Lessons Learned Implementing User-Level Failure Mitigation in MPICH

Wesley Bland, Huiwei Lu, Sangmin Seo, Pavan Balaji
Argonne National Laboratory
Mathematics and Computer Science Division
Argonne, IL USA
{wbland, huiweilu, sseo, balaji}@anl.gov

***Abstract*—User-level failure mitigation (ULFM) is becoming the front-running solution for process fault tolerance in MPI. While not yet adopted into the MPI standard, it is being used by applications and libraries and is being considered by the MPI Forum for future inclusion into MPI itself. In this paper, we introduce an implementation of ULFM in MPICH, a high-performance and widely portable implementation of the MPI standard. We demonstrate that while still a reference implementation, the runtime cost of the new API calls introduced is relatively low.**

I. Introduction

In an effort to introduce a standardized way of handling fail-stop process failures that have become increasingly common on large-scale hardware, the MPI Forum is investigating the user-level failure mitigation (ULFM) [3] proposal put forth by the Fault Tolerance Working Group. The proposal defines the mechanisms necessary to implement fault tolerance in applications and libraries in order to allow applications to continue execution after failures.

In this paper, we investigate the challenges faced while implementing ULFM in MPICH and demonstrate how such challenges were overcome. We describe novel failure discovery mechanisms, internal tracking, and algorithmic implementations. We also demonstrate that because of the tight coordination between layers in the MPICH architecture, we are able to simplify internal tracking of communication object status and to propagate failures across the entire user application quickly, compared with existing implementations [1].

The remainder of the paper is organized as follows. Section II discusses some of the background work with ULFM and what the ULFM proposal covers; Section III discusses the implementation of ULFM in MPICH [2]; Section IV demonstrates the performance of our implementation; Section V summarizes our conclusions and proposes some future work.

II. Background

ULFM was designed as a way to allow both applications and libraries to implement resilience mechanisms with MPI. While applications can use it to develop their own application-specific solutions, the original intention of ULFM (as with MPI itself) was to encourage libraries to implement generic solutions for MPI fault tolerance that could be portable to other applications and MPI implementations. In this section, we discuss the ULFM proposal as well as other work being done related to ULFM.

A. ULFM Specification

At the time of writing, ULFM has yet to be adopted by the MPI Forum as part of the MPI standard. The specification is largely stable, however, and has been available to users since 2012. While the current version focuses on communicator-based operations and defers one-sided and file I/O operations to follow-on specifications, the available document [1] includes the entire proposed specification. ULFM defines a limited set of new operations with the intention that they would be the low-level API for other fault-tolerant libraries. The operations are divided into four categories: failure notification, failure discovery, failure propagation, and failure recovery.

For failure notification, no functions are actually necessary. Instead, failures are reported via the return codes of all MPI communication operations. If a process failure prevents an operation from fulfilling its defined specification, it will return the error class `MPI_ERR_PROC_FAILED`. This signals to the user that the operation did not complete successfully and the user needs to take some action to repair the application or continue with fewer processes. An important tenet of ULFM is to preserve failure-free performance, something it accomplishes by defining failure notification as local. When a failure is reported to one process via the return code or error handler, there is no guarantee that other processes will receive the same notification. When such global knowledge is required, failure discovery functions must be used.

For the application to discover which processes in a communicator have failed, it needs to use a combination of two functions: `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED`. These together provide the application with an MPI group containing all failed processes in the communicator. As mentioned already, this group is not guaranteed to be consistent across all processes. One way to ensure global knowledge of failures is to use the function `MPI_COMM_AGREE`. This function propagates failure knowledge to all participating processes, ensuring that future failure discovery calls will contain at least the processes that have failed up to this point. It also performs a fault-tolerant agreement among alive processes to determine the bitwise "and" value of an integer provided as an argument. This allows the application to do its own fault tolerance logic at points where synchronization is necessary, such as the end of an algorithm.

Failure propagation ensures that other processes that need to know about failures receive that knowledge. While some

of this is automatic when failures prevent an operation from completing normally, manual propagation is necessary at times to prevent incorrect application behavior or deadlocks. For such cases, `MPI_COMM_REVOKE` is provided. With this function, a single process causes the communicator to become invalid for communication for all other processes. This function will cause all other processes to eventually be unable to use the communicator for anything other than local operations. All nonlocal operations will return the error code `MPI_ERR_REVOKED`. While this function has a strong effect on all ranks, it is sometimes necessary because of the limited automatic propagation of errors that MPI provides. Two operations will still complete on a communicator that has been revoked: `MPI_COMM_AGREE` and `MPI_COMM_SHRINK` (described next).

Recovery can happen in various ways. If collective communication is not required, recovery may not be necessary at all: discovering the location of the failure and excluding that process from further operations could be sufficient. For most applications, however, communication must be repaired in some way. In such a case, a new MPI communicator must be created on which further communication will take place. To this end, ULFM proposes the new function `MPI_COMM_SHRINK`. This function internally determines the group of failed processes and creates a new communicator based on another communicator that excludes those processes. This function is not complete for all forms of recovery, however. For instance, applications may require that failed processes be replaced in order to maintain computational capacity, or that processes retain their original ranks within the MPI communicator. Existing MPI calls can be used to accomplish these supplementary needs (`MPI_COMM_SPAWN` and `MPI_COMM_SPLIT`, respectively).

B. Related Work

Currently only one other publicly available implementation of the ULFM specification exists. This version is based on a branch of Open MPI from 2012 and is being developed by a team at the University of Tennessee's Innovative Computing Laboratory [1]. While we do not intend to invalidate the work done on that implementation, we are making ULFM available more publicly in an official release of MPICH. Because MPICH is used as the basis for many derivative implementations (MVAPICH, Intel MPI, IBM's MPI products, Cray MPI, etc.), implementing ULFM in MPICH will also expedite adoption more widely by large-scale MPI implementations.

Previous attempts have been made to implement fault tolerance in MPI. The most well known is FT-MPI [4], which began from similar ideas but explored more automated recovery rather than allowing the user to have more control. More recently, other projects have investigated MPI fault tolerance, including FA-MPI [6], which implements MPI resilience through transactions, and Red MPI [5], which achieves soft error resilience by executing MPI jobs redundantly.

III. Implementation

The implementation of ULFM provided in MPICH is designed to be a reference implementation, not optimized for any particular platform. MPICH derivatives generally optimize individual operations for their targeted platform, something that can easily be done for all of the new features provided by ULFM. This section gives details about the implementation of the new ULFM API and accompanying runtime changes required by the specification.

A. Failure Notification

Before any API can be implemented for ULFM, the underlying runtime layer must be able to accurately detect failures and report them to the user. In MPICH, one of the ways this is handled is by the process manager, Hydra. Hydra handles launching, monitoring, and shutting down MPI jobs by running as a set of daemons on each node, independent of the application execution. They are started just before the application is launched and cleaned up as the application shuts down. For faults that do not cause an entire node to fail, the local Hydra daemon will detect the failure with traditional Unix process management tools such as the abnormal termination of its child process. If the fault does cause a full-node failure, it is detected via the connections between daemons. These connections are persistent, and their abnormal termination signals the system that a failure has occurred. Currently, Hydra supports only a flat topology for its launcher, which has each daemon connect back to the main `mpiexec` process. Thus, the process manager's communication topology does not need to be repaired after a failure. Ongoing work on a next-generation process manager may change this situation in the future, however.

Failures are also discovered by the MPICH runtime itself. MPICH uses modules called netmods as the low-level data transportation code. These netmods provide their own error checking and notification, which is propagated back up through the network stack via internal request objects that track each communication operation. When a failure is detected, the request object is marked, and the internal communication objects are updated appropriately to track whether certain operations are allowed. For instance, after a failure, all communicators that contain the failed process can no longer be used for operations that use `MPI_ANY_SOURCE`. Because MPICH uses integers as internal identifiers for its communication objects, finding all affected MPI communicators is trivial, something that is more complex in those implementations that do not maintain ways to easily track internal communication objects.

Another complexity of failure notification comes from nonblocking operations when calling one of the MPI completion functions that accepts a list of `MPI_Request` objects (i.e., `MPI_TESTALL`, `MPI_TESTSOME`, `MPI_TESTANY`, `MPI_WAITALL`, `MPI_WAITSOME`, `MPI_WAITANY`). In such a scenario, some requests might have failed directly, such as an `MPI_IRECV` from rank 1 when rank 1 has failed. In this case, all other request objects in the same operation must also return with a new error code, `MPI_ERR_PROC_FAILED_PENDING`. This error code is intended to notify the user of a failure, but not complete the request, meaning that the user should be able to continue such a request in the future. Such a situation requires careful management of the request queue within MPICH. We handle this situation by signaling the progress engine that request operations have completed and then checking for the situation at the MPI level. If no failure exists, the MPI level re-enters the progress engine to complete the call later. Otherwise, the

MPI call marks the requests with the appropriate error code and returns to the user. We use a similar technique to handle nonblocking operations that specify MPI_ANY_SOURCE as their source.

B. Agreement

MPI_COMM_AGREE accomplishes multiple tasks simultaneously. First, all processes must determine a consistent view of which processes in the communicator have failed. If that set is not the same, they must all return MPI_ERR_PROC_FAILED. We accomplished this by performing a simple reduction to a single rank to determine the group of failed processes, then broadcasting the resulting group back out to the other processes. Once the reduction has taken place, the group that the single rank contains is the one that will be used for the remainder of the algorithm. If any new failures are detected before the entire function is completed, they will be reported to the user with an error.

After the group of failed processes is determined and all ranks agree on a consistent group, another allreduce is done to perform a bitwise *AND* on the value of the flag passed into the agreement call. This operation is simple in MPICH because all allreduce functions are based on an operation that takes a group as an argument. The final piece of the operation is to ensure that all ranks return a consistent return code. This is done with a final allreduce in which all processes contribute the error code from the previous reduction. If any process contributes an error code other than success, all processes will see the result and collectively return a non-success error code. Otherwise, all processes will return MPI_SUCCESS and exit.

C. Revocation

Implementing revoke was one of the trickiest parts of ULFM. The challenge is to issue a command that will eventually reach all processes in a communicator but will not revoke other communicators. The most difficult piece of the implementation is to ensure that revoke messages will not impact future communicators. If the revoke messages are still being received at some point in the future because of message delay, they must not cause that communicator to also be revoked, including one that reuses the same context ID (the internal identifier of a communicator) as the communicator being revoked.

The basic implementation of our revocation involves performing a message flood between all alive processes in a communicator. When a process calls MPI_COMM_REVOKE, it send a message to all other processes in the communicator announcing the revoke call. Whenever another process receives the revoke message, it will also send a message to all other processes to indicate that it has received the revoke command and has also revoked its own communicator. By implementing the revocation as a message flood, we solve the problem of protecting future use of the same context ID from also being revoked. No process will release its use of a context ID until it has received a message from all other processes in a communicator indicating that they too have revoked their own communicators. Once a process has received the correct number of notifications, it releases its use of the internal context ID, and it becomes available for future allocation. By treating a failure notification as a revoke in this algorithm, we also prevent failed processes from causing a deadlock by not participating in the revoke algorithm. When a process receives a failure notification, it decrements the counter keeping track of the expected number of revoke messages.

In addition to correctly propagating the revoke messages, our implementation must also correctly clean up ongoing operations. MPICH has a small number of internal message queues that the revoke operation traverses to cancel any operations on the specified communicator. This process is implemented with a simple search of the posted and unexpected message queues.

D. Communicator Shrinking

Shrinking the communicator is a multiphase process similar to agreement. In the first phase, all processes must determine the same group of failed processes. In the second phase, all processes that have not failed construct a communicator using the standard communicator construction functions. The first phase is implemented in almost the same way as in MPI_COMM_AGREE. The primary difference in the implementation between MPI_COMM_SHRINK and MPI_COMM_AGREE is that during shrinking, an undetected failed process does not cause the operation to abort. Instead the failure discovery phase is rerun until all processes form a consistent group.

Once the group of failed processes is consistently created, the alive processes construct the new communicator via existing mechanisms in MPICH. This portion of the code is actually the same as what is used to implement MPI_COMM_CREATE_GROUP, a communicator creation function that takes an MPI group as an argument to determine the participants in the new communicator. If this function fails during its execution because of a new process failure, the entire algorithm is restarted from the failure discovery.

IV. Evaluation

Since this is a reference implementation, performance is not the primary motivator of this work. Nevertheless, we show initial performance results here in order to demonstrate its viability as an unoptimized implementation of the ULFM specification. We demonstrate MPI_COMM_AGREE and MPI_COMM_SHRINK. Our tests were run using the Fusion cluster at Argonne National Laboratory. Fusion is a 320-node cluster with two 8-core Intel Nehalem processors and 36 GB of RAM per node per node and InfiniBand QDR interconnect, although we use the MPICH TCP netmod for these experiments. We repeated each test 30 times, removing statistical outliers that can become prevalent during heavy usage of the cluster.

A. Agreement Performance

To demonstrate the performance of MPI_COMM_AGREE, we compare in Figure 1 three instances of the algorithm: with no failures, with unacknowledged failures, and with acknowledged failures. We see that the unacknowledged failures version causes the worst performance. At first glance, one might expect this version of the algorithm to be the fastest because it allows the algorithm to leave early. However, we must still complete the entire agreement even if a failure is

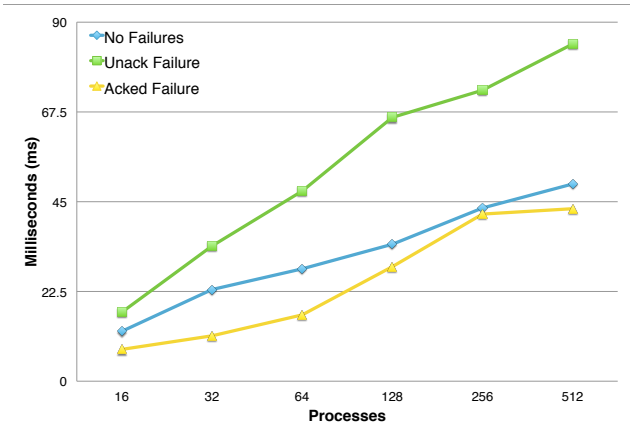

Fig. 1: Runtime of Agreement

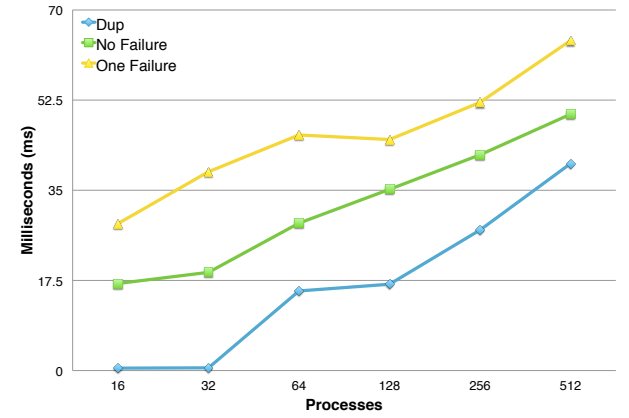

Fig. 2: Runtime of Shrink

detected in order to avoid a deadlock, so this version actually has a higher runtime because it attempts to communicate with the failed process repeatedly and triggers parts of the failure notification code each time the failure is rediscovered.

This is same reason that the performance of the failure-free execution and that of the acknowledged failure execution is similar. Both codes still perform the failure discovery to determine whether all processes have a consistent view of the system, and both do not need to communicate with failed processes. The version with a failure does have slightly fewer processes with which to communicate, which removes a round of communication for the recursive doubling algorithm used by MPICH's allreduce algorithm [7].

B. Shrink Performance

In Figure 2 we compare the performance of `MPI_COMM_SHRINK` with that of the simplest communicator creation function in MPI, `MPI_COMM_DUP`. First, we run the shrink algorithm with no failures, demonstrating the added overhead of a single round of failure discovery before beginning the communicator creation algorithm. This shows approximately 20% overhead at larger scales.

Second, we inject a failure at the point in the shrinking algorithm that would cause the largest performance degradation, immediately after the failure discovery. This will cause all the processes to do the failure discovery portion at least twice: at least once where some subset of the processes do not discover the failure and once where all of the processes do discover the failure. We see that the performance for this version of the algorithm adds approximately 30% additional overhead at larger scales for the processes to repeat the algorithm the necessary number of times to converge on a set of alive processes.

V. Conclusion

As the fault tolerance community converges on a proposal for MPI to handle process failures, a wider variety of implementations must become available to ensure that ULFM can be used on many systems. In this work, we have demonstrated a new implementation in MPICH that provides all the functionality with reasonable, though unoptimized, performance.

We plan to implement a more scalable algorithm for the failure propagation used by both `MPI_COMM_SHRINK` and `MPI_COMM_AGREE`. We also are working on various application studies and libraries to improve the usability of ULFM for application developers.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357.

We gratefully acknowledge the computing resources provided on Fusion, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] [Online]. Available: http://www.fault-tolerance.org

[2] P. Balaji, W. Bland, W. Gropp, R. Latham, H. Lu, A. J. Pena, K. Raffenetti, R. Thakur, and J. Zhang, "MPICH Users Guide," 2014.

[3] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in *Recent Advances in the Message Passing Interface*. Springer, 2012, pp. 193–203.

[4] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," pp. 346–353, 2000.

[5] D. Fiala, F. Mueller, C. Engelmann, K. Ferreira, R. Brightwell, and R. Riesen, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the* 25^{th} *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2012*. Salt Lake City, UT, USA: ACM Press, New York, NY, USA, Nov. 2012, pp. 78:1–78:12.

[6] A. Hassani, A. Skjellum, and R. Brightwell, "Design and evaluation of fa-mpi, a transactional resilience scheme for non-blocking mpi," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 750–755.

[7] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2003, pp. 257–267.