by the progress of a Revoke operation on the network. The benchmark comprises two communication plans (illustrated in Figure 2). Plan A is a loop that performs a given collective operation on a communicator that spans on all available processes (`commA`). At some iteration, an initiator process does not match the collective operation, but, instead, invokes `MPIX_COMM_REVOKE` on `commA`, which effectively ends plan A. Plan B is a similar loop performing the same collective operation in a duplicate communicator (`commB`) that spans on the same processes as `commA`. However, because it is a distinct communicator, operations on `commB` do not match operations on `commA`; in particular, the Revoke operation on `commA` has no effect on the semantic of collective operations posted in `commB`, all ranks need to match the operation, and it completes normally. We consider that the duration of a particular collective operation is the maximum latency across all ranks, and we then compute the average over 2,000 repetitions of the benchmark. We report the latency of operations on `commA` before it is revoked, and when one rank does not match the operation and instead invokes `MPIX_COMM_REVOKE`; this *Revoked collective communication* gives an estimate of the Revoke propagation time. Last, we report the latency of the first op-

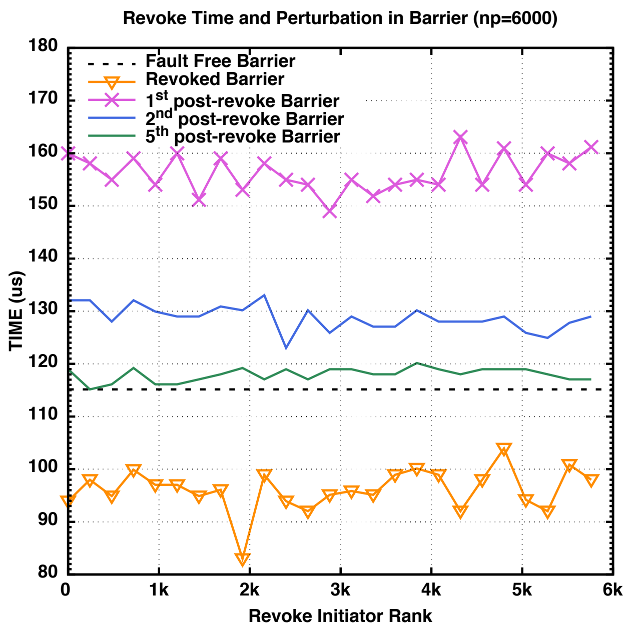


Figure 3: Revoke cost in Barrier depending on the initiator rank calling `MPIX_COMM_REVOKE` (6,000 processes).

erations posted on `commB` until the typical latency becomes similar to pre-Revoke operations on `commA`.

The collective communication patterns are inherited, without modification, from the Open MPI non-fault tolerant "`tuned`" module. The Cray optimized MPI can, in some instances, achieve higher performance. For the purpose of our evaluation, the `tuned` generic implementation, based on MPI point-to-point message exchanges, is representative of users' communication patterns commonly found in typical, portable HPC applications.

4.2 Initiator Location and Revoke Impact

Figure 3 presents the latency of Barriers on 6,000 processes, depending on the rank of the initiator process that invokes the `MPIX_COMM_REVOKE` operation. Thanks to the symmetric nature of the BMG topology, the Revoked Barrier latency is stable and independent of the initiator rank. One can note that the time to complete a Revoked Barrier is smaller than the time to complete a normal Barrier. The normal Barrier has a strong synchronizing semantic: the operation cannot complete before every process has entered the barrier. A Revoked Barrier doesn't enforce that synchronization anymore and it can complete locally before some processes have participated. Instead, the latency of the Revoked operation denotes the time taken by the Revoke resilient broadcast to reach every rank for the first time; this propagation latency is similar to the cost of a small message Broadcast.

However, as stated before, when the Revoke notification has been delivered to every rank, the reliable broadcast has not terminated yet, and some Revoke token messages have been freshly injected in the network (at the minimum, the $2log_2(n)$ messages injected by the last rank to deliver the Revoke notification are still circulating in the network). As a consequence, the performance of the first post-Revoke collective operation sustains some performance degradation resulting from the network jitter associated with the circulation of these tokens. This performance degradation is moderate, with the latency approximately doubling. The jitter noise is equally spread on the BMG topology, therefore, the increased latency of the first (and the much reduced impact on the 2^{nd} to 5^{th}) Barrier is also independent of the initiators' rank.

Although after the first post-Revoke Barrier, no new Revoke tokens are injected (when the first Barrier of plan B completes, a Revoke token has been delivered at every rank, thus every rank has already injected its reliable broadcast tokens), the absorption of delayed tokens and the lost synchrony resulting from the initial jitter combine to impact slightly the Barrier performance. After the fifth Barrier (approximately $700\mu s$), the application is fully resynchronized, and the Revoke reliable broadcast has terminated, therefore leaving the application free from observable jitter.

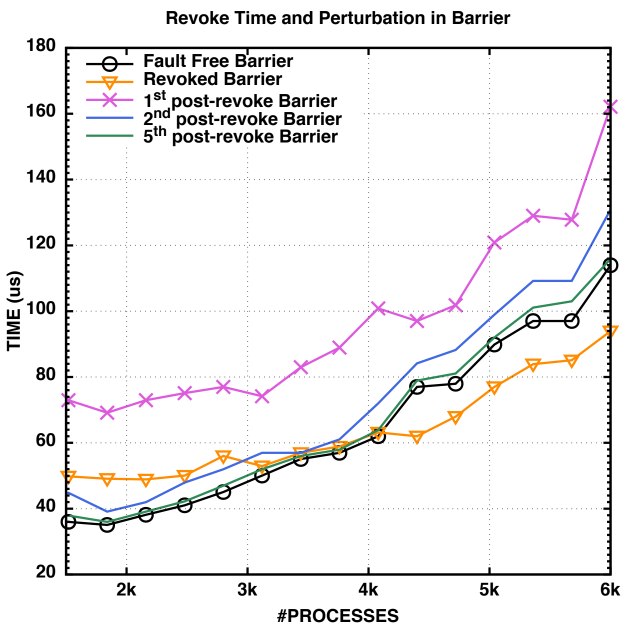
4.3 Scalability

Figure 4 presents the scalability of the Barrier (left) and AllReduce (right) collective communications in the Revoke benchmark. The first observation is that the performance of post-Revoke collective communications follows the same scalability trend as the pre-Revoke operations, even those impacted by jitter. In the case of the AllReduce collective communication, aside from the 1^{st} post-Revoke AllReduce communication, which still exhibit a moderate overhead from jitter, the 2^{nd} post-Revoke AllReduce is only mildly impacted and the 3^{rd} AllReduce exhibit no significant difference from the failure free case, illustrating that the jitter introduced by the reliable broadcast algorithm has a lesser impact on this communication pattern. When the number of processes increases, the impact of jitter —the difference between the failure-free and the 1^{st} post-Revoke operation— is almost constant (or slightly decreasing). If this trend were to continue at larger scales, the impact of jitter could become asymptotically negligible.

Last, while the implementations of the "`tuned`" collective operations differ in performance trends on this Cray machine (for reasons outside of the scope of this work, but rooting in the internal collective algorithm selection logic being tuned for the Infiniband network), the performance of the revoked operation is similar in both cases, illustrating that, as long as MPI progress is triggered, the propagation latency of the BMG reliable broadcast is independent from the communication plan being revoked.

4.4 AllReduce and Message Size

Figure 5 presents the latency of the AllReduce collective communication when the message size varies. Focusing first on the cost of the Revoked AllReduce operation, one can observe that the duration of the operation remains independent of the message size until the message size increases to 1MB or more. As the Revoked operation is interrupted before exchanging the entire communication volume, this behavior is expected. For larger message sizes, however, the delivery of the Revoke notification may be delayed by the granularity of the ongoing reduction computation; as these computations are progressing, the MPI progress engine is managing them with maximum priority, and thus does not consider incoming fragments for that time duration. As soon as one



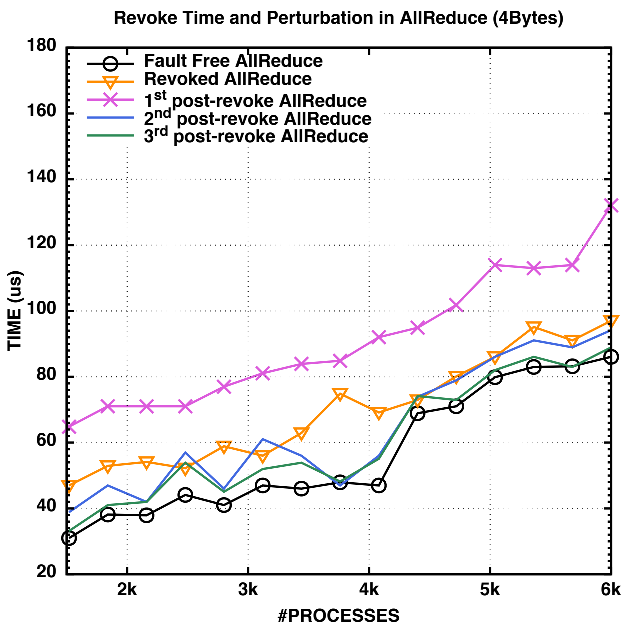


Figure 4: Impact of Revoke on collective communication patterns, depending on the number of processes.

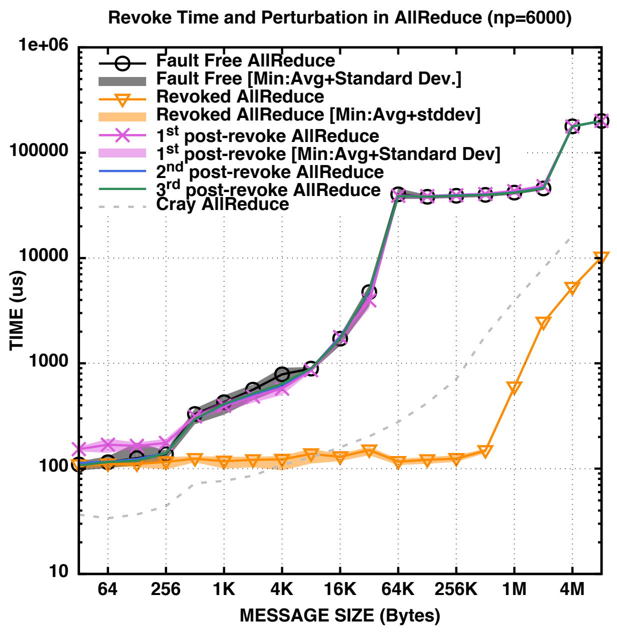


Figure 5: Impact of Revoke on AllReduce, depending on the message size (6,000 processes).

of these computation completes, the Revoke notification is delivered, supplementary computation on pipelined blocks are discarded, and further data transfers cancelled.

For post-Revoke AllReduce operations, the impact of jitter on performance is visible only for small message operations. As soon as the message size is larger than 512 bytes, the initial performance difference is absorbed in the cost of the operation itself.

Interestingly, the standard deviation (between 2,000 runs) for both Revoked and jitter-disturbed AllReduce operations remains low, and of the same magnitude as the natural, failure-free standard deviation of the operation.

5. RELATED WORKS

Any production distributed system needs a failure detection and propagation mechanism, if only for the platform job manager to be able to cleanup a failed application and release the compute resources. This approach pairs well with legacy coordinated checkpoint/restart, which is most applicable to machines with a low failure rate, thus the centralized, slow handling of failure in the job manager is acceptable in that case.

However, in a distributed infrastructure that aims to effectively support machines with moderate to high failure rates, the quick recovery of the communication infrastructure itself is crucial. MapReduce [14] is often lauded for its capability to gracefully tolerate failures; the runtime essentially decomposes the workload in a master-worker problem, in which the handling of a failure is entirely localized at the master process currently managing the failed worker, thereby enabling the seemingly instantaneous recovery of the infrastructure. Although this operational mode is shared by some HPC applications, many are tightly coupled, and require the reestablishment of a consistent environment after a failure, and thus failure knowledge propagation.

In FT-MPI [15], when the infrastructure detects a failure, it repairs the state of MPI internally, according to the preselected recovery mode. All nodes are notified of failures in unison and collectively re-exit from the `MPI_INIT` function, thereby requiring an implicit global propagation of failure knowledge, inflexibly invoked from within the MPI library. In PVM [18], while the triggering of the failure propagation remains implicit, the user code subscribes explicitly to failure notifications, which could —in theory— restrict the scope of the failure knowledge propagation to self-declared interested parties. In GVR [12], users explicitly subscribe to, and publish failure conditions. A local failure detection can then be escalated by any node, by publishing a global failure condition.

Compared to these interfaces, the Revoke operation is explicit at the publisher, but its subscriber scope is implicitly derived from the communicator (or window, file) object on which it operates. The pre-established subscriber set is beneficial because it offers a static naming of processes and helps build an efficient diffusion topology. For the propagation algorithm itself, many approaches have been employed in the past. Gossip [21] algorithms, or Chord/Tapestry-like topologies [25] have been considered to disseminate (or gather) knowledge in peer-to-peer systems. These approaches have to operate under a set of adverse constraints, where the network topology and the unique process mapping have to be established online. The reliable BMG broadcast exchanges some of this flexibility for improved propagation latency, and much less system noise (in effect, the total absence of system noise during failure-free periods), which makes it a better match for the Revoke operation.

In addition, a unique property of the Revoke operation is to provide a clear specification of what is to happen to in-flight operations that are pending on the corresponding communication object when the error callback is triggered at a subscriber.

6. CONCLUSIONS

Without a consistent view of the failed processes, some tightly coupled, or transitively dependent communication patterns prevent legitimate applications from deploying a meaningful recovery strategy. When a process detects a failure, and thus switches to a recovery communication plan, it takes the risk of leaving in a deadlock unmatched operations at processes that are oblivious of the failure, and therefore continue to follow the original communication plan. Unfortunately, providing a consistent common view of failure knowledge at all times implies unnecessary, severe overheads that should be reserved for cases where such knowledge is imperative, and not imposed on all scenarios, especially the failure-free case. Therefore, the propagation of the failure knowledge must be provided as a separate construct that can be triggered explicitly by the application, on a voluntary base, and only when necessary after some application process effectively observed a failure.

We introduce the Revoke operation to enable interrupting a failed communication plan, and giving applications the opportunity to regroup into a new, different communication plan, as needed by their recovery strategy. An effective implementation of the Revoke operation depends upon a scalable, reliable broadcast algorithm, for which we have delineated relaxed theoretical requirements, and have proposed to deploy it over a BMG overlay network. The BMG topology features both a low graph degree, a requirement for scalability, yet a strong resistance to the formation of partitioned cliques that threaten the correctness of the reliable broadcast.

We implemented and evaluated a Revoke reliable broadcast based on the BMG topology, and demonstrated that it can effectively interrupt typical MPI communication plans (as featured in commonly employed collective communication patterns) with a low latency and without incurring long-lasting jitter. Experiments outline that after a short period of time, of the same order of magnitude as a Broadcast communication, the recovery communication plan can start, and after a couple more Broadcast latencies, the post-Revoke performance of this communication plan is free of jitter and returns to the nominal, failure free latency.

The performance and scalability improvements provided by the BMG reliable broadcast, which are demonstrated at scale for the first time here, also have application beyond the Revoke operation, and with minor adaptations can be leveraged in other HPC contexts where the failure resistant dissemination of information is essential, such as the interruption of failed transactions in transactional fault tolerance [20], or the propagation of failure conditions in PGAS programming models [12].

7. ACKNOWLEDGMENTS

This work is partially supported by the CREST project of the Japan Science and Technology Agency (JST), and by NSF award #1339820.

8. REFERENCES

[1] M. M. Ali, J. Southern, P. E. Strazdins, and B. Harding. Application level fault recovery: Using fault-tolerant Open MPI in a PDE solver. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 1169–1178. IEEE, 2014.

[2] S. Amarasinghe and et al. Exascale Programming Challenges. In *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Jul 2011.

[3] T. Angskun, G. Bosilca, and J. Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In I. Stojmenovic, R. Thulasiram, L. Yang, W. Jia, M. Guo, and R. de Mello, editors, *Parallel and Distributed Processing and Applications*, volume 4742 of *Lecture Notes in Computer Science*, pages 471–482. Springer Berlin Heidelberg, 2007.

[4] T. Angskun, G. Bosilca, G. Fagg, J. Pjesivac-Grbovic, and J. Dongarra. Reliability analysis of self-healing network using discrete-event simulation. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 437–444, May 2007.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Int. Conf. High Performance Computing, Networking, Storage and Analysis SC'11*, 2011.

[6] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI

communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[7] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of user-level failure mitigation support in MPI. *Computing*, 95(12):1171–1184, Dec. 2013.

[8] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[9] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency and Computation: Practice and Experience*, 25(4):572–585, 2013.

[10] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Trans. Parallel Comput.*, 1(2):10:1–10:28, Feb. 2015.

[11] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), Mar. 1996.

[12] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber, J. Hammond, J. Dinan, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M. Hoemmen, M. Heroux, K. Teranishi, and A. Siegel. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *Journal of Computational Science*, (0):–, 2015.

[13] T. Davies, C. Karlsson, H. Liu, C. Ding, , and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*. ACM, 2011.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[15] G. Fagg and J. Dongarra. FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *7th Euro PVM/MPI User's Group Meeting2000*, volume 1908 / 2000, Balatonfured, Hungary, september 2000. Springer-Verlag Heidelberg.

[16] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

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

[18] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8, 1996.

[19] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, chapter 5, pages 97–145. ACM/Addison-Wesley, 1993.

[20] A. Hassani, A. Skjellum, and R. Brightwell. Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 750–755, Washington, DC, USA, 2014. IEEE Computer Society.

[21] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC'00, pages 253–267, London, UK, 2000. Springer-Verlag.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov. 2010.

[23] F. Petrini, E. Frachtenberg, A. Hoisie, and S. Coll. Performance Evaluation of the Quadrics Interconnection Network. *Cluster Computing*, 6(2):125–142, Apr. 2003.

[24] B. Schroeder and G. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, pages 12–22. IOP Publishing, 2007.

[25] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.