Computing (2013) 95:1171–1184
DOI 10.1007/s00607-013-0331-3

An evaluation of User-Level Failure Mitigation support in MPI

Wesley Bland · Aurelien Bouteiller · Thomas Herault · Joshua Hursey · George Bosilca · Jack J. Dongarra

Received: 18 December 2012 / Accepted: 3 May 2013 / Published online: 29 May 2013

Abstract As the scale of computing platforms becomes increasingly extreme, the requirements for application fault tolerance are increasing as well. Techniques to address this problem by improving the resilience of algorithms have been developed, but they currently receive no support from the programming model, and without such support, they are bound to fail. This paper discusses the failure-free overhead and recovery impact of the user-level failure mitigation proposal presented in the MPI Forum. Experiments demonstrate that fault-aware MPI has little or no impact on performance for a range of applications, and produces satisfactory recovery times when there are failures.

Keywords MPI · Fault tolerance · User-level fault mitigation

Mathematics Subject Classification 68M14 · 68M15

W. Bland · A. Bouteiller · T. Herault · G. Bosilca (✉) · J. J. Dongarra
Innovative Computing Laboratory, University of Tennessee, Knoxville, USA
e-mail: bosilca@icl.utk.edu

W. Bland
e-mail: bland@icl.utk.edu

A. Bouteiller
e-mail: bouteill@icl.utk.edu

T. Herault
e-mail: herault@icl.utk.edu

J. J. Dongarra
e-mail: dongarra@icl.utk.edu

J. Hursey
University of Wisconsin-La Crosse, La Crosse, USA
e-mail: jjhursey@cs.uwlax.edu

1 Introduction

In a constant effort to deliver steady performance improvements, the number of hardware resources of high performance computing (HPC) systems, as observed by the Top 500 ranking,[1] has grown tremendously over the last decade. This trend is unlikely to stop, as outlined by the International Exascale Software Project (IESP) [11] projection of the Exaflop platform, a milestone that should be reached as soon as 2019. Based on the foreseeable limits of the infrastructure costs, an Exaflop capable machine is expected to be built from gigahertz processing cores, with thousands of cores per computing node, thus requiring millions of computing cores to reach the mark. Even under the most optimistic assumptions about the individual components' reliability, probabilistic amplification from using millions of nodes has a dramatic impact on the mean time between failure (MTBF) of the entire platform. The probability of a failure happening *during the next hour* on an Exascale platform is disturbingly close to 1; thereby many computing nodes will inevitably fail during the execution of an application [9]. It is even more alarming that most popular fault tolerant approaches (coordinated and uncoordinated checkpoint/restart) see their efficiency plummet at Exascale [5,6], calling for application centric failure mitigation strategies [17].

The prevalence of distributed memory machines promotes the use of the message passing model. An extensive and varied spectrum of domain science applications depend on libraries compliant with the MPI standard.[2] Although unconventional programming paradigms are emerging [20,22], 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 [15]. To address the growing interest in fault-aware MPI, a working group has been formed in the context of the MPI Forum. Their User-Level Failure Mitigation (ULFM) [2] proposal features the basic interface and new semantics to enable applications and libraries to repair the state of MPI and tolerate failures. The purpose of this paper is to evaluate the tradeoffs that are needed for the integration of this fault mitigation specification and its impact (or lack thereof) on MPI performance and scalability. The contributions of this work are to evaluate the difficulties faced by MPI implementors, and demonstrate the feasibility of a low-impact implementation on the failure-free performance as well as an estimate of the recovery time of the MPI state after a failure. This paper extends on the conference version [3] by presenting a more scalable implementation of the Revoke construct, and by deepening the performance analysis.

The remainder of this paper is organized as follows: the next section introduces a short history of fault tolerance in MPI; Sect. 3 presents the constructs introduced by the proposal; Sect. 4 discusses the challenges faced by MPI implementors; then the performance impact of the implementation in Open MPI is discussed in Sect. 5 before we conclude in Sect. 6.

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

2 http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

2 Related work

Efforts toward fault tolerance in MPI have previously been attempted. Automatic fault tolerance [7,8] is a compelling approach for users, as failures are completely masked and handled internally by the MPI library, which requires no new interfaces to MPI or application code changes. Unfortunately, many recent studies point out that automatic approaches, either based on checkpoints or replication, will exhibit poor efficiency on Exaflop platforms [5,6].

Application based fault tolerance (ABFT) [10,12,17] is another approach that promises better scalability, at the cost of significant algorithm and application changes. Despite some limited successes [4,15], MPI interfaces need to be extended to effectively support ABFT. The most notable past effort is FT-MPI [13]. Several recovery modes were available to the user. In the *Blank* mode, failed processes were replaced by `MPI_PROC_NULL`; messages to and from them were silently ignored and collective algorithms had to be significantly modified. In the *Replace* mode, faulty processes were replaced with new processes. In all cases, only `MPI_COMM_WORLD` would be repaired and the application was in charge of rebuilding any other communicators, leading to difficult library composition. No standardization effort was pursued, and it was mostly used as a playground for understanding the fundamental concepts.

A more recent effort to introduce failure handling mechanisms was the Run-Through Stabilization proposal [18]. 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, or using 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.

3 New MPI constructs

This section succinctly presents the prominent interfaces proposed to enable effective support of User-Level Failure Mitigation for MPI applications. For a more detailed description of the proposed interfaces, the interested reader should refer to the technical document [2] and to the amended standard draft.[3] This specification supports the permanent crash failure mode, where a process acts correctly until it stops (as the result of either a resource or a software error) and no subsequent results are delivered. Transient or byzantine errors are outside the scope of this work, but we will discuss in Sect. 4.2 how the common case of temporary network failures can be resolved.

Designing the mechanism that users would use to manage failures was built around three concepts: (1) simplicity, the API should be easy to understand and use in most common scenarios; (2) flexibility, the API should allow varied fault tolerant models to be built as external libraries and; (3) absence of deadlock, in every failure situation, the application can regain control and no MPI call (point-to-point or collective)

[3] http://svn.mpi-forum.org/trac/mpi-forum-web/ticket/323.

is blocked indefinitely after a failure, but must either succeed or raise an MPI error. Two major pitfalls must be avoided: 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 then that errors (`MPI_ERR_PROC_FAILED`) are not indicative of the return status on remote processes, but are raised only at a particular rank, when a particular operation cannot complete because a participating peer has failed.

When a failure occur, the state of MPI is left unchanged, only deliberate actions from the user alters the state of MPI regarding post-failure behavior. Obviously, communications involving a failed process (point-to-point or collective) will not be able to complete anymore, an error is raised by the completion of the operation, indicating that the operation could not complete as a consequence of process failures. However, the internal state of MPI objects is left unchanged; the communicator object (or window in the case of RMA) remains a valid object able to support further communications. New operations posted on the communicator may complete or raise errors, following the same semantic as before the first failure was reported, as described above. Notably, operations that do not involve the failed process can continue to operate normally, even on communicators that include failed processes (although most collective operations are expected to systematically result in an error).

Another important feature is that any error related to process failure has a local scope, it strictly indicates that the operation could not complete at the rank it has been reported. In the general case, the semantic information given by the error code is insufficient to infer which rank has failed, or if that same error has been raised at all ranks. Moreover, the operation may have successfully completed at other ranks, because the failure happened after the necessary internal steps of the operation completed, or because the failed process was not having an active role in the operation.

The proposed approach has the benefit of intruding the least with failure free operations, at the expense of a very relaxed consistency in the view of failures. The only strong guarantee is that for any communication, the operation will either complete with success, or raise an error (thereby ensuring that MPI programs can ensure the non-deadlock property). Such a minimalistic error semantic would prove challenging to overcome for many applications, as the MPI state being unchanged, errors would continue to be raised during the remainder of the execution. The following functions provide the basic blocks for restoring consistency and enabling recovery of the state of MPI, into a state where all past failures have been accounted for and resolved in a way that fits the application needs.

`MPI_COMM_REVOKE:` Because failure detection is not global to the communicator, some processes may raise an error for an operation, while others do not. This inconsistency in 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. Yet no failed process participates in the operation and it should not raise an error. The receive operation is effectively deadlocked. 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. It is notable that although this operation is not collective (a process can enter it alone), it affects remote ranks without a matching call.

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).

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 which will be used as a reference. The function to get the acknowledged failures refers back to this reference point and returns the group of processes which were locally known 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 (such as collective operations) will likely continue to raise errors.

4 Implementation issues

In this section, we detail the challenges and advantages of the aforementioned MPI constructs. They unfold along three main axis, the amount of supplementary state and memory to be kept within the MPI library, the additional operations to be executed on the critical path of communication routines, and the algorithmic cost of failure recovery routines. We discuss, in general, options available to implementors, and highlight issues with insight from a prototype implementation in Open MPI [14].

4.1 Impact on communication routines

Memory: Because a communicator cannot be repaired, tracking the state of failed processes imposes a minimal memory overhead. From a practical perspective each node needs a global list of detected failures, shared by all communicators; its size grows linearly with the number of failures, and it is empty as long as no failures occur. Within each communicator, the supplementary state is limited to two values: whether the communicator is revoked or not, and an index in the global list of failures denoting the last acknowledged failure (with MPI_COMM_FAILURE_ACK). For efficiency reasons, an implementation may decide to cache the fact that some failures have happened in the communicator so that collective operations and MPI_ANY_SOURCE receptions can bail out quickly. Overall, the supplementary memory consumption from fault tolerant constructs is small, independent of the total number of nodes, and unlikely to affect the cache and TLB hit rates. The size of messages sent on the network transport is unchanged (including the size of the message header).

Conditionals: Another concern is the number of supplementary conditions on 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.

Moreover, in order to cleanly release the resources in case of errors –an important feature to avoid wasting precious machine time on zombie applications– most MPI libraries, even those that are not fault tolerant, already include a conditional that checks for failures and errors. That same conditional can be repurposed, by extending the test condition, so that all error cases are handled without adding any supplementary condition to existing implementations.

Matching logic: MPI_COMM_REVOKE does not have a matching call on other processes on which it has an effect. As such, it might add detrimental complexity to the matching logic. However, any MPI implementation needs to handle unexpected messages. The order of revocation message delivery is loose enough that the handling of revocation notices can be integrated within the existing unexpected message matching logic. In our implementation in Open MPI, we leverage the active message low level transport layer to introduce revocation as a new active message tag, without a single change to the matching logic.

Collective operations: A typical MPI implementation supports a large number of collective algorithms, which might be dynamically selected depending on criteria such as communicator or message size and hardware topology. The loose requirements of the proposal concerning process failure error reporting limits the impact it has on collective operations. Typically, the collective communication algorithms and selection logic are left unchanged. The only new requirement is that failures happening at any rank of the communicator cause all processes to exit the collective (successfully for some, with an error for others). Due to the underlying loosely-connected topologies used by some algorithms, a point-to-point based implementation of a collective communication is unlikely to detect all process failures. Fortunately, a practical implementation exists that does not require modifying any of the collective operations: when a rank raises an error because of a process failure, it can revoke an internal, temporary communication context associated with the collective operation. As the revocation notice propagates on the internal communicator, it interrupts the point-to-point operations of the collective. An error code is returned to the high level MPI wrapper, which in turn raises the appropriate error on the user's communicator.

4.2 Failure detector

As stated earlier, it is not necessary to implement a total failure detector, or to inject jitter prone heartbeat messages for the proposed model to work. Indeed, as failures are reported only when an operation cannot complete, failures are triggered only when communicating directly with a failed processor. This approach match with the behavior of most network transport layers which report (at least asynchronously) communications and links errors when the peer of a communication cannot be reached. As an additional protection layer or for networks failing to exhibit the previous property, an independent TCP overlay error detector can be used to monitor the health of all peers actively communicating with. All other failures can be safely ignored, and there is no need to trigger active monitoring of these processes, thereby reducing greatly the latency and jitter generated by total failure detection. A notable exception is when using wildcard receives from `MPI_ANY_SOURCE`: because all ranks in the communicator are potential sources, a process stalled waiting on such an operation may have to trigger complete error detection on the communicator. In all other cases, errors are detected only on-necessity, without intrusive failure detection actions.

The case of network errors (or transient errors, where a process is unresponsive for a limited period of time before coming back online) can be handled at the implementation level. Once a process has been reported as failed to a particular process, this process will consistently ignore and discard further communications with this failed process. As a result the transient error is promoted to a fail-stop error. Because failure reporting is not consistent by default, it is already a possible outcome in a fail-stop only execution that some processes do not detect a failure as early as another one. The usual recovery routine (revoke and shrink) have the expected effect when a process invokes them to react to a promoted fail-stop failure: the transient failure is promoted as fail-stop for all processes on the communication object. It is to be noted that this behavior is a possible implementation that tackles in a sensible way transient failures, not a requirement of the specification.

4.3 Recovery routines

Some of the recovery routines described in Sect. 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. The remainder of this section describes the algorithms that can be used to deliver this specification and their cost.

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 [21]. The algorithm first performs a reduction of the input values to an elected coordinator in the communicator. The coordinator then makes a decision on the output value and broadcasts that value back to all of the alive processes in the communicator. The complexity of the agreement algorithm appears when adapting to an emerging process failure of the coordinator and/or participants. A more extensive discussion of the algorithmic complexity has been published by Hursey et al. [19]. The algorithmic complexity of this implementation is $O(log(n))$ for the failure free case, matching that of an `MPI_ALLREDUCE` operation over the alive processes in the communicator.

Revoke: Although the revoke operation is not collective, the revocation notification needs to be propagated to all alive processes in the specified communicator, even when new failures happen during the revoke propagation. These requirements are not without recalling those from the *reliable broadcast* [16]. Among the four defining qualities of a reliable broadcast (*Termination, Validity, Integrity, Agreement*), the termination and integrity criteria can be relaxed in the context of the revoke algorithm. If a failure during the Revoke algorithm kills the initiator as well as all the already notified processes, the Revoke notification is indeed lost, but the observed behavior, from the view of the application, is indiscernible from a failure at the initiator before the propagation started. As the algorithm still ensures agreement, there are no opportunities for inconsistent views.

In the ULFM implementation, we use a Binomial Graph (BMG) [1]. The initiator marks the communicator as revoked, and sends a Revoke message to $log(n)$ other processes (in a communicator of size n). Upon the reception of a revoke message, if the communicator is not already revoked, it is then revoked and the process acts as a new initiator. This topology has the advantage of featuring a lower message count ($O(nlog(n))$ messages are exchanged, instead of $O(n^2)$). Because the revoke algorithm is asynchronous, the performance benefit, as observed by applications, is low. However, complete flooding was suffering from a practical problem at scale: it required opening a large number of connections, resulting in crashing the Open IB driver. The BMG topology is protected against that defect, because in this topology, the degree per node is $O(log(n))$; in the experiments, no crashes were observed. The converse drawback is that if $log(n)$ simultaneous failures strike in an exact pattern, and the failure had been detected by only one process, the revoke algorithm could fail to deliver the notification to some alive processes. The probability of such an outcome is very low (it is a generalization of the birthday problem for $log(n)$ simultaneous birthdays), and we are confident that it will never be observed in practice.

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

The following analysis used a prototype of the ULFM proposal based on the development trunk of Open MPI [14] (r26237). The test results presented were gathered from

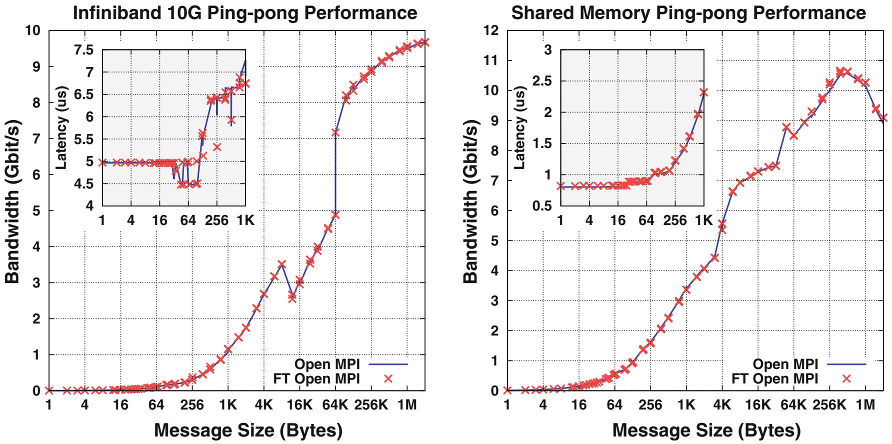

Fig. 1 Netpipe latency and bandwidth impact of enabling fault tolerance support (on smoky)

Table 1 Standard deviation of the NetPIPE results (on smoky)

Interconnect	Vanilla	Std. Dev.	FT	Std. Dev.	Difference
1-Byte latency (microseconds) (cache hot)					
Shared memory	0.8008	0.0093	0.8016	0.0161	0.0008
TCP	10.2564	0.0946	10.2776	0.1065	0.0212
OpenIB	4.9637	0.0018	4.9650	0.0022	0.0013
Bandwidth (Mbps) (cache hot)					
Shared memory	10,625.92	23.46	10,602.68	30.73	−23.24
TCP	6,311.38	14.42	6,302.75	10.72	−8.63
OpenIB	9,688.85	3.29	9,689.13	3.77	0.28

the Smoky system at Oak Ridge National Laboratory. Each node contains four quad-core 2.0 GHz AMD Opteron processors with 2 GB of memory per compute core. Compute nodes are connected with gigabit Ethernet and InfiniBand. Some shared-memory benchmarks were conducted on Romulus, a 6×8-core AMD Opteron 6180 SE with 256 GB of memory (32 GB per socket) at the University of Tennessee.

The NetPIPE benchmark (v3.7) was used to assess the 1-byte latency and bandwidth impact of the modifications necessary for the ULFM support in Open MPI. We compare the vanilla version of Open MPI (r26237) with the ULFM enabled version on smoky (labelled as FT). Figure 1 compares the bandwidth and latency achieved by these two builds of Open MPI. As can be observed, for the entire range of message sizes, the performance difference is insignificant, either when using the Infiniband network transport, or with the shared-memory interface, which is most susceptible to latency. Table 1 presents the standard deviation across 100 runs, and further highlights the fact that the differences in performance are not only well below the noise limit, but that the standard deviation difference is negligible, thus proving the performance stability and lack of impact.

The impact on shared memory systems, which are sensitive even to small modifications of the MPI library, has been further assessed on the Romulus machine—a

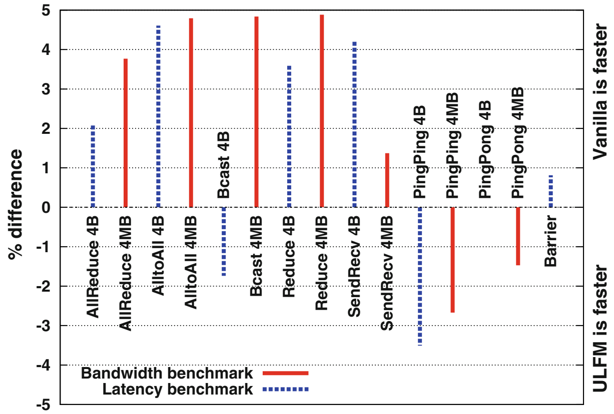

Fig. 2 The intel MPI benchmarks: relative difference between ULFM and the vanilla Open MPI on shared memory (Romulus). Standard deviation ≈5 % on 1,000 runs

large shared memory machine—using the IMB benchmark suite (v3.2.3). As shown in Fig. 2, the difference on all the benchmarks (point-to-point and collective) remains below 5 %, within the standard deviation of the implementation on that machine.

To measure the impact of the prototype on a real application, we used the Sequoia AMG benchmark.[4] This MPI intensive benchmark is 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*. In Fig. 3, we compare the time slicing of three main phases (Solve, Setup, and SStruct) of the benchmark, with, side by side, the vanilla version of the Open MPI implementation, and the ULFM enabled one. The application itself is not fault tolerant and does not use the features proposed in ULFM. The goal of this benchmark is to demonstrate that a careful implementation of the proposed semantic does not impact the performance of the MPI implementation, and ultimately leaves the behavior and performance of legacy applications unchanged. The results show that the performance difference is negligible.

To assess the overheads of recovery constructs, we developed a synthetic benchmark that mimics the behavior of a typical fixed-size tightly-coupled fault-tolerant application. Unlike a normal application it performs an infinite loop, where each iteration contains a failure and the corresponding recovery procedure. Each iteration consists of 5 phases: in the first phase (*Detection*), all processes but a designated victim enter a Barrier on the intracommunicator. The victim dies, and the failure detection mechanism makes all surviving processes exit the Barrier, some with an error code. In Phase 2 (*Revoke*), the surviving processes that detected a process-failure related error during the previous phase invoke the new construct `MPI_COMM_REVOKE`. Then they proceed to Phase 3 (*Shrink*) where the intracommunicator is shrunk using `MPI_COMM_SHRINK`. The two other phases serve to repair a full-size intracommunicator using MPI-2 spawn and intercommunicator merge operations to allow the benchmark to proceed to the next round.

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

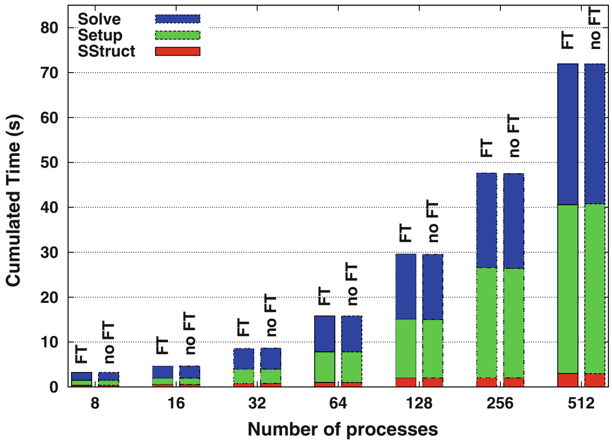

Fig. 3 Comparison of the vanilla and ULFM versions of Open MPI running Sequoia-AMG on smoky

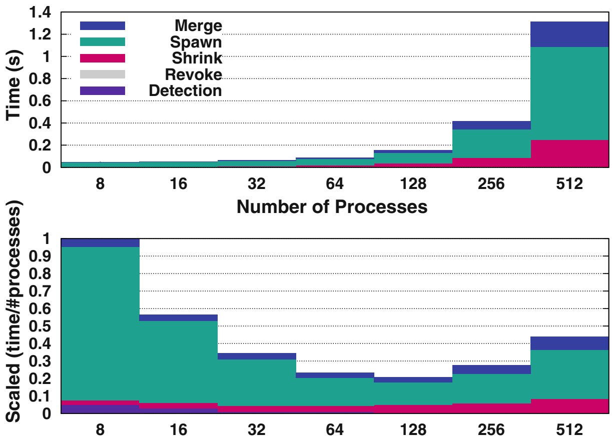

Fig. 4 Fault injection benchmark with full recovery on smoky. The *top graph* presents absolute times, the *bottom graph* presents the same results divided by the number of processes, to highlights scalability trends

In Fig. 4, we present the timing of each phase, averaged upon 50 iterations of the benchmark loop, for a varying number of processes on the smoky machine. The scaled graph presents the same result, but scaled down accordingly to the number of processors used; the resulting normalized unit is irrelevant, but improves readability for small deployments and better highlight scalability trends.

The failure detection is mildly impacted by the scale. In the prototype implementation, the detection happens at two levels, either in the runtime system or in the MPI library (when it occurs on an active link). Between the two detectors, all ranks get notified within 30ms of the failure (this compares to the 1s timeout at the link level).

Although the revoke call injects a logarithmic number of messages in the network to implement the level of reliability required for this operation, the duration of this call itself is under 50 μs and is not visible in the figure. The network is disturbed for a longer period, due to the processing of the messages, but this disturbance will appear in the network only after a failure occurred. According to the performance of the next operation (shrink), this disturbance has no practical consequences.

The duration of the new construct to shrink a communicator behaves similarly to other communicator manipulation operations (as illustrated by comparing with the intercomm merge operation). Indeed, the shrink operation includes two operations, one is the agreement on the group of failed processes, the second one is the allocation of a new communicator identifier (an operation that also appears in intercomm merge). Because in this benchmark, no new failure disrupts the (shrink-internal) agreement operation, it completes in the same time as a regular collective communication would. Consequently, a significant portion of the time of the Shrink operation is spent in the underlying communicator creation functions (unmodified from MPI-2).

The Spawn operation, directly inherited from MPI-2, and left unmodified in the ULFM prototype, exhibits poor performance and scalability. The reason is mostly historical: `MPI_COMM_SPAWN` has seen little use in the past, and thereby has not been the focus of intensive optimizations from implementors. Should the use of this construct become more ubiquitous, it is expected that a careful implementation would reach adequate performance, for it is not prevented by theoretical difficulties.

An interesting observation is that all three operations shrink, spawn, an merge pay for the cost of the allocation of a communicator identifier; an overhead that appears to be significant at scale. This suggests that the ULFM specification could benefit from the addition of an operation realizing these three operations at once, thereby dividing this overhead by three.

6 Conclusion

Many responsible voices agree that sharp increases in the volatility of future, extreme scale computing platforms are likely to jeopardize our ability to use them to maximize the research productivity of advanced long-running applications that deliver meaningful scientific results. Moreover, it become clear that any lightweight techniques developed to address this volatility must be supported in the programming and execution model. 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, MPI 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. It not only delivers support for a number of new and innovative resilience techniques, it provides this support through a simple, straightforward and familiar API that requires minimal modifications of the underlying MPI implementation. It maintains the backward compatibility with previous versions of the MPI standard, so that non fault-tolerant applications (legacy or otherwise) are supported without any changes to the code. At the same time it provided a low-level portable API, that will allow other fault tolerant approaches to be implemented as libraries. Perhaps most significantly, applications can use ULFM-enabled MPI without experiencing any degradation in their performance, as we demonstrate in this paper.

Several applications, ranging from Master-Worker to tightly coupled, are currently being refactored to take advantage of the semantics in this proposal. Beyond applications, the expressivity of this proposal is investigated in the context of providing extended fault tolerance models as convenience, portable libraries.

References

1. Angskun T, Bosilca G, Dongarra J (2007) Binomial graph: a scalable and faulttolerant logical network topology. In: ISPA07. Number 4742 in LNCS, Springer, pp 471–482
2. Bland W, Bosilca G, Bouteiller A, Herault T, Dongarra J (2012) A proposal for user-level failure Mitigation in the MPI-3 standard. Department of Electrical Engineering and Computer Science, University of Tennessee
3. Bland W, Bouteiller A, Herault T, Hursey J, Bosilca G, Dongarra JJ (2012) An evaluation of user-level failure mitigation support in MPI. In: Träff JL, Benkner S, Dongarra JJ (eds) EuroMPI, Lecture Notes in Computer Science, vol 7490, Springer, pp 193–203
4. Bland W, Du P, Bouteiller A, Herault T, Bosilca G, Dongarra JJ (2012) A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI. In: 18th Euro-Par, LNCS, vol 7484, Springer, pp 477–489
5. Bosilca G, Bouteiller A, Brunet É, Cappello F, Dongarra J, Guermouche A, Herault T, Robert Y, Vivien F, Zaidouni D (2012) Unified model for assessing checkpointing protocols at extreme-scale. Tech. report RR-7950, INRIA
6. Bougeret M, Casanova H, Robert Y, Vivien F, Zaidouni D (2012) Using group replication for resilience on exascale systems. Tech. Rep. 265, LAWNs
7. Bouteiller A, Bosilca G, Dongarra J (2010) Redesigning the message logging model for high performance. CCPE 22(16):2196–2211
8. Buntinas D, Coti C, Herault T, Lemarinier P, Pilard L, Rezmerita A, Rodriguez E, Cappello F (2008) Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. FGCS 24(1):73–84. doi:10.1016/j.future.2007.02.002
9. Cappello F, Geist A, Gropp B, Kalé LV, Kramer B, Snir M (2009) Toward exascale resilience. IJHPCA 23(4):374–388
10. Davies T, Karlsson C, Liu H, Ding C, Chen Z (2011) High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: 25th ICS, ACM, pp 162–171
11. Dongarra J, Beckman P et al (2011) The international exascale software roadmap. IJHPCA 25(11):3–60
12. Du P, Bouteiller A et al (2012) Algorithm-based Fault Tolerance for dense matrix factorizations. In: 17th SIGPLAN PPoPP, ACM, pp 225–234
13. Fagg G, Dongarra J (2000) FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In: 7th EuroPVM/MPI, LNCS, vol 1908, Springer, pp 346–353
14. Gabriel E et al (2004) Open MPI: Goals, concept, and design of a next generation MPI implementation. In: 11th EuroPVM/MPI, LNCS, vol 3241, Springer, pp 353–377
15. Gropp W, Lusk E (2004) Fault tolerance in Message Passing Interface programs. IJHPCA 18:363–372. doi:10.1177/1094342004046045
16. Hadzilacos V, Toueg S (1993) Distributed systems (2nd edn). In: Fault-tolerant broadcasts and related problems, ACM/Addison-Wesley, pp 97–145
17. Huang K, Abraham J (1984) Algorithm-based Fault Tolerance for matrix operations. IEEE Trans Comput 100(6):518–528
18. Hursey J, Graham RL, Bronevetsky G, Buntinas D, Pritchard H, Solt DG (2011) Run-through stabilization: an MPI proposal for process fault tolerance. In: 18th EuroMPI, LNCS, vol 6690, Springer, pp 329–332
19. Hursey J, Naughton T, Vallee G, Graham RL (2011) A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In: 18th EuroMPI, LNCS, vol 6690, Springer, pp 255–263
20. Lusk E, Chan A (2008) Early experiments with the OpenMP/MPI hybrid programming model. In: 4th IWOMP, LNCS, vol 5004, Springer, pp 36–47

21. Mohan C, Lindsay B (1985) Efficient commit protocols for the tree of processes model of distributed transactions. In: SIGOPS OSR, vol 19, ACM, pp 40–52
22. Sterling T (2011) HPC in phase change: towards a new execution model. In: HPCCS-VECPAR 2010, LNCS, vol 6449, Springer, pp 31–31