User Level Failure Mitigation in MPI

Wesley Bland

Innovative Computing Laboratory, University of Tennessee
wbland@eecs.utk.edu

1 Introduction

In a constant effort to deliver steady performance improvements, the size of High Performance Computing (HPC) systems, as observed by the Top 500 ranking[1], has grown tremendously over the last decade. This trend, along with the resultant decrease of the Mean Time Between Failure (MTBF), is unlikely to stop; thereby many computing nodes will inevitably fail during application execution [5]. It is alarming that most popular fault tolerant approaches see their efficiency plummet at Exascale [3, 4], calling for more efficient approaches evolving around application centric failure mitigation strategies [7].

The prevalence of distributed memory machines promotes the use of a message passing model. An extensive and varied spectrum of domain science applications depend on libraries compliant with the MPI Standard Although unconventional programming paradigms are emerging, most delegate their data movements to MPI and it is widely acknowledged that MPI is here to stay. However, MPI has to evolve to effectively support the demanding requirements imposed by novel architectures, programing approaches, and dynamic runtime systems. In particular, its support for fault tolerance has always been inadequate [6]. To address the growing interest in fault-aware MPI, a working group in the context of the MPI Forum has proposed basic interfaces and semantics called User-Level Failure Mitigation (ULFM) [1] to enable libraries and applications to survive the increasing number of failures, and, subsequently, to repair the state of the MPIs world. The contributions of this work are at providing a high-level overview of the proposed semantics, and to evaluate the difficulties faced by MPI implementors on delivering a low-impact implementation on the failure-free performance. For a more complete evaluation and discussion of ULFM and its related work, refer to this abstract's accompanying paper [2].

2 Motivation

A major concept followed in the design of any type of parallel programming paradigm, and this in order to provide a meaningful and understandable programming approach, is the avoidance of any potential deadlocks. In addition to this critical requirement the mechanisms involved in the fault management were build with the goal of flexibility, simplicity and performance.

[1] http://www.top500.org/

Clearly the most important point, no MPI call (point-to-point or collective) can block indefinitely after a failure, but must either succeed or raise an MPI error. Fault tolerance at the application level is impossible if the application cannot regain full control of the execution after a process failure. The MPI library must guarantee that it will automatically stabilize itself following a process failure, and provide the tools necessary for the application to resolve its own deadlock scenarios on an application specific basis.

Second, the API should allow varied fault tolerant models to be built. MPI has been conceived with the goal of portability and extendability, and have been constructed to support libraries leveraging existing MPI constructs to create more abstractions or tighter integration with libraries. Maintaining this design strength was paramount for ULFM, and it provides this capability so other levels of consistency can be supported as needed by higher-level concepts. Transactional fault tolerance, strongly uniform collective operations, and other FT techniques can all therefore be built upon the proposed set of constructs.

An API should be easy to understand and use in common scenarios, as complex tools have a steep learning courve and a slow adoption by the targeted communities. To this end, the number of newly proposed constructs have been reduced to five (along with nonblocking variants). These five functions provide the minimal set of tools to implement fault tolerant applications and libraries.

Two major pitfalls must be avoided when implementing these concepts: jitter prone, permanent monitoring of the health of peers a process is not actively communicating with, and expensive consensus required for returning consistent errors at all ranks. The operative principle is that fault-related errors are local knowledge, and are not indicative of the return status on remote processes. Errors are raised at a particular rank, when based on local knowledge it is known that a particular operation cannot complete because a participating peer has failed.

3 New MPI Constructs

`MPI_COMM_FAILURE_ACK` *&* `MPI_COMM_FAILURE_GET_ACKED`*:* These two calls allow the application to determine which processes within a communicator have failed. The acknowledgement function serves to mark a point in time used as a reference for the second function which returns the group of processes which were locally know to have failed. After acknowledging failures, the application can resume `MPI_ANY_SOURCE` point-to-point operations between non-failed processes, but operations involving failed processes will continue to raise errors.

`MPI_COMM_REVOKE`*:* For scalability purposes, failure detection is local to a process's neighbors as defined by the application's communication pattern. This non-global error reporting may result in some processes continuing their normal, failure-free execution path, while others have diverged to the recovery execution path. As an example, if a process, unaware of the failure, posts a reception from another process that has switched to the recovery path, the matching send will never be posted and the receive operation will deadlock. The revoke operation

provides a mechanism for the application to resolve such situations before entering the recovery path. A revoked communicator becomes improper for further communication, and all future or pending communications on this communicator will be interrupted and completed with the new error code `MPI_ERR_REVOKED`.

`MPI_COMM_SHRINK`*:* The shrink operation allows the application to create a new communicator by eliminating all failed processes from a revoked communicator. The operation is collective and performs a consensus algorithm to ensure that all participating processes complete the operation with equivalent groups in the new communicator. This function cannot return an error due to process failure. Instead, such errors are absorbed as part of the consensus algorithms and will be excluded from the resulting communicator.

`MPI_COMM_AGREE`*:* This operation provides an agreement algorithm which can be used to determine a consistent state between processes when such strong consistency is necessary. The function is collective and forms an agreement over a boolean value, even when failures have happened or the communicator has been revoked. The agreement can be used to resolve a number of consistency issues after a failure, such as uniform completion of an algorithmic phase or collective operation, or as a key building block for strongly consistent failure handling approaches (such as transactions).

4 Implementation Issues

Some of the recovery routines described in Section 3 are unique in their ability to deliver a valid result despite the occurrence of failures. This specification of correct behavior across failures calls for resilient, more complex algorithms. In most cases, these functions are intended to be called sparingly by users, only after actual failures have happened, as a means of recovering a consistent state across all processes. This section describes the algorithms that can be used to deliver this specification and their cost. An evaluation of the failure-free impact on implementations can be found in [2].

Agreement: The agreement can be conceptualized as a failure-resilient reduction on a boolean value. Many agreement algorithms have been proposed in the literature; the log-scaling two-phase consensus algorithm used by the ULFM prototype is one of many possible implementations of `MPI_COMM_AGREE` operation based upon prior work in the field. Specifically, this algorithm is a variation of the multi-level two-phase commit algorithms [9]. A more extensive discussion of the algorithm and its complexity has been published by Hursey, et.al. [8].

Revoke: A concern with the revoke operation is the number of supplementary conditions introduced to the latency critical path. Indeed, most completion operations require a supplementary conditional statement to handle the case where the underlying communication context has been revoked. However, the prediction branching logic of the processor can be hinted to favor the failure free

outcome, resulting in a single load of a cached value and a single, mostly well-predicted, branching instruction, unlikely to affect the instruction pipeline. It is notable that non-blocking operations raise errors related to process failure only during the completion step, and thus do not need to check for revocation before the latency critical section.

Shrink: The Shrink operation is, algorithmically, an agreement on which the consensus is done on the group of failed processes. Hence, the two operations have the same algorithmic complexity. Indeed, in the prototype implementation, `MPI_COMM_AGREE` and `MPI_COMM_SHRINK` share the same internal implementation of the agreement.

5 Performance Analysis

A short performance analysis follows which summarizes the efficiency of a representative ULFM implementation based on the development trunk of Open MPI (r26237). A more complete analysis can be found in [2].

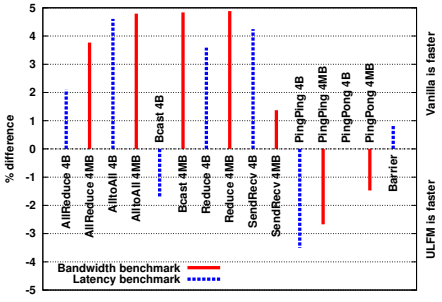

Fig. 1. IMB: ULFM vs. Vanilla Open MPI (Romulus)

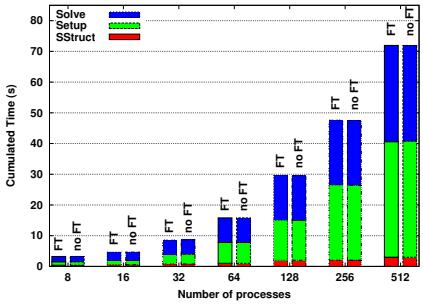

Fig. 2. Sequoia-AMG: ULFM vs. Vanilla Open MPI at various scales (Smoky)

The impact on shared memory systems, sensitive to small modifications of the MPI library, has been assessed on Romulus – a large shared memory machine – using the IMB benchmark suite (v3.2.3) as shown in Figure 1. The duration of all benchmarks remains below 5%, within the standard deviation of the machine's implementation

To measure the impact of the prototype on a real application, we used the Sequoia AMG benchmark[2], an Algebraic Mult-Grid (AMG) linear system solver for unstructured mesh physics. A weak scaling study was conducted up to 512 processes following the problem *Set 5*. Figure 2 compares the time slicing of three main phases (Solve, Setup, and SStruct) of the benchmark, with the vanilla implementation of Open MPI, and the ULFM enabled one. The application itself is not fault tolerant and does not use the features proposed in ULFM. This benchmark demonstrates that a careful implementation of ULFM need not impact the performance of the MPI implementation.

[2] https://asc.llnl.gov/sequoia/benchmarks/#amg

6 Conclusion

Many responsible voices agree that sharp increases in the volatility of future, extreme scale computing platforms are likely to imperil our ability to use them for advanced applications that deliver meaningful scientific results and maximize research productivity. Since MPI is currently, and will likely continue to be – in the medium-term – both the de-facto programming model for distributed applications and the default execution model for large scale platforms running at the bleeding edge, it is the place in the software infrastructure where semantic and run-time support for application faults needs to be provided.

The ULFM proposal is a careful but important step forward toward accomplishing this goal delivering support for a number of new and innovative resilience techniques through simple, familiar API calls, but it is backward compatible with previous versions of the MPI standard, so that non fault-tolerant applications (legacy or otherwise) are supported without any changes to the code. Perhaps most significantly, applications can use ULFM-enabled MPI without experiencing any degradation in their performance, as we demonstrate in this paper. Some of these applications along with other portable libraries are currently being refactored to take advantage of ULFM semantics.

The author would like to acknowledge his co-authors in the full paper [2]: Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra.

References

1. Bland, W., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: A proposal for User-Level Failure Mitigation in the MPI-3 standard. Tech. rep., Department of Electrical Engineering and Computer Science, University of Tennessee (2012)
2. Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.J.: An evaluation of User-Level Failure Mitigation support in MPI. In: 19th EuroMPI. Springer, Vienna, Austria (Sep 2012)
3. Bosilca, G., Bouteiller, A., Brunet, É., Cappello, F., Dongarra, J., Guermouche, A., Hérault, T., Robert, Y., Vivien, F., Zaidouni, D.: Unified Model for Assessing Checkpointing Protocols at Extreme-Scale. Tech. Rep. RR-7950, INRIA (2012)
4. Bougeret, M., Casanova, H., Robert, Y., Vivien, F., Zaidouni, D.: Using group replication for resilience on exascale systems. Tech. Rep. 265, LAWNs (2012)
5. Cappello, F., Geist, A., Gropp, B., Kalé, L.V., Kramer, B., Snir, M.: Toward exascale resilience. IJHPCA 23(4), 374–388 (2009)
6. Gropp, W., Lusk, E.: Fault tolerance in Message Passing Interface programs. IJHPCA 18, 363–372 (2004)
7. Huang, K., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 100(6), 518–528 (1984)
8. Hursey, J., Naughton, T., Vallee, G., Graham, R.L.: A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In: 18th EuroMPI. LNCS, vol. 6690, pp. 255–263. Springer (2011)
9. Mohan, C., Lindsay, B.: Efficient commit protocols for the tree of processes model of distributed transactions. In: SIGOPS OSR. vol. 19, pp. 40–52. ACM (1985)