International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Special Issue Article

Post-failure recovery of MPI communication capability: Design and rationale

The International Journal of High Performance Computing Applications
27(3) 244–254
© The Author(s) 2012
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342013488238
hpc.sagepub.com
SAGE

Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca and Jack Dongarra

Abstract
As supercomputers are entering an era of massive parallelism where the frequency of faults is increasing, the MPI Standard remains distressingly vague on the consequence of failures on MPI communications. Advanced fault-tolerance techniques have the potential to prevent full-scale application restart and therefore lower the cost incurred for each failure, but they demand from MPI the capability to detect failures and resume communications afterward. In this paper, we present a set of extensions to MPI that allow communication capabilities to be restored, while maintaining the extreme level of performance to which MPI users have become accustomed. The motivation behind the design choices are weighted against alternatives, a task that requires simultaneously considering MPI from the viewpoint of both the user and the implementor. The usability of the interfaces for expressing advanced recovery techniques is then discussed, including the difficult issue of enabling separate software layers to coordinate their recovery.

Keywords
Fault tolerance, message passing interface, user-level failure mitigation

1. Introduction

Innovation in science and engineering strongly depends on the pervasive use of computer-assisted design and simulation, thereby demanding breakthrough computing capabilities. In the last decade, supercomputers have relied on increasing the number of processors to deliver unrivaled performance. The rationale behind this development is, essentially, driven by the lower operational cost of designs featuring a large number of low-power processors (Bright et al., 2005). According to current projections in processor, memory and interconnect technologies, and ultimately the thermal limitations of semiconductors, this trend is expected to continue into the foreseeable future (Dongarra et al., 2011). An unfortunate consequence of harnessing such a large amount of individual components is the resulting aggregate unreliability. As system size increases exponentially over the years, the improvements in component manufacture are outpaced, and long-running applications spanning the entire system experience increasing disruption from failures (Schroeder and Gibson, 2007; Cappello, 2009).

Message passing, and in particular the Message Passing Interface (MPI) (The MPI Forum, 2012), is the prevailing approach for developing parallel applications on massive-scale high-performance computing (HPC) systems. Historically, many MPI applications have relied on rollback recovery to recover from failures, a strategy that can be achieved without support from the MPI library. However, recent studies outline that, in light of the expected mean time between failures (MTBF) of exascale HPC systems and beyond (Dongarra et al., 2011), checkpoint–restart-based rollback recovery could underperform to the point where replication would become a compelling option (Ferreira et al., 2011). The literature is rich in alternative recovery strategies permitting better performance in a volatile, high-failure-rate environment. The variety of techniques employed is very wide, and notably include checkpoint–restart variations based on uncoordinated rollback recovery (Bouteiller et al., 2011), replication (Ferreira et al., 2011), algorithm-based fault tolerance where mathematical properties are leveraged to avoid checkpoints (Davies et al., 2011; Du et al., 2012), etc. A common feature found in most of these advanced failure recovery strategies is that,

Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA

Corresponding author:
George Bosilca, Innovative Computing Laboratory, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, TN 37996-3450, USA.
Email: bosilca@icl.utk.edu

unlike historical rollback recovery where the entire application is interrupted and later restarted from a checkpoint, the application is expected to continue operating despite processor failures, thereby reducing the incurred I/O, downtime and computation loss overheads. However, the MPI Standard does not define a precise behavior for MPI implementations when disrupted by failures. As a consequence, the deployment of advanced fault-tolerance techniques is challenging, taking a strain on software development productivity in many applied science communities, and fault tolerant applications suffer from the lack of portability of *ad hoc* solutions.

Several issues prevented the standardization of recovery behavior by the MPI Standard. Most prominently, the diversity of the available recovery strategies is, in itself, problematic: there does not appear to be a single best practice, but a complex ecosystem of techniques that apply best to their niche of applications. The second issue is that, without a careful, conservative design, fault-tolerance additions generally take an excruciating toll on bare communication performance. Many MPI implementors, system vendors and users are unwilling to suffer this overhead, an attitude further reinforced by the aforementioned diversity of fault tolerance techniques which results in costly additions being best suited for somebody else's problem.

In this paper, we describe a set of extended MPI routines and definitions called user-level failure mitigation (ULFM), that permit MPI applications to continue communicating across failures, while avoiding the two issues described above. The main contributions of this paper are (1) to identify a minimal set of generic requirements from MPI that enable continued operations across failures; (2) to propose a set of semantics that limit the complexity for MPI implementors, a feature that, beyond comfort, is paramount for maintaining extreme communication performance; (3) expose the rationale for the design choices and discuss the consequences of alternative approaches; (4) illustrate how high-level fault-tolerance techniques can benefit from the proposed constructs; (5) discuss how these semantics and constructs can be effectively used in production codes that intermix multiple layers of libraries from distinct providers.

The rest of this paper is organized as follows: Section 2 provides a background survey of fault-tolerance techniques and identifies common requirements, Section 3 gives an overview of the goals of providing fault tolerance in the MPI Standard, Section 4 describes the new constructs introduced by ULFM, Section 5 explores some of the rationale behind the ULFM design, Section 6 gives an overview of some possible compositions of fault-tolerance techniques on top of ULFM , Section 7 describes some of the previous work with integrating fault tolerance within MPI, before we conclude and look beyond the scope of MPI in Section 8.

2. Background

In this work, we consider the effect of fail-stop failures (that is, when a processor crashes and stops responding completely). Network failures are equally important to tolerate, but are generally handled at the link protocol level, thereby relieving MPI programs from experiencing their effect. Silent errors that damage the dataset of the application (memory corruption) without hindering the capacity to deliver messages (or resulting in a crash), are the sole responsibility of the application to correct. The survey by Cappello (2009) provides an extensive summary of fail-stop recovery techniques available in the literature. Since the focus of this work is to design an extension to the MPI runtime to enable effective deployment of advanced fault tolerance techniques, it is critical to understand the specificities, issues, common features and opportunities offered by this wide range of recovery techniques.

2.1. Checkpoint–restart (with coordinated rollback)

Rollback recovery is based on the intuitive procedure of restarting the failed application each time it is impacted by a failure. In order to diminish the amount of lost computation, the progress of the application is periodically saved by taking checkpoints. Should a failure happen, instead of restarting the application from the beginning it will be restarted from the last saved state, a more advanced state toward the application completion. In a parallel application, the matter is complicated by the exchange of messages: if the message initiation at the sender and its delivery at the receiver cross the line formed by the state of processes when reloaded from a checkpoint, the state of the application may be inconsistent, and the recovery impossible. The traditional approach is to construct a set of *coordinated checkpoints* that eliminates such messages completely, so that the checkpoint set is consistent (Chandy and Lamport, 1985). However, in such a case, the consistent recovery point is guaranteed only if the entire application is restarted from the checkpoint set.

Many MPI libraries provide coordinated checkpointing automatically, without application knowledge or involvement (Buntinas et al., 2008; Hursey et al., 2007). Because of the use of system-based checkpoint routines, these libraries have to be internally modified to remove the MPI state from the checkpoints. However, these modifications do not alter the interface presented to users and the performance hit on communication routines is usually insignificant. More generally, coordinated rollback recovery has been widely deployed by message-passing applications without any specific requirements from the MPI implementation. The program code flow is designed so that checkpoints are taken at points when no messages have been injected into MPI (hence, the network is empty and the checkpoint set consistent) (Silva and Silva, 1998).

2.2. Checkpoint–restart (with partial rollback)

In checkpoint–restart with partial rollback recovery, processes that have not been damaged by a failure are kept alive and can continue computing as long as they do not depend on a message from a failed process. To permit independent recovery of processes restarted from a checkpoint, a procedure called message logging (Alvisi and Marzullo, 1995; Elnozahy et al., 2002) stores supplementary information every time communications are involved. The message log is then used to direct the recovery of restarted processes toward a state that is consistent with the global state of processes that continued without restart. Recent advances in message logging (Bouteiller et al., 2010; Esteban Meneses and Kalé, 2010; Guermouche et al., 2012; Bouteiller et al., 2011) have demonstrated that this approach can deliver a compelling level of performance and may exceed the performance of coordinated rollback recovery.

2.3. Replication

Replication (Ferreira et al., 2011) is the idea that in order to provide fault tolerance for an application, rather than changing the application to incorporate fault-tolerance techniques or spend time writing checkpoints to disk and then performing full-system restarts, an application can execute multiple concurrent copies of itself simultaneously. In most variations, the replicates need to remain strongly synchronized, and messages' delivery are effectively atomic commits to multiple targets. As long as one of the replicates is still alive, no data loss has happened and the application can continue. New clones of failed processes can be recreated on the fly to ensure continued protection from further failures. While replication has a large overhead from duplicating computation and requiring heavy synchronization on message deliveries, it has been shown to provide a higher level of efficiency than checkpoint/restart under the extreme pressure of numerous, frequent failures.

2.4. Migration

Process migration (Chakravorty et al., 2006; Wang et al., 2008) is a form of fault tolerance which combines advanced, proactive failure detectors with some other form of fault tolerance, often checkpoint/restart. To reduce the increasing overhead of other forms of fault tolerance at scale, process migration detects that a failure is likely to occur at a particular process and moves it (or replicates it) to a node in the system less likely to fail. Migration requires accurate failure predictors to be useful, but when successful, it can reduce the overhead of other fault tolerance mechanisms significantly.

2.5. Transactions

Transactional-based computation can be seen as a form of speculative progress with lightweight checkpoints. The basic idea is that the algorithm is divided into blocks of code. Each block is concluded with a construct that decides the status of all communication operations which occurred within the block, as opposed to checking the status of each communication operation as it occurs (Skjellum, 2012). If the construct determines that a process failure had occurred in the preceding block, it allows the application to return to the status before the beginning of the block, giving it the opportunity to execute the block again (after replacing the failed process).

2.6. Algorithm-based fault tolerance

Algorithm-based fault tolerance (Davies et al., 2011; Du et al., 2012) is a family of recovery techniques based on algorithmic properties of the application. In some naturally fault-tolerant applications, when a failure occurs, the application can simply continue while ignoring the lost processes (typical of master–slave applications). In other cases, the application uses intricate knowledge of the structure of the computation to maintain supplementary, redundant data, that is updated algorithmically and forms a recovery dataset that does not rely on checkpoints. Although generally exhibiting excellent performance and resiliency, algorithm-based fault tolerance requires that the algorithm is innately able to incorporate fault tolerance and therefore might be a less generalist approach.

3. Design goals

After evaluating the strengths and weaknesses of the previous efforts toward fault tolerance both within MPI and with other models, we converged on four overarching goals for ULFM. More specifics on the design, rationale and generally how ULFM meets these goals can be found in Sections 4 and 5.

Flexibility in fault response is paramount: not all applications have identical requirements. In the simple case of a Monte Carlo master–worker application that can continue computations despite failures, the application should not have to pay for the cost of any recovery actions; in contrast, consistency restoration interfaces must be available for applications that need to restore a global context (a typical case for applications with collective communications). As a consequence, and in sharp contrast with previous approaches (see Section 7), we believe that MPI should not attempt to define the failure recovery model or to repair applications. It should inform applications of specific conditions that prevent the successful delivery of messages, and provide constructs and definitions that permit applications to restore MPI objects and communication functionalities. Such constructs must be sufficient to express advanced high-level abstractions (without replacing them), such as transactional fault tolerance, uncoordinated checkpoint/restart, and programming language extensions. The failure recovery strategies can then be featured by independent portable packages that provide tailored, problem specific recovery techniques and drive the recovery of MPI on behalf of the applications.

Resiliency refers to the ability of the MPI application not only to survive failures, but also to recover into a consistent state from which the execution can be resumed. One of the most strenuous challenges is to ensure that no MPI operation stalls as a consequences of a failure, for fault tolerance is impossible if the application cannot regain full control of the execution. An error must be raised when a failure prevents a communication from completing. However, we propose that such a notice indicates only the local status of the operation, and does not permit inferring whether the associated failure has impacted MPI operations at other ranks. This design choice avoids expensive consensus synchronizations from obtruding into MPI routines, but leaves open the danger of some processes proceeding unaware of the failure. Therefore, supplementary constructs must be sparingly employed in the application code to let processes which have received an error resolve their divergences.

Productivity and the ability to handle the large number of legacy codes already deployed in production is another key feature. Backward compatibility (i.e. supporting unchanged non fault tolerant applications) and incremental migration are necessary. A fault-tolerant API should be easy to understand and use in common scenarios, as complex tools have a steep learning curve and a slow adoption rate by the targeted communities. To this end, the number of newly proposed constructs must be small, and have clear and well-defined semantics that are familiar to users.

Performance impact outside of recovery periods should be minimal. Failure protection actions within the implementation must be outside the performance critical path, and recovery actions triggered by the application only when necessary. As most functions are left unmodified (as an example, the implementation of collective operations), they continue to deliver the extraordinary performance resulting from years of careful optimization. Overheads are tolerated only as a consequence of actual failures.

4. ULFM constructs

ULFM was proposed as an extension to the MPI Forum[1] to introduce fault-tolerance constructs in the MPI standard. It is designed according to the criterion identified in the previous section: to be the minimal interface necessary to restore the complete MPI capability to transport messages after failures. As requested by our flexibility goal, it does not attempt to define a specific application recovery strategy. Instead, it defines the set of functions that can be used by applications (or libraries and languages that provide high-level fault-tolerance abstractions) to repair the state of MPI. In this section, we summarize the new definitions and functions added by ULFM ; the rationale behind these design choices will be discussed in Section 5.

4.1. Failure reporting

Failures are reported on a per-operation basis, and indicate essentially that the operation could not be carried out successfully because a failure occurred on one of the processes involved in the operation. For performance reasons, not all failures need to be propagated, in particular, processes that do not communicate with the failed process are not expected to detect its demise. Similarly, during a collective communication, some processes may detect the failure, while some other may consider that the operation was successful; a particularity that we name non-uniform error reporting. Let's imagine a broadcast communication using a tree-based topology. The processes that are high in the tree topology, close to the root, complete the broadcast earlier than the leaves. Consequently, these processes may report the successful completion of the broadcast, before the failure disrupts the communication, or even before the failure happens, while processes below a failed process cannot deliver the message and have to report an error.

The first new construct, `MPI_COMM_REVOKE`, is the most crucial and complex, and is intended to resolve the issues resulting from non-uniform error reporting. As seen above, if non-uniform error reporting is possible, the view of processes, and accordingly the actions that they will undergo in the future, may diverge. Processes that have detected the failure may need to initiate a recovery procedure, but they have the conflicting need to match pending operations that have been initiated by processes that have proceeded unaware of the failure, as otherwise these may deadlock while waiting for their operation to complete. When such a situation is possible, according to the communication pattern of the application, processes that have detected that recovery action is needed and intend to interrupt following the normal flow of communication operations can release other processes by explicitly calling the `MPI_COMM_REVOKE` function on the communication object. Like many other MPI constructs `MPI_COMM_REVOKE` is a collective operation over the associated communicator. However, unlike any other collective MPI constructs it does not require a symmetric call on all processes, a single processes in the communicator calling the revoke operation ensure the communicator will be eventually revoked. In other words it has a behavior similar to `MPI_ABORT` with the exception that it does not abort processes, instead it terminate all ongoing operations on the communicator and mark the communicator as improper for future communications.

As an example, in Figure 1, four processes are communicating in a point-to-point pattern. Process 2 is waiting to receive a message from process 3, which is waiting to receive a message from process 0, itself waiting to receive a message from process 1. In the meantime, process 1 has failed, but this condition is detected only by process 0, as other processes do not communicate with process 1 directly. At this point, without a new construct, the algorithm would reach a deadlock: the messages that processes 2 and 3 are waiting for will never arrive because process 0 has branched to enter recovery. To resolve this scenario, before switching to the recovery procedure, process 0 calls `MPI_COMM_REVOKE`, which notifies all other processes in the communicator that a condition requiring recovery actions has been reached. When receiving this notification, any communication on the communicator (ongoing or future) is interrupted and a special error code returned. From this point, any

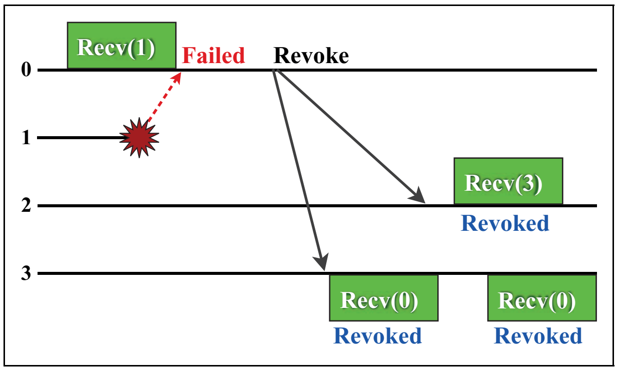

Figure 1. An example of a scenario where MPI_COMM_REVOKE is necessary to resolve a potential deadlock in the communication pattern.

operation (point-to-point or collective) on the communicator returns that same error code, and the communicator becomes effectively unusable for any purpose. Then, all surviving processes can safely enter the recovery procedure of the application, knowing that no alive process belonging to that communicator can deadlock on a communication that will not happen.

4.2. Rebuilding communicators

The next construct provides a recovery mechanism: `MPI_COMM_SHRINK`. Although the state of a communicator is left unchanged by process failures, and point-to-point operations between non-failed processes are still functional, it is to be expected that most collective communication will always raise an error, as they involve all processes in the communicator. Therefore, to restore full communication capacity, MPI communicators objects must be repaired. The `MPI_COMM_SHRINK` function create a new functional communicator based on an existing, revoked communicator containing failed processes. It does this by creating a duplicate communicator (in the sense of `MPI_COMM_DUP`) but omitting any processes which are agreed to have failed by all remaining processes in the shrinking communicator. If there are new process failures which are discovered during the shrink operation, these failures are absorbed as part of the operation.

4.3. Continue without revoke

Revoking a communicator is an effective but heavy-handed recovery strategy, as no further communication can happen on the revoked communicator, and a new working communicator can only be created by calling `MPI_COMM_SHRINK`. Depending on the application communication pattern, the occurrence of a failure may never result in a deadlock (an opportunity that is impossible to detect at the implementation level, but that may be known by the programmer, typically in a domain decomposition application). In accordance to the flexibility principle, such applications should not have to pay for the cost of complete recovery when they can simply continue to operate on the communicator without further involving the failed processes.

4.4. Retrieving the local knowledge about failed processes

The next two functions, `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED` are introduced as a lightweight mechanism to continue using point-to-point operations on a communicator that contains failed processes. Using these functions, the application can determine which processes are known to have failed, and inform the MPI library that it acknowledges that no future receive operation can match sends from any of the reported dead processes. `MPI_COMM_FAILURE_GET_ACKED` returns the group containing all processes which were locally known to have failed at the time the last `MPI_COMM_FAILURE_ACK` was called. These functions can be used on any type of communicator, be it revoked or not.

The operation of retrieving the group of failed processes is split into two functions for two reasons. First, it permits multiple threads to synchronize on the acknowledge, to prevent situations were multiple thread read a different group of failed processes. Second, the acknowledge acts as a mechanism for alerting the MPI library that the application has been notified of a process failure, permitting to relax error reporting rules for "wildcard" `MPI_ANY_SOURCE` receives. Without an acknowledgement function, the MPI library would not be able to determine whether the failed process is a potential matching sender, and would have to take the safe course of systematically returning an error, thereby preventing any use of wildcard receives after the first failure. Once the application has called `MPI_COMM_FAILURE_ACK`, it becomes its responsibility to check that no posted "wildcard" receive should be matched by a send at a reported dead process, as MPI stops reporting errors for such processes. However, it will continue to raise errors for named point-to-point operations with the failed process as well as collective communications.

4.5. Ensuring consistent state

The last function permits deciding on the completion of an algorithmic section: `MPI_COMM_AGREE`. This function, which is intrinsically costly, is designed to be used sparingly, for example when a consistent view of the status of a communicator is necessary, such as during algorithm completion. This operation performs an agreement algorithm, computing the conjunction of boolean values provided by all alive processes in a communicator. Dead processes "participate" with the default value 'false'. It is important to note that this function will continue successfully even if a communicator has known failures (or if failures happen during the operation progress).

4.6. Beyond communicators

While communicator operations are the historic core of MPI, the standard has been extended over the years to support other types of communication contexts, namely shared memory windows (with explicit put/get operations) and collective file I/O. The same principles described in this paper are extended to these MPI objects in the complete proposal; in particular, windows and files have a similar Revoke function. A notable difference, though, is that file and window object do not have repair functions. These objects are initially derived from a communicator object, and the expected recovery strategy is to create a repaired copy of this communicator, before using it to create a new instance of the window or file object. While windows also have the failure introspection function `MPI_WIN_GET_FAILED`, which is useful for continuing active target operations on the window when failed processors can be ignored (similarly to point-to-point operations on a communicator), all file operations are collective, hence this function is not provided, as the only meaningful continuation of a failure impacting a file object is to revoke the file object. It should be noted that in the case of file objects, only failures of MPI processes (that may disrupt collective operations on the file) are addressed. Failures of the file backend itself are already defined in MPI-2.

5. Design rationale

In this section we discuss the rationale behind the proposed design by taking the view of MPI implementors in analyzing the challenges and performance implications that result from possible implementations, and explain why sometime counterintuitive designs are superior. While presenting implementation details or practical results is outside the scope of this paper, our claims that the proposed design does indeed achieve excellent performance is supported by an implementation, presented in Bland et al. (2012a).

5.1. 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 are intractable in theory, the existence of an appropriate failure detector permits resolving most of the theoretical impossibilities (Chandra and Toueg, 1996). One of the crucial goals of ULFM is to prevent deadlocks from arising, which indeed requires the use of some failure detection mechanism (in order to discriminate between arbitrarily long message delays and failures). However, because the practicality of implementing a particular type of failure detector strongly depends on the hardware features, the specification is intentionally vague and refrains from forcing a particular failure detection strategy. Instead, it leaves open to the implementations choices that better match the target system. On some systems, hardware introspection may be available and provide total awareness of failures (typically, an IPMI capable batch scheduler). However, on many systems, a process may detect a failure only if it has an active open connection with the failed resource, or if it is actively monitoring its status with heartbeat messages. In the latter situation requiring complete awareness of failures of every process by every process would generate an immense amount of system noise (from heartbeat messages injected into the network and the according treatments on the computing resources to respond to them), and it is known that MPI communication performance is very sensitive to system noise (Petrini et al., 2003). Furthermore, processes that are not trying to communicate with the dead process do not need to be aware of its failure, as their operations are with alive processors and therefore deadlock-free. As a consequence, to conserve generality and avoid extensive generation of system noise, failure detection in ULFM requires only to detect failures of processes that are active partners in a communication operation, so that this operation eventually returns an appropriate error. In the ideal case, the implementation should be able to turn on failure monitoring only for the processes it is expecting events from (such as the source or destination in a point-to-point operation). Some cases (such as wildcard receives from any source) may require a wider scoped failure detection scheme, as any processor is a potential sender. However, the triggering of active failure detection can be delayed according to implementation internal timers, so that latency critical operations do not have to suffer a performance penalty.

5.2. Communication objects status

A natural conception is to consider that detection of failures results in MPI automatically altering the state of all communication objects (i.e. communicators, windows, etc.) in which the associated process appears. 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 distributed system. As failure awareness is not expected to be global, the implementation would then require internal and asynchronous propagation of failure detection, a process prone to introduce jitter. Furthermore, MPI messages would be able to cross the toggling of the communication object into an invalid state, resulting in a confuse semantic where operations issued on a valid communication object would still fail, diluting the meaning of a valid and invalid state of communication objects.

We decided to take the opposite stance on the issue, failures never automatically modify the state of communication objects. Even if it contains failed processes, a communicator remains a valid communication object. Instead, error reporting is not intended to indicate that a process failed, but to indicate that an operation cannot complete. As long as no failures happen, the normal semantic of MPI must be respected. When a failure has happened, but the MPI operation can proceed without disruption, it completes normally. Obviously, when the failed process is supposed to participate to the result of the operation, it is impossible for the operation to succeed, and an appropriate error is returned. Posting more operations that involve the dead processes is allowed, but is expected to result in similar errors.

There are multiple advantages to this approach. First, the consistency of MPI objects is always guaranteed, as their state remains unchanged as long as users don't explicitly change it with one of the recovery constructs. Second, there is no need to introduce background propagation of failure detections to update the consistent state of MPI objects, because operations that need to report an error do actively require the dead process' participation, thereby active failure detection is forced only at the appropriate time and place.

5.3. Local or uniform error reporting

In the ULFM design, errors notify the application that an operation could not satisfy its MPI specification. However, most MPI operations are collective, or have a matching call at some other process. Should the same error be returned *uniformly* at all ranks that participated in the communication? Although such a feature is desirable for some users, as it permits easily tracking the global progress of the application (and then infer a consistent synchronized recovery point), the consequences on performance are dire. This would require that each communication conclude with a global agreement operation to determine the success or failure of the previous communication as viewed by each process. Such an operation has been shown to require at least $O(n^2)$ messages (where n is the number of processes participating in the communication), and would thus impose an enormous overhead on communication. With regards to the goal of maintaining unchanged level of performance, it is clearly unacceptable to double, at best, the cost of all communication operations, even when no failure happened.

As a consequence, in ULFM, the reporting of errors has a local semantic: the local completion status (in error, or successfully) cannot be used to assume whether the operation has failed or succeeded at other ranks. In many applications, this uncertainty is manageable, because the communication pattern is simple enough. When the communication pattern does not allow such flexibility, the application is required to resolve this uncertainty itself by explicitly changing the state of the communication object to *Revoked*. Indeed, it is extremely difficult for MPI to assess whether a particular communication pattern is still consistent (it would require computing global snapshots after any communication), while the user can know through algorithm invariants when it is the case. Thanks to that flexibility, the cost associated with consistency in error reporting is paid only after an actual failure has happened, and applications that do not need consistency can enjoy unchanged performance.

5.4. Restoring consistency and communication capabilities

Revoking a communication object result in a definitive alteration of the state of the object, that is consistent across all processes. This alteration is not to be seen as the (direct) consequence of a failure, but as the consequence of the user calling a specific operation on the communication object. In a sense, revoking a communication object explicitly achieves the propagation of failure knowledge that has intentionally not been required, but is provided when the user determines necessary. Another important feature of that change of state is that it is definitive. After a communication object has been revoked, it can never be repaired. The rationale is to avoid the matching to have to check for stale messages from past incarnations of a repaired communication object. Because the object is discarded definitively, any stale message matches the revoked object and is appropriately ignored without modifications in the matching logic. In order to restore communication capacity, the repair function derive new, fresh communication objects, that do not risk intermixing pre-failure operations.

5.5. Library construction

At the heart of all of the design decisions in ULFM was a minimalistic approach which encouraged extensions via supplementary libraries. Because fault-tolerance research shows such a clear need in the future and no single practice has emerged as dominant, ULFM provides the foundations to construct new consistency models on top of MPI as a library. For instance, if an application is willing to pay the performance cost of globally consistent collective operations which uniformly return error codes among participating processes, it can create a library which amends the existing collective operations with an agreement operation to decide the status of the communication. Further discussion of composing fault-tolerant techniques on top of ULFM can be found in Section 6.

While not specifically mentioned in the ULFM specification, library composition is a complex topic that required some consideration in the design. To ensure that libraries could interoperate and maintain a consistent view of the system throughout the software stack, a sample library stack was envisioned to demonstrate the feasibility of ULFM with other libraries. Figure 1 demonstrates one possible method of propagating failure information up through the stack to the application, then performing a top-down recovery to repair communication and continue the algorithm. As the figure shows, the recovery operations should occur at the highest level first, rather than the lowest. This is especially true when the algorithm requires the replacement of failed processes with new processes. If this replacement occurs at the lowest level, transparent to libraries and applications which sit on top of it, the MPI communicators lose their consistency. For example, if an application provides a communicator to a library, rank 2 in the communicator fails and the library automatically spawns a new process and inserts it into the communicator, the original application has no knowledge of this new process and therefore cannot bring it back into the original application. However, if the original application is responsible for repairing the communicator by recreating failed processes and providing the repaired communicator to the library, both levels can now communicate with the replacement process and the application can continue.

6. Composition of fault-tolerant libraries and applications

ULFM was specifically created not to promote any particular form of fault tolerance over another. Instead, it was designed to enable the support of many types of fault-tolerance techniques provided either via extension libraries or independent packages, with a portable and efficient interface. The portability claim is paramount, as one of the major obstacles for progress in the fault-tolerance area is the lack of consistent interfaces to retrieve and handle process failures from multiple MPI implementations. With ULFM constructs, as fault tolerance evolves as a research field and new ideas come about, their implementations can be built using the consistent and portable set of constructs with minimal overhead. This section will expand on how existing fault-tolerance techniques from Section 2 could be constructed in conjunction with ULFM.

6.1. Automatic methods

As fault tolerance continues to evolve, checkpoint/restart will continue to be a popular design for legacy codes and therefore should be supported by any new fault-tolerant environments. ULFM supports the coordinated rollback form of checkpoint/restart without modification as there are many libraries which function with the existing MPI Standard (Duell, 2002). However, partial rollback forms of checkpoint/restart can be developed which could take advantage of the additions in ULFM without requiring the application to restart entirely. Most of the event-logging mechanisms can be implemented in a portable way by using the already standardized PMPI interface. When a failure happens, the message logging library can revoke communicators internally in the PMPI hooks, silently swap them with replacements obtained by the usual shrink, spawn, merge combination, and continue.

In the case of replication and migration, the PMPI hooks are usually used to redirect messages to the appropriate target, or to integrate the consistent delivery protocol to multiple replicates. When operations are addressed to other processes, the library can intercept the MPI calls and recalculate their targets. For example, the library might redirect the communication to point to the currently active target depending on which replica is being used or if a process has migrated. When a failure happens, the similar internal hot swapping of the communication object can be realized in the PMPI interface.

By employing message-logging techniques and using ULFM constructs, the application might not need to roll back any processes, but could use an MPI_COMM_AGREE function to decide the status of the failed process and then consistently replay the messages that were sent to the recovered process.

6.2. Algorithm-based fault tolerance

Algorithm-based fault-tolerant (ABFT) codes are another form of application that could easily be directly integrated with ULFM. For ABFT codes which tolerating the lost of processes, a simple `MPI_COMM_REVOKE` followed by `MPI_COMM_SHRINK` approach could restore functionality by dropping failed processes. Other applications which require a full set of processes to continue can replace the failed processes by adding a call to `MPI_COMM_SPAWN`, and reintegrating the new processes in the original set. To the best of the authors' knowledge, neither of these types of applications require another level of complexity through an external library, but receive all of the fault-tolerance support they require through ULFM.

6.3. Transactional fault tolerance

Transactional fault tolerance can also be implemented as a library on top of ULFM by adding a new function that remaps to `MPI_COMM_AGREE` to determine the completion status of previous operations and, on success, saves any necessary data to stable storage, or, on failure, loads the necessary data from storage and returns to a previous point in the code. It would also be necessary to perform operations to replace failed processes using existing MPI-2 dynamic processing functions. Timers and bailouts that are defined in some transactional frameworks can be introduced in PMPI hooks.

7. Related work

Gropp and Lusk (2004) describe methods using the then current version of the MPI Standard to perform fault tolerance. They described methods including checkpointing, `MPI_ERRHANDLER`s, and using inter-communicators to provide some form of fault tolerance. They outline the goals of providing fault tolerance without requiring changes to the MPI Standard. However, at the time of writing, fault tolerance was still an emerging topic of research with few solutions beyond checkpointing and simple ABFT in the form of master–worker applications. As fault tolerance has evolved to include those paradigms mentioned in Section 2, the requirements on the MPI implementation have also grown, and the limited functionality emphasized are insufficient for general communication purposes.

Another notable effort was FT-MPI (Fagg and Dongarra, 2000). The overreaching goal was to support ABFT techniques, it thereby provide three failure modes adapted to this type of recovery techniques, but difficult to use in other contexts. In the *Blank* mode, failed processes were automatically replaced by `MPI_PROC_NULL`; messages to and from them were silently discarded and collective communications had to be significantly modified to cope with the presence of `MPI_PROC_NULL` processes in the communicator. In the *Replace* mode, faulty processes were replaced with new processes. In the *Shrink* mode, the communicator would be changed to remove failed processes (and ranks reordered accordingly). In all cases, only `MPI_COMM_WORLD` would be repaired and the application was in charge of rebuilding any other communicators. No standardization effort was pursued, and it was mostly used as a playground for understanding the fundamental concepts. A major distinction with the ULFM design is that when FT-MPI detects a failure, it repairs the state of MPI internally according to the selected recovery mode, and then only triggers the coordinated user recovery handle at all nodes. Library composition is rendered difficult by the fact that recovery preempts the normal flow of execution and returns to the highest level of the software stack without alerting intermediate layers that a failure happened.

A more recent effort to introduce failure handling mechanisms was the run-through stabilization proposal (Hursey et al., 2011). This proposal introduced many new constructs for MPI including the ability to "validate" communicators as a way of marking failure as recognized and allowing the application to continue using the communicator. It included other new ideas such as failure handlers for uniform failure notification. Because of the implementation complexity imposed by resuming operations on failed communicators, this proposal was eventually unsuccessful in its introduction to the MPI Standard.

Simultaneously with the proposed changes to the MPI Standard, Checkpoint-on-Failure (CoF) (Bland et al., 2012b) is a new protocol designed to permit supporting forward recovery strategies in the context of the existing MPI standard. In this strategy, when a failure happens, MPI operations return an error, but do not abort the application (a behavior defined as a "high-quality" implementation in MPI-2). However, it is not expected from MPI that communications can continue or resume at a later time. Instead of trying to recover immediately, the entire application undergoes checkpoint. Because the checkpoints are taken only after effective failures have happened, the traditional overhead of customary periodic checkpoint is eliminated and checkpoints are indeed taken at the optimal interval (one checkpoint per fault). After the checkpoints are taken, the application undergoes a complete restart, because, unlike in ULFM, MPI communications cannot be repaired without such a drastic measure. Once that full restart is completed, the application can proceed with its forward recovery strategy (typically including communicating) to restore the dataset of processes that have failed before completing their checkpoint.

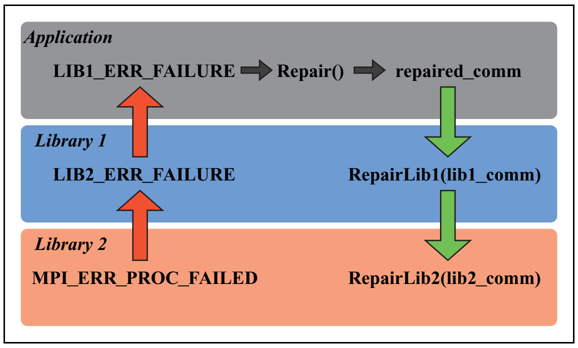

Figure 2. An example of library construction, error propagation, and recovery through the software stack.

8. Conclusion

Simple communication interfaces, such as sockets or streams, have been featuring robust fault tolerance for decades. It may come as a surprise that specifying the behavior of MPI when fail-stop failures strike is so challenging. In this paper we have identified the contentious issues, rooted in the fact that the state of MPI objects is implicitly distributed and that specifying the behavior of collective operations and communication routines requires a careful, precise investigation of unexpected consequences on the concepts as well as on the performance. We first took a review of the field of fault tolerance and recovery methods; most require that MPI can restore the full set of communication functionalities after a failure happened. Then, we proposed the ULFM interface, which responds to that demand, and took the critical viewpoint of the implementor unwilling to compromise performance, on a number of hidden, but crucial issues regarding the state of MPI objects when failure happen. Lastly, we took the viewpoint of MPI users, and depicted how the ULFM specification can be used to support high-level recovery strategies.

We believe that, beyond MPI, the insight gained in the ULFM design is applicable to other communication middleware relying on generic concepts such as stateful communication objects representing the context of a communication or defining collective operations. In particular, the pitfalls associated with defining a particular type of recovery strategy that matches only a niche of applications, rather than defining the minimal set of functionalities that permit restoring communication capabilities, as well as the caveats of returning uniform errors and its implementation cost should highlight similar difficulties in any type of distributed memory framework, and we hope some of the insight presented in this paper can be reused in this context.

ULFM is currently considered for standardization by the MPI Forum. More libraries and applications are being adapted to take advantage of its new constructs. As these developments conclude, a more compelling argument for ULFM will take shape and hopefully drive its adoption as a critical part of the future versions of the MPI standard.

Funding

This document describes combined research conducted under the following contracts: NSF-0904952 and NSF-1144042 between the U.S. National Science Foundation and the University of Tennessee, and DE-FC02-11ER26059 supported by the U.S. Department of Energy.

Note

1. The interested reader may refer to chapter 17 of the complete draft, available from http://fault-tolerance.org/ulfm/ulfm-specification

References

Alvisi L and Marzullo K (1995) Message logging: pessimistic, optimistic, and causal. In: *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*. Los Alamitos, CA: IEEE CS Press, pp. 229–236.

Bland W, Bouteiller A, Herault T, Hursey J, Bosilca G and Dongarra JJ (2012 a) An evaluation of User-Level Failure Mitigation support in MPI. In: Träff JL, Benkner S and Dongarra J (eds.) *19th EuroMPI*, Vienna, Austria. Berlin: Springer.

Bland W, Du P, Bouteiller A, Herault T, Bosilca G and Dongarra JJ (2012 b) A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI. In: *18th Euro-Par*, Rhodes Island, Greece. Berlin: Springer.

Bouteiller A, Bosilca G and Dongarra J (2010) Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience* 22(16): 2196–2211.

Bouteiller A, Herault T, Bosilca G and Dongarra JJ (2011) Correlated set coordination in fault tolerant message logging protocols. In: *Proceedings of Euro-Par'11 (II) (Lecture Notes in Computer Society*, vol. 6853). Berlin: Springer, pp. 51–64. DOI: http://dx.doi.org/10.1007/978-3-642-23397-5_6.

Bright A, Ellavsky M, Gara A, et al. (2005) Creating the BlueGene/L supercomputer from low-power SoC ASICs. In: *Solid-State Circuits Conference, Digest of Technical Papers (ISSCC)*, volume 1. Piscataway, NJ: IEEE, pp. 188–189. DOI: 10.1109/ISSCC.2005.1493932.

Buntinas D, Coti C, Herault T, et al. (2008) Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Computer Systems* 24(1): 73–84. DOI: 10.1016/j.future.2007.02.002.

Cappello F (2009) Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications* 23(3): 212.

Chakravorty S, Mendes CL and Kalé LV (2006) Proactive fault tolerance in MPI applications via task migration. In: *HiPC 2006, the IEEE High performance Computing Conference*. Los Alamitos, CA: IEEE Computer Society Press, pp. 485–496.

Chandra TD and Toueg S (1996) Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43(2): 225–267.

Chandy KM and Lamport L (1985) Distributed snapshots: determining global states of distributed systems. *Transactions on Computer Systems* 3(1): 63–75.

Davies T, Karlsson C, Liu H, Ding C and Chen Z (2011) High performance Linpack benchmark: a fault tolerant implementation without checkpointing. In: *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*. New York: ACM Press.

Dongarra J, Beckman P, et al. (2011) The international exascale software roadmap. *International Journal of High Performance Computing Applications* 25(11): 3–60.

Du P, Bouteiller A, et al. (2012) Algorithm-Based Fault Tolerance for dense matrix factorizations. In: *17th SIGPLAN PPoPP*. New York: ACM Press, pp. 225–234.

Duell J (2002) The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941.

Elnozahy ENM, Alvisi L, Wang YM and Johnson DB (2002) A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey* 34: 375–408.

Esteban Meneses CLM and Kalé LV (2010) Team-based message logging: Preliminary results. In: *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010)*.

Fagg G and Dongarra J (2000) FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In: *EuroPVM/MPI*.

Ferreira K, Stearley J, Laros J, et al. (2011) Evaluating the viability of process replication reliability for exascale systems. In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Request Permissions, pp. 1–12.

Gropp W and Lusk E (2004) Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications* 18: 363–372. DOI: 10.1177/1094342004046045.

Guermouche A, Ropars T, Snir M and Cappello F (2012) HydEE: failure containment without event logging for large scale send-deterministic MPI applications. In: *2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pp. 1216 –1227. DOI: 10.1109/IPDPS.2012.111.

Hursey J, Graham RL, Bronevetsky G, Buntinas D, Pritchard H and Solt DG (2011) Run-through stabilization: An MPI proposal for process fault tolerance. In: *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece.

Hursey J, Squyres J, Mattox T and Lumsdaine A (2007) The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: *IEEE International Parallel and Distributed Processing Symposium, 2007 (IPDPS 2007)*, pp. 1–8. DOI: 10.1109/IPDPS.2007.370605.

Petrini F, Frachtenberg E, Hoisie A and Coll S (2003) Performance evaluation of the Quadrics interconnection network. *Cluster Computing* 6(2): 125–142. DOI: 10.1023/A:1022852505633.

Schroeder B and Gibson GA (2007) Understanding failures in petascale computers. *Journal of Physics: Conference Series* 78: 012022.

Silva LM and Silva JG (1998) System-level versus user-defined checkpointing. In: *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, Washington, DC. Los Alamitos, CA: IEEE Computer Society Press, p. 68.

Skjellum A (2012) Middle-out Transactional Requirements on Exascale Parallel Middleware, Storage, and Services. Technical Report UABCIS-TR-2012-020312, University of Alabama at Birmingham, Computer and Information Sciences.

The MPI Forum (2012) MPI: A Message-Passing Interface Standard, Version 3.0. Technical report.

Wang C, Mueller F, Engelmann C and Scott SL (2008) Proactive process-level live migration in HPC environments. In: *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ: IEEE Press, pp. 1–12.

Author biographies

Wesley Bland is a Graduate Research Assistant in the Innovative Computing Laboratory at the University of Tennessee, Knoxville. He will be defending his thesis on methods to add fault tolerance to high-performance computing middle-wares, specifically MPI. He received his B.S. degree at Tennessee Technological University in Cookeville, TN in 2007 and his M.S. degree from the University of Tennessee, Knoxville in 2009. Upon graduation, he will begin a Postdoctoral Appointment at Argonne National Laboratory in Chicago, IL.

Aurelien Bouteiller is currently a researcher at the Innovative Computing Laboratory, University of Tennessee. He received is Ph.D. from University of Paris in 2006 on the subject of rollback recovery fault tolerance. His research is focused on improving performance and reliability of distributed memory systems. Toward that goal, he investigated automatic (message-logging-based) checkpointing approaches in MPI, algorithm-based fault tolerance approaches and their runtime support, mechanisms to improve communication speed and balance within nodes of many-core clusters, and employing emerging data flow programming models to negate the raise of jitter on large-scale systems. These works resulted in over thirty publications in international conferences and journals and three distinguished paper awards from IPDPS and EuroPar. He his also a contributor to Open MPI and one of the leading designer of MPI Forum efforts toward fault-tolerance interfaces.

Thomas Herault is a Research Scientist in the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research focuses on fault-tolerance in high-performance computing, middleware for communication and data-flow environment in distributed memory systems. He received his B.S. and M.S. degrees in computer science from the Paris-Sud University at Orsay, France. He defended his PhD in computer science on failure detection and mending in self-stabilizing systems at the Paris-Sud University, Orsay, France, in 2003, then joined the Grand-Large INRIA team and the Parall team of the Informatics Research Laboratory of the University of Orsay, France, where he held the position of assistant professor for 5 years. In 2008, he then joined the Innovative Computing Laboratory of the University of Tennessee, Knoxville.

George Bosilca holds an Assistant Research Professor at the University of Tennessee. He received his Ph.D. from University of Paris in 2003 in the domain of Parallel Architectures and programming models. He joined the University of Tennessee in 2003 as a Post-doctoral Researcher. He specialized in several aspects of high-performance computing, from low-level drivers up to high-level programming paradigms. He is actively involved in several projects preparing the software stack for the challenges of tomorrow's hardware requirements, in terms of heterogeneity, degree of parallelism, scalability and resilience. He remains an active contributor and lead architect of several software packages, such as FT-MPI, Open MPI, and PaRSEC.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; and in 2011 he was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.