This article has been accepted for publications in: Future Generation Computer Systems. Content may change prior to final publications. Citation information: https://doi.org/10.1016/j.future.2020.01.026

Fault Tolerance of MPI Applications in Exascale Systems: The ULFM Solution

Nuria Losada[a], Patricia González[b], María J. Martín[b], George Bosilca[a], Aurélien Bouteiller[a], Keita Teranishi[c]

[a]{nlosada, bosilca, bouteill}@icl.utk.edu
Innovative Computing Laboratory, the University of Tennessee, Knoxville, USA
[b]{patricia.gonzalez, mariam}@udc.es
Computer Architecture Group, CITIC, University of A Coruña, Spain
[c]knteran@sandia.gov
Sandia National Laboratories, Livermore, USA

Abstract

The growth in the number of computational resources used by high-performance computing (HPC) systems leads to an increase in failure rates. Fault-tolerant techniques will become essential for long-running applications executing in future exascale systems, not only to ensure the completion of their execution in these systems but also to improve their energy consumption. Although the Message Passing Interface (MPI) is the most popular programming model for distributed-memory HPC systems, as of now, it does not provide any fault-tolerant construct for users to handle failures. Thus, the recovery procedure is postponed until the application is aborted and re-spawned. The proposal of the User Level Failure Mitigation (ULFM) interface in the MPI forum provides new opportunities in this field, enabling the implementation of resilient MPI applications, system runtimes, and programming language constructs able to detect and react to failures without aborting their execution. This paper presents a global overview of the resilience interfaces provided by the ULFM specification, covers archetypal usage patterns and building blocks, and surveys the wide variety of application-driven solutions that have exploited them in recent years. The large and varied number of approaches in the literature proves that ULFM provides the necessary flexibility to implement efficient fault-tolerant MPI applications. All the proposed solutions are based on application-driven recovery mechanisms, which allows reducing the overhead and obtaining the required level of efficiency needed in the future exascale platforms.

Keywords:
MPI, Resilience, ULFM, Application-level Checkpointing

1. Introduction

Many scientific computing fields rely on HPC and its supercomputers for solving their most challenging problems. Today these machines provide high computational power, in the order of 10^{15} floating-point operations per second, enabling the resolution of large scientific, engineering, and analytic problems. However, the computational demands of state-of-the-art science grows driven by two major factors: new problems for which the resolution time is critical (such as the design of personalized pharmaceutical drugs, in which patients cannot wait years for the specific molecule they need), and the exponential growth in the amount of data that must be processed (for instance, data generated by

large telescopes, particle accelerators and detectors, social networks, or smart cities sensors).

The exascale era is expected to be reached in the near future using supercomputers comprised of millions of cores and able to perform 10^{18} operations per second. This is a great opportunity for HPC applications; however, it is also a hazard for the completion of their execution. Recent studies show that, as HPC systems continue to grow larger, the mean time to failure for a given application also shrinks, resulting in a high failure rate overall. Even if one compute node presents a failure every one century, a machine with 100,000 nodes will encounter a failure every 9 hours on average [18]. More alarming, a machine built with 1,000,000 of those nodes will be hit by a failure every 53 minutes on average, an execution time too short for most scientific applications to deliver meaningful results.

But the completion or correctness of applications' execution is not the only challenge raised by a decreasing mean time to failure. Di Martino et al. [17] studied the failure behavior of the Blue Waters Cray supercomputer, reporting that failed applications noticeably run for about 9% of the total production node hours. The electricity cost of not using any fault tolerance mechanism in the failed applications was estimated at almost half a million dollars during the studied period of time (261 days). Therefore, the efficient exploitation of HPC resources for long-running applications will need to rely on fault tolerance techniques, not only to ensure the completion of their execution in exascale systems but also to guarantee correctness and save energy.

The Message Passing Interface (MPI) is the de facto standard for programming HPC parallel applications in distributed-memory architectures. However, the current MPI standard and its implementations lack fault tolerance support, and the default behavior, in the event of a failure, consists of aborting the execution of the application. This is the reason why, traditionally, MPI failures are addressed with stop & restart checkpointing solutions, techniques where each process in the application periodically saves its state to stable storage into checkpoint files. In case of a failure, the application is restarted from one of the intermediate states of execution once it is re-spawned. However, in large parallel systems, failures frequently have a limited impact and affect only a subset of the cores or computation nodes used by the application. Under these circumstances, a complete cancellation of the MPI application followed by a full restart yields unnecessary overheads and particularly stresses the parallel file system.

The User Level Failure Mitigation (ULFM) interface [6], under discussion in the MPI Forum, proposes the inclusion of resilient capabilities in the MPI standard. ULFM includes new semantics for process failure detection, communicator revocation, and reconfiguration—that is, what is needed to repair the communication capabilities. These new functionalities provide the minimal set of features necessary to deliver resilience support, without imposing a strict recovery model. Therefore, it does not include any specialized, non-portable mechanism to recover the application state at failed processes, providing developers of applications or higher-level frameworks the flexibility to implement the most optimal methodology, taking into account the properties of the target application or domain.

The main purpose of this paper is to summarize recent experiences in MPI applications' fault tolerance using the ULFM specification, pointing out open issues and challenges for the exascale era with a focus on application's perspective. The paper includes:

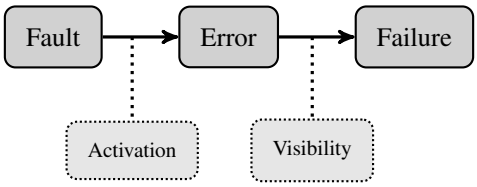


Figure 1: Relation between faults, errors, and failures.

- A review of the capabilities of ULFM, the most engaged project towards the incorporation of fault tolerant support into the MPI standard.
- The analysis of its potential through an exhaustive review of the different ULFM application-level solutions present in the literature to implement resilient applications.

For illustrative purposes, this paper also includes a comparison of the resilient approaches with the traditional stop & restart solutions to depict the performance benefits that can be obtained exploiting the ULFM constructs in fault tolerance techniques.

The structure of the paper is as follows. Section 2 focuses on the characterization and classification of faults to outline the scope of this work. Section 3 summarizes related work that, for the most part, deals with ULFM alternative fault tolerant frameworks for MPI applications. Section 4 describes the ULFM interface. A review of recent application-level resilience solutions using the ULFM functionalities is covered in Section 5, while Section 6 compares the resilient solution versus the traditional stop and restart approach. Finally, Section 7 concludes this paper.

2. Fault Tolerance Coverage

The terminology used in this paper follows the taxonomy of Avižienis and others [4, 13, 46], summarized in Figure 1. Faults (e.g., a physical defect in the hardware) can cause system errors, that is, systems incorrect states. Errors may propagate and lead to failures when they cause the incorrect service of the system—in other words, an incorrect system's functionality and/or performance that can be externally perceived. Faults can be active or inactive, depending on whether or not they cause errors; and permanent or transient, depending on whether or not their presence is continuous in time.

Hardware faults correspond to physical faults (i.e., permanent or transient faults in any of the components of the system) and can result in: (1) Detectable Correctable Error (DCE), (2) Detectable Uncorrectable Error (DUE), and (3) Silent Error (SE) or Silent Data Corruption (SDC). DCEs are managed by hardware mechanisms such as error correcting codes (ECCs), parity checks, and Chipkill-Correct ECCs, and are oblivious to the applications. DUEs can lead to the interruption of the execution, while SDCs can lead to a scenario in which the application does not experience a runtime failure yet returns incorrect results, and the user might not be aware of it.

Software faults can be classified as: (1) pure software errors, (2) hardware problems mishandled by software, and (3) software causing a hardware problem. Pure software errors correspond to classical correctness issues (such as

incorrect control flows), concurrency errors (concurrent code is hard to develop and debug), and performance errors (originated by resource exhaustion that can lead to actual crashes due to timeouts). Examples of the second category include node faults not being handled by software at other nodes, or a disk fault causing a file system failure. Finally, software can trigger an unusual usage pattern for the hardware, causing the manifestation of hardware errors.

Resilience is defined as the collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults. The literature covered in this paper focuses on faults that cause process fail-stop failures in distributed applications—that is, failures derived from hardware and software errors that have the direct, drastic, impact of unexpectedly and permanently rendering non-responsive some of the application processes. The goal of the considered techniques is thus to mitigate the effect of process failures, so that, from the application perspective, a process failure remains at the level of an application error because the software infrastructure prevents the escalation to an application failure.

3. Related Work

Currently, ULFM is the most active project towards integrating fault tolerant support into the MPI standard. Previous to the ULFM project, other attempts such as MPI-FT [37], MPI/FT [5], FT-MPI [20], or FEMPI [48], had started this important research path. Though none of them have been adopted into the MPI standard, and nowadays they are no longer maintained, they certainly deserve credit as the seed for many other projects in the field.

Several recent research studies focused on alternatives to ULFM, making use of layers outside the MPI library itself, to avoid the dependence on the communication pattern, and, thus, accelerating the detection of the failures. Some of these works have a basis in Laguna et al. [30], which proposes the conceptual interface of *Reinit*. A significant difference between this proposal and those based on ULFM is the assumption of a fault detector within the target system (MPI and/or runtime environment, or some additional system services). Conceptually, the *Reinit* model assumes that in case of a failure, all MPI processes reinitialize themselves, in the same state they have been after the initial call to `MPI_Init`, strongly limiting the flexibility of the recovery procedure users may implement, as all original processes must participate. Figure 2 illustrates this approach. This behavior can be assimilated to a global synchronous restart of the entire parallel application and provides realistic support only for tightly coupled applications that belong to the bulk synchronous parallel (BSP) type. In [32] an initial prototype of Reinit is described, as well as a qualitative comparison with ULFM. However, a performance comparison could not be provided due to the lack of a full real-world implementation of the Reinit interface.

In a more recent work, Chakraborty et al. [15] use the Reinit's interface in terms of technical design and implementation, and proposes EReinit (efficient Reinit), a more scalable implementation that inherits the same type of restrictions as Reinit. The authors present scalability results comparing EReinit with Fenix [23], a fault tolerance approach based on ULFM described in Section 5.2. The main difference between these approaches is that, instead of implementing the recovery functions on top of MPI, using ULFM, EReinit aims to co-design them between MPI

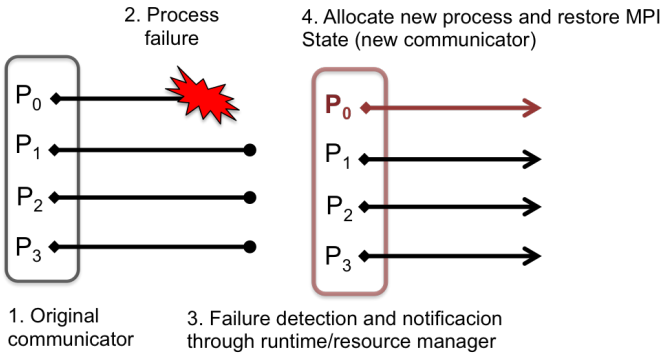


Figure 2: Resilient solution at low-level layers. After a failure, processes are warned by the runtime framework or the resource manager, they stop their execution, new processes are allocated to replace the failed ones and, finally, all the processes perform the initialization phase to create a new communicator.

and the resource managers. The experimental results show that the EReinit implementation is faster than the Fenix approach for detecting failures and reinitializing the execution, especially when dealing with concurrent failures.

Emani et al. [19] propose *Checkpointable MPI*, also based on the roots of Reinit, in which the MPI state is saved in a checkpoint along with the application state and restored in case of failure. In Checkpointable MPI the core functionalities of failure detection, notification, and spawning of replacement processes are performed in layers below MPI, using the Process Management Interface for Exascale (PMIx) [14] and Slurm [54] as a resource manager.

NR-MPI [49] is a resilient MPI solution built on top of MPICH that implements the semantics derived from the FT-MPI project. Recently, a new proposal for NR-MPI [50] implements some fault tolerance semantics of ULFM based on MPICH. However, it relies on an external failure detector (assumed integrated into the resource manager) instead of detecting the failures in the MPI library, under the assumption that an external detector might reduce the overhead of failure-free executions.

Other resilient approaches explore the use of alternative programming solutions. Fault-Aware MPI (FA-MPI) [27, 26] provides a set of extension APIs for MPI to support a lightweight transactional model for fault-awareness. However, it is restricted to non-blocking MPI communication operations. It introduces transactions around user-specified code blocks; thus, failures are not detected nor recovered in each failed MPI communication operation (in contrast to ULFM). The granularity of fault-awareness, and, thus, the associated overhead, is configurable in FA-MPI through transaction duration and length and hierarchically nesting transactions. Likewise, Fault Tolerance Assistant MPI (FTA-MPI) [22] is a programming model that exploits a try/catch exception-handling syntax to enable failure detection and transparent recovery in MPI applications. Although more versatile than FA-MPI, since it allows blocking MPI calls, FTA-MPI also detects and repairs failures within a user-defined code block. Using FTA-MPI a *conversation* is declared in a try-catch code block, so that at the end of the conversation, all participants can detect a failure (if it occurred). In such a case, FTA-MPI automatically recovers the application-level state (by means of a checkpoint)

and MPI-level state (by repairing the communicators). The granularity of the failure detection is a *conversation*.

A different approach is featured by FMI [44], a prototype programming model providing a similar semantic to MPI that issues fault tolerance, including checkpointing application state, restarting failed processes, and allocating additional nodes as necessary. In contrast to the solution proposed by ULFM, FMI acts as an isolation layer and applications are unconscious of failures. The FMI prototype demonstrates encouraging results, achieving a very low-latency recovery by means of a survivable communication runtime coupled with fast, in-memory checkpoint/restart, and dynamic node allocation. However, the current prototype implementation only supports a subset of MPI functions. Among the missing capabilities, two stand out due to their impact on the application scalability and efficiency: collective functions and communicator creation via split. Collective functions increase their impact at large scale. Split of communicators is often dynamically used in order to balance the workload in many applications. Thus, the efficient support of these functions will be key in FMI future plans. Similarly, in MPICH-V [11], coordinated checkpointing and uncoordinated checkpointing were deployed within the MPI infrastructure in order to capture the state of the application, and rollback to a checkpoint in case of failure in an automatic, transparent manner. The communication overhead of this solution is demonstrated to be acceptable; however, the automatic placement of checkpoints without regard for the application's structure, and the system-level process restart prevents a wide range of checkpointing optimizations resulting in a relatively high cost of checkpointing activities when compared to non-automatic solutions.

Recently, high-level HPC programming systems, such as Partitioned Global Address Space (PGAS) languages like Coarray Fortran (CAF) [40] and X10 [16], are gaining popularity in production applications. They usually rely on high-performance transport layers, such as MPI, to achieve low communication latency, portability, and scalability on large-scale systems. In this context, several attempts have been already made to provide fault tolerance to these high-level systems using ULFM. Hamouda et al. [25] describe the use of ULFM to achieve an efficient transport layer for Resilient X10. In [21] the failed images CAF feature were implemented using ULFM in the MPI-based version of OpenCoarrays.

4. The ULFM interface

Applications exhibit a wide variety of pre- and post-failure behaviors, where the needs for the recovery procedure range from applications that perform only point-to-point communications with a small set of close neighbors, to applications that routinely perform collective communication. Unsurprisingly, this wide variety of communication patterns demand a diversity of recovery strategies. Among their key differences are (1) how many processes are involved in managing a failure and its consequences, (2) what the expectations are in terms of restoring the mapping of processes and data onto the physical resources (i.e., the difference between malleable jobs which can adapt on the fly to a changing deployment topology, and inflexible jobs for which the data distribution and process mapping have to adhere to some predefined rules, like a cartesian grid), and (3) how the data is to be restored after the process failure

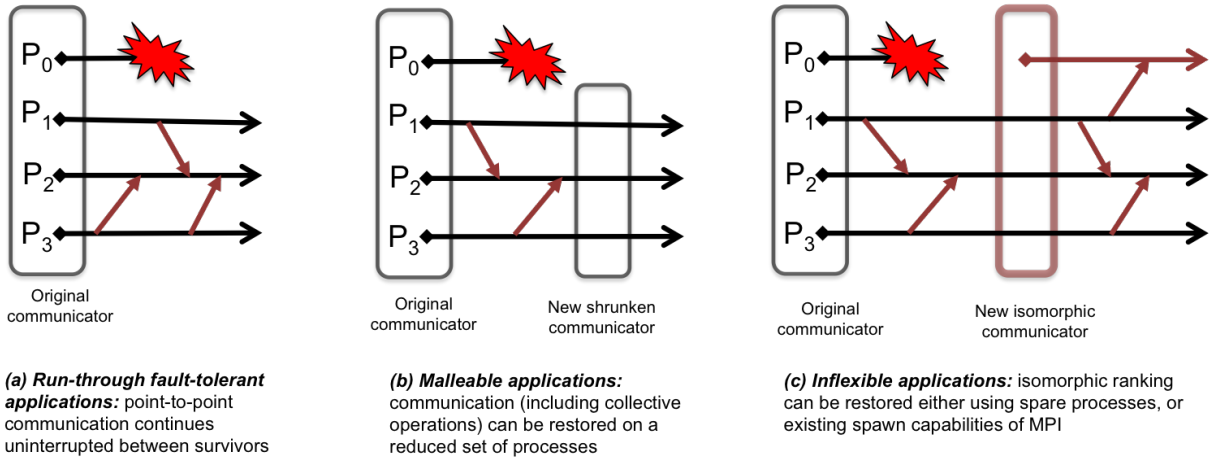


Figure 3: Run-through, malleable and inflexible applications have different needs in terms of restoring communication capabilities.

expunged part of the dataset (e.g., from a checkpoint, by interpolating neighboring data, with additional iterations). Each application is likely to exhibit an original combination of these, and many other, criteria. The ULFM interface thus provides a generic method to restore MPI's communication infrastructure and capabilities after a failure, but it does not concern itself with providing a strategy for data restoration. This rejuvenated communication capability can then be employed at the application's leisure to perform state introspection and dataset restorative actions, like communicating checkpoint data, using normal MPI primitives.

We identify three features an application may require to restore its communication capabilities: failure notification, failure propagation and interruption, and communication context restoration. Different types of applications, such as run-through, malleable, and inflexible applications illustrated in Figure 3, may choose to use a subset, or all of those features to tolerate failures, depending upon the intended usage pattern. Next subsections describe in detail the basic interfaces of the ULFM interface that provide these features, and Subsection 4.4 provides some examples of their use.

4.1. Failure Notification

Since ULFM does not mask process failures to the application, it needs to provide mechanisms to produce actionable application errors upon their occurrence. ULFM includes new error codes denoting the occurrence of events related to process failures, which are summarized in Table 1. The ULFM specification follows closely the preexisting conventions from the MPI standard and employs the long-existing concept of MPI error handlers to report errors to the application. Using the `MPI_Comm_set_errhandler` provided by MPI, the user can specify whether errors codes should be returned to the application (`MPI_ERRORS_RETURN`) or whether a user-defined error handler procedure should be invoked.

By default, ULFM reports an error only for operations whose semantic cannot be fulfilled because of the failure (i.e., a fault that could result in an observable defect). A notable example is when the communication operation involves a failed process as a peer (as a source, destination, or a member of the group on which the operation is

Error code	MPI calls involved	Description
MPIX_ERR_PROC_FAILED	Blocking operations & completion functions.	The operation involved a dead process.
MPIX_ERR_PROC_FAILED_PENDING	Non-blocking MPI_ANY_SOURCE.	A potential sender has been discovered dead.
MPIX_ERR_REVOKED	All MPI routines but shrinking & agreement.	Communicator marked as improper for further communication.

Table 1: ULFM specific error codes.

collective). Consider a point to point communication: if the destination process specified in a send operation or the source process in a receive operation have failed, the operation cannot be completed anymore and needs to report an appropriate error to the caller. For collective communications, a process reporting any type of error, including process failures, does not imply that the collective operation will complete with the same error at all participants. For example, in a broadcast that follows a tree topology, all processes in the subtree rooted at a failed process will report an error, while the rest of the processes may complete the operation successfully (given that they have fulfilled the local semantic of the operation and that the message is locally available in the reception buffer). In the case of non-blocking communications—point to point or collective—the same rationale applies, but raising errors is postponed until the corresponding completion routine is invoked.

Note that the fact that the error notification is restricted only to those operations whose semantic cannot be fulfilled avoids imposing limits to the wide variety of post-failure behaviors that programmers may choose to deploy. Operations that do not directly involve failed processes will complete normally, which enables the implementation of run-through failures strategies (exemplified in Figure 3). Consider the case of a master-worker type of workload, in which a master dispatches work to worker processes. In this context, it is clear that the failure of an independent worker is not an important event from the perspective of another independent worker, and only the master process should be informed that a failure occurred so as to dispatch the work to another worker. Disturbing the communication of independent processes simply adds noise and complexity to the design of simple resilient patterns.

Unnamed receptions (i.e., operations using the MPI_ANY_SOURCE peer) add a bit of extra complexity and require special handling due to the message delivery order imposed by MPI (for any pair of processes the message delivery order matches the sending order.) It is not possible for the MPI implementation to determine with certainty whether the operation would block infinitely when a potential sender has failed; thus, in ULFM, such operations are also interrupted with a special error code. The operation MPIX_Comm_failure_ack enables users to acknowledge all locally notified failures in the communication context.[1] When using unnamed communications, this routine pro-

[1] A communication context can be a communicator, window or file. We will use communicator and communication context interchangeably without loss of generality.

vides the application a way to resume any-source operations, as long as the list of failed processes does not change. `MPIX_Comm_failure_ack` can be used in tandem with `MPIX_Comm_failure_get_acked` to introspect the current state of the MPI processes, and build the list of failed processes.

4.2. Failure Propagation and Interruption

Another feature provided by ULFM for the recovery of the communication infrastructure is error propagation and interruption. Obviously, not all applications follow the master-worker paradigm, and it is likely that most applications will need a more tightly coupled global recovery strategy. One possible programming technique for resilient distributed applications is to progress in macro-steps, or transactions, that have to be validated before the program moves forward to the next step. The routine `MPIX_Comm_agree` delivers a resilient operation that performs consensus on the knowledge about faults. It provides a reliable reduce-like operation (similar to an `MPI_Allreduce`) on a synchronization variable, however, `MPIX_Comm_agree` guarantees uniform failure reporting across the participating peers. Typical use cases include validating the success or failure of one or more iterations during an iterative process, or of a collective operation, or more generally validating the commit of checkpoint files. The agreement operation permits programmatic reasoning on the progress of the application and the explicit propagation of error condition in a structured manner. Therefore, it can be used for global error detection in a given communication context for some applications. However, the agree operation has a higher cost than an `MPI_Allreduce` operation and should, therefore, be used in a parsimonious manner. The current state-of-the-art agreement algorithm [28] has a complexity of $2\log_\delta n$ parallel steps if no failures happen and at most $O(2\log_\delta n + f\delta)$ parallel steps if f failures happen, with δ being the maximum degree of the tree defined by the *Parent*/*Children* functions, and n the number of nodes in the tree. And although this strategy for error propagation may be sufficient in some applications, it requires that all processes not affected by the failure reach the agreement operation, which might not always happen if the application is not reasonably well tightly coupled. In some applications with a neighborhood communication pattern, a failure in a group may remain unreported to processes in other groups, and it thus could be necessary to interrupt that ongoing communication pattern before all processes can proceed with the recovery procedure. Consider the case presented in Figure 4 of a chain of processes moving a token from the lowest rank to the highest rank. In such a program, all processes except 0 have posted a named reception to obtain the token from their predecessor in the chain. When a failure occurs, the immediate successor of a failed process will receive an error from the posted reception operation. However, the remaining processes are still waiting on a reception from a live process, and have no reason yet to interrupt that call. Nevertheless, if the remainder of the recovery procedure requires these processes to participate, they need to be released from the "failed communication pattern" where they expect a message from a predecessor that will not send it anymore. This can be done programmatically by sending all the remaining messages, but this approach is cumbersome, error-prone, and inefficient. The ULFM interface provides the explicit revocation call `MPIX_Comm_revoke` as a better alternative to prevent "failed communication pattern" deadlocks. When a process calls this function, the operation triggers an error at all ranks in the communication context and invalidates that communicator for further

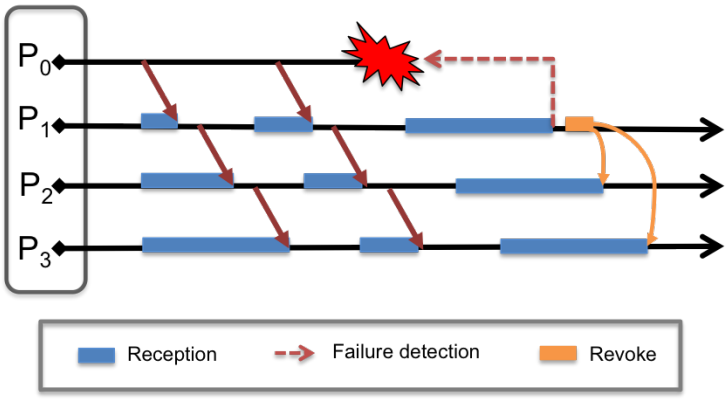


Figure 4: A transitive dependence in the communication can be interrupted with the explicit error propagation call `MPIX_Comm_revoke` when the recovery of the application requires global failure notification in the communication context.

communication, so that pending and future API calls on this communicator will return an error code without blocking. Thus, this operation effectively achieves the propagation of errors under the exclusive control of the application.

4.3. Communication Context Restoration

Some applications, such as malleable and inflexible applications illustrated in Figure 3, need to restore a fully functional communication context. This implies not only being able to successfully perform point-to-point operations between live processes, but also restoring the capability to carry out collective communications. To that end, all failed processes need to be entirely removed from the group of processes in the communication context. The interface `MPIX_Comm_shrink` produces a new communicator in which all processes that are known to have failed before (or during) the operation are expunged. This new communicator is thus a fully functional communicator in which all processes are presumed live (barring the occurrence of supplementary failures). Collective communication can be employed normally in this new, currently sane, communicator. Note that obtaining a sane communicator is critical in the case of inflexible applications, as the existence of a fully functional communicator is a prerequisite to permit employing the `MPI_Comm_spawn` operation to spawn supplementary processes and reconstruct the application world.

Table 2 summarizes the minimal set of routines necessary to deliver recovery capabilities. It could be tempting to provide more elaborate constructs to facilitate other types of recovery support, but it is important to retain one of the major characteristics of MPI: being a toolbox with which more complex algorithms can be built. Thus, instead of limiting how the MPI library can react to a fault by providing a strict recovery model, the three conceptual features described above provide a basic toolset of global recovery capabilities that represent a minimalist base for higher-level constructs, or even domain-specific approaches that can now be provided by additional libraries or frameworks.

4.4. Typical Patterns

Here we present a set of examples of use for the functions described above. Note that these examples are presented for illustrative purposes, but they are not the sole use cases for each routine, nor the only means to deploy a particular

Function	Description
`MPIX_Comm_failure_ack(comm)`	Acknowledgment of reported process failure errors. Resumes matching for `MPI_ANY_SOURCE`.
`MPIX_Comm_failure_get_acked(comm, &group)`	Obtains the group of processes acknowledged to have failed.
`MPIX_Comm_agree(comm, &mask)`	Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce).
`MPIX_Comm_revoke(comm)`	Non-collective with effect on the entire comm, communications on comm (future or active, at all ranks) are interrupted with `MPIX_ERR_REVOKED`.
`MPIX_Comm_shrink(comm, &newcomm)`	Collective, creates a new communicator without failed processes (identical at all ranks).

Table 2: ULFM minimal set of fault tolerance routines.

strategy. More code examples can be found in the Supercomputing tutorial material [9].

An example of application that can run through failures is illustrated in Figure 5. This figure shows the master code of a master-worker application that handles failures by ignoring failed processes and rebalancing the workload dynamically. It illustrates how the routines `MPIX_Comm_failure_ack` and `MPIX_Comm_failure_get_acked` are used in tandem to acknowledge the failure and to obtain the group of failed processes, which permits counting how many workers are still active and requeue the failed tasks in the pool of pending work. It also demonstrates the different failure cases that may occur when posting receptions from `MPI_ANY_SOURCE`. Both `MPIX_ERR_PROC_FAILED` and `MPIX_ERR_PROC_FAILED_PENDING` can be reported to the applications, and they represent slightly different scenarios. For `MPIX_ERR_PROC_FAILED_PENDING`, the request is still pending and once the failure is acknowledged, the request can be reused; while for `MPIX_ERR_PROC_FAILED` it has matched a message from a process that failed and completed and thus needs to be reposted.

A template for applications that can continue their execution on a reduced set of processes is shown in Figure 6. In this code, at the end of each iteration, a collective operation is performed. The return code is obtained from the operation, and the `MPIX_Comm_agree` routine is used to verify whether any other process has returned an error. Then, the `MPIX_Comm_shrink` routine is used to create a new communicator excluding the failed processes. An example of this kind of applications is found in [3], where an iterative application is rendered moldable by redistributing the rest of the dataset among the surviving processes.

Though the previous examples of embarrassingly and loosely coupled applications may need a small effort to incorporate resilience support to their codes, there are other—usually tightly coupled—parallel applications that need a more sophisticated failure handling and restorative actions to be able to resume the execution. A frequent solution, in this case, is the use of error handler routines to encapsulate the notification and restoration operations needed. Figure 7

```c
int master(void)
{
    MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
    MPI_Comm_size(comm, &size);

    /*... create a pool of tasks and dispatch initial tasks to workers ... */

    /*... master waits for replies from workers ...*/
    /*... submit initial request ...*/
    MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );

    /*... execution progress listening from workers ...*/
    while( (active_workers > 0) &&  pending_tasks  ) {
        rc = MPI_Wait( &req, &status );

        /* Failures handle */
        MPI_Error_class(rc, &ec);
        if( (MPIX_ERR_PROC_FAILED == ec) ||
            (MPIX_ERR_PROC_FAILED_PENDING == ec) ) {
            /* Obtain the group of failed processes */
            MPIX_Comm_failure_ack(comm);
            MPIX_Comm_failure_get_acked(comm, &g);
            MPI_Group_size(g, &gsize);

            /* Count the active workers */
            active_workers = size - gsize - 1;
            MPI_Group_free(&g);

            /* ... identify the failed task and
                   requeue it into the pool of tasks ... */

            /* Request is still pending, no need to repost it */
            if( rc == MPIX_ERR_PROC_FAILED_PENDING ) {
                continue;
            }
        } else {
            /*... process the answer from a worker and update pending tasks ... */
        }

        /*... repost a reception to keep on listening for completed
             work ...*/
        MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );
    }

    /*... cancel request and cleanup ...*/

}
```

Figure 5: Example of a master-worker application that can run through failures.

```
1  MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
2  while( !stop_condition ) {
3      /* ... new iteration ... */
4
5      /* At the end of the iteration a collective
6       operation is needed between processes */
7      rc = MPI_<COLLECTIVE_OPERATION>(..., comm);
8
9      /* Detect errors across all processes */
10     allsucceeded = (rc == MPI_SUCCESS);
11     rc = MPIX_Comm_agree(comm, &allsucceeded);
12     MPI_Error_class(rc, &ec);
13     /* In case of failed processes */
14     if( ec == MPIX_ERR_PROC_FAILED || !allsucceeded ) {
15         /* Shrink the communicator */
16         MPIX_Comm_shrink(comm, &scomm);
17         MPI_Comm_free(comm);
18         comm = scomm;
19     } else {
20         stop_condition = update_stop_condition(...);
21     }
22 }
```

Figure 6: Example of a simple iterative application with failures handled at the end of each iteration, that can resume the execution on a reduced set of processes.

shows an example of a general parallel application that uses an error handler routine to revoke the communicator after a failure is detected and invoke an application-specific repair procedure. This example illustrates the usage of the `MPIX_Comm_revoke` to ensure that all processes will be notified of a process failure. In the application-specific repair procedure, it also exemplifies the use of the `MPIX_Comm_shrink` routine combined with the `MPI_Comm_spawn` and `MPI_Comm_Intercomm_merge` to create a new version of the "world" communicator with the same number of ranks.

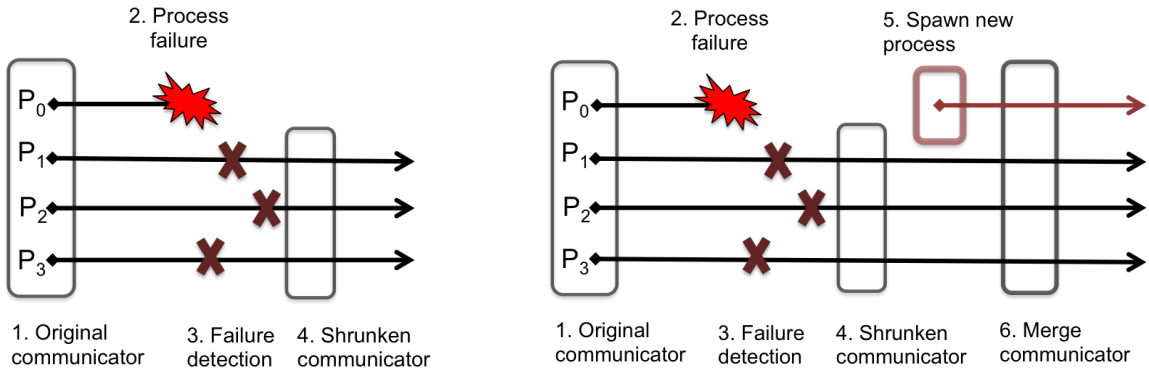
5. Application-Level Resilient Solutions Using ULFM

As presented above, ULFM includes new semantics for failure notification, failure propagation and interruption, and communication context restoration. However, no recovery mechanism is mandated. Instead, ULFM provides a set of basic interfaces to allow the users to adapt the recovery method to the characteristics of their applications. This flexibility has led researchers and production teams to propose different recovery strategies at the application level. Resilience proposals can be classified into:

- *Shrinking or Non-Shrinking*: In shrinking approaches, the failed processes are not replaced; thus, the number of processes running the application decreases after each failure. Shrinking solutions are restricted to applications that tolerate modifying the number of processes at runtime, the so-called malleable applications. On the other hand, non-shrinking recoveries preserve the number of running processes after a failure by replacing the failed processes with spare ones. Figure 8 illustrates the differences between both strategies.

```c
int main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_create_errhandler(&errhandler_respawn, &errh);

    MPI_Comm_get_parent( &parent );
    if( MPI_COMM_NULL == parent ) {
        /* Original processes: create an initial world */
        MPI_Comm_dup( MPI_COMM_WORLD, &world );
    } else {
        /* Spawned processes: repair world */
        repair(&world);
    }
    MPI_Comm_set_errhandler( world, errh );

    /* ... MPI calls appear at different locations in the code ... */
    /* ... no need to check for error return codes, process failures ...*/
    /* ... will invoke the error_handler ... */
    MPI_...(...,world,...);

    MPI_Finalize();
}

static void errhandler_respawn(MPI_Comm* pcomm, int* errcode, ...) {
    int eclass;
    MPI_Error_class(*errcode, &ec);
    if( MPIX_ERR_PROC_FAILED == ec || MPIX_ERR_REVOKED == ec ||
                MPIX_ERR_PROC_FAILED_PENDING == ec) {
        /* Revoke communicator */
        MPIX_Comm_revoke(*pcomm);
        /* Repair the world */
        repair(&world);
    }
}

static int repair(MPI_Comm *comm) {
    if( comm != MPI_COMM_NULL ) {  /* Survivors */
        /* Shrink communicator */
        MPIX_Comm_shrink(comm, &scomm);
        MPI_Comm_size(scomm, &ns);
        MPI_Comm_size(comm, &nc);
        nd = nc-ns; /* number of deads */
        /* Spawn new processes to replace dead ones */
        MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
                               0, scomm, &icomm, MPI_ERRCODES_IGNORE);
        /* Merge the intercomm, to reconstruct an intracomm */
        MPI_Intercomm_merge(icomm, 1, &mcomm);
        /* For simplicity, the management of multiple errors detected */
         /* during the recovery phase is not presented here */
        MPI_Comm_free(comm);
        comm = mcomm;
    }
    /* Both survivors and new spawnees may need to perform other actions,
    such as reassign ranks, data restoration, etc ... */
}
```

Figure 7: Example of a general application using an error handler to manage failures and repair the world communicator.



(a) Shrinking solution.

(b) Non-shrinking solution.

Figure 8: Shrinking vs non-shrinking using ULFM. After a failure, all processes are made aware of the fault and the communicator is shrunk. In the shrinking approach the execution continues from this point, while in the non-shrinking approach new processes are spawned to replace the failed ones and the communicator is reconstructed by merging the shrunken and the new one.

- *Backward or Forward*: In backward solutions, after a failure, the application is restarted from a previously saved state. Forward recovery solutions, in contrast, attempt to find a new state to successfully continue the execution of the application.

- *Global or Local*: In the global approaches, the application repairs a global state to survive the failure. In MPI SPMD applications that means restoring the state of all application processes to a saved state, in order to obtain the necessary global consistency to resume the execution. On the contrary, local recovery solutions attempt to repair failures by restoring a small part of the application—for example, a single process. Due to inter-process communication dependencies, these solutions require the use of message logging techniques for its general application.

The rest of this section surveys and classifies the different ULFM-based fault tolerance methodologies proposed in the literature, remarking on their applicability and scalability.

5.1. Shrinking Solutions

Some proposals exploit the particular characteristics of the applications to avoid the re-spawning of replacement processes. To support such an approach it is clear that the underlying algorithms must have some desirable properties compatible with such a dynamic world. There are many examples of such applications: for instance, iterative asynchronous algorithms where the input of dead processes is not critical for the algorithm to converge and it can be dismissed in exchange for a relative increase in the number of iterations or potentially a loss of accuracy in the results.

This is the case for the proposal of Pauli et al. [41, 42] for the Monte Carlo (MC) and the multi-level MC methods, where a shrinking, forward, local recovery is implemented. MC methods rely on repeated random sampling to obtain numerical results. The fault-tolerant version uses ULFM to detect failures and continue with the computation using

only the survivor processes. In case of failure, the final result is computed using the samples unaffected by failures, while the samples affected by failures are disregarded. In this way, neither checkpoint/restart nor re-computation of lost samples is needed. The proposal obtains a fault-tolerant version with a very low overhead at the expense of potential qualitative degradation of the results when failures occur.

Another shrinking, forward, local recovery implementation is proposed in Strazdins et al. [47]. They present an algorithm-based resilient solution for partial differential equation (PDE) solvers using ULFM. In this approach, a numerical method called the Sparse Grid Combination Technique (SGCT) is employed to approximate the solution to PDEs. Instead of solving the PDEs in a regular full grid, it solves several grids, called components grids, with fewer grid points. Each component grid is run in a subset of processes, and solutions on these components are combined to obtain an approximate global solution. Upon fault detection, the faulty communicator is shrunk, and the data structures of the processes that compute a component grid affected by the failure are updated to adjust the whole range of grid points of that component to the new number of available processes. The processes that compute components not affected by failures continue running without disruption. The component grids associated with the failed processes are not taken into account when building the combined global solution. The redistribution of data is unnecessary, but depending on which components grids are disregarded due to failures, the loss of accuracy will be more or less important. The proposal is compared with a resilient version where the failed processes are replaced by newly spawned ones and with a classical stop & restart solution. Results prove that the shrinking approach leads to better application performance, especially for a high number of cores. Both resilience approaches (with replacement and shrinkage regimes) outperform the stop & restart version.

Rizzi et al. [43] use a similar approach to implement a resilient task-based domain-decomposition preconditioner for PDEs. The algorithm transforms the original PDE problem into many local sampling problems, followed by a regression stage where the local pieces are linked to construct the final global solution. To guarantee a resilient computation, the number of samples generated within each subdomain is the number needed for the fault-free execution multiplied by an oversampling factor. Thus, upon failure, crashed tasks do not need to be recomputed and the application simply discards them and continues the execution using the processes that are alive.

Forward recovery is not the only approach compatible with shrinking solutions; in some cases, a backward recovery can also be applied. In these cases, load computation has to be redistributed among the remaining processes, which introduces an overhead dependent on the particular redistribution algorithm. This is the case shown in [31], where a backward, global, shrinking recovery model is implemented to convert a molecular dynamics program, *ddcMD*, in a resilient application. ULFM is used to detect failures, revoke, and shrink the communicator. A shrinking recovery can be applied to ddcMD because its load can be easily rebalanced at runtime. The implementation uses an in-memory checkpointing mechanism in which each MPI process stores a checkpoint in local memory and replicates it on an adjacent process (similar to buddy checkpointing). In case of failure, the application is restarted from the last checkpoint, and the process that contains the replicated in-memory checkpoint is responsible for the recovery of the data of the failed process. Then, the load is rebalanced and the application continues with its execution.

The re-spawning can be also avoided in MPI master-worker codes. In these applications, ULFM can be used to detect failed workers so that the associated tasks are put back to the queue to be reassigned to other workers. Thus, a shrinking, backward, local recovery can be implemented: only the failed tasks are recomputed using the available resources and the load is automatically redistributed by the master among the available workers. This is the working principle of Falanx [51], a middleware infrastructure for the development of exascale applications. In Falanx, parallel applications are described as a set of tasks, and resilience is achieved by taking the tasks as partial rollback units. It uses ULFM to detect failures and it is equipped with a resource management system for task scheduling and a mechanism for data protection based on data replication.

ULFM is also used in Lemarinier et al. [33] to build malleable MPI applications—that is, those able to dynamically expand or shrink to adapt themselves to the number of available resources. In this work, the ULFM approach is compared with a traditional stop & restart mechanism where the Scalable Checkpoint Restart (SCR) library [39] is used to relaunch the application with a new number of processes after saving the state. Experimental results prove that the ULFM solution enables faster reconfiguration.

Shrinking solutions avoid the overhead associated with the re-spawning of replacement processes to take over the failed ones. However, the execution time of the application may be negatively affected by the use of a smaller number of computational resources.

Ashraf et al. [3] compare a shrinking and a non-shrinking solution for a fault-tolerant version of the generalized minimal residual (GMRES) algorithm. The algorithm already offers protection against silent data corruptions, and protection against hard errors is implemented combining ULFM and a backward global recovery using diskless checkpointing. In the shrinking solution, upon a failure, the workload is redistributed among the survivor processes. Due to the characteristics of the application, the workload redistribution overheads are negligible and thus, the recovery overheads of the shrinking and non-shrinking approaches are comparable. On the other hand, replacement processes are usually mapped to far nodes, which can lead to higher communication overheads. Nevertheless, the time-to-solution is larger in the shrinking solution due to the smaller number of processes to carry out the computation. Authors conclude that the shrinking approach can be a good alternative when spares are not available, or at a large scale when there are enough workers to share the workload of the failed processes.

5.2. Non-Shrinking Solutions

Although shrinking strategies avoid the extra cost of setting up replacement processes to take over the failed ones, they can negatively impact the performance when multiple processes are lost. Moreover, an unbalanced distribution of computation among the survivor processes can further penalize the performance of the application. In addition to this, the implementation of a dynamic adjustment of the workload across survivor processes may not be feasible in many applications, as it may imply large programming efforts from the user. On the other hand, non-shrinking solutions preserve the number of processes running the application after a failure. Once a failure has been detected and the communication engine is repaired, replacement processes will take over the failed one's workload. Replacement

processes can be spawned on demand for each failure, using the MPI spawning capabilities. Alternatively, a pool of spare processes can be created at the beginning of the execution, avoiding the spawning overhead during the recovery. Such a reconstruction of the parallel execution setup comes with its own overheads: data lost due to the dead processes must be recovered and transferred to the replacement processes before the execution can continue.

Ali et al. [1] focus on non-shrinking recovery of PDE-based applications, re-spawning replacement processes on the same node, when they are still available, or otherwise in pre-allocated spare-nodes. As in their shrinking proposal [47], they use the SGCT Algorithm-Based Fault Tolerance (ABFT) strategy to approximate recovery of multiple failures, rather than the exact recovery through checkpointing. Data from failed processes is recovered using an alternate component grid combination formula by adding some redundancy to recover the data from lost processes in a local forward recovery.

Teranishi et al. [52] propose the Local Failure Local Recovery (LFLR) framework. LFLR uses ULFM capabilities to detect failures and repair the communication engine. Then, failed processes are substituted by warm spare ones from a pool of processes, which run a skeletonized version of the application code (the program logic execution, but skipping the real computation). The recovery is based on checkpointing but, to minimize the performance impact, the framework makes use of the spare processes to implement diskless checkpointing. The framework is tested with the MiniFE application, a parallel finite element analysis code for thermal PDEs, which enables failure detection by using the `MPIX_Comm_agree` routine to stop all the processes in the same iteration.

Cantwell et al. [12] also target non-shrinking resilient PDE solvers. In this case, they focused on reducing the amount of checkpointed data and minimizing the amount of instrumentation in the application code by using message logging. The authors distinguish between static data, which is fixed during the simulation after an initialization phase, and dynamic data, or time-evolving data. The user is responsible for marking the initialization phase of the application by annotating the code. During the initialization phase, the outcome of each MPI communication is logged. Then, once the initialization phase is completed, the dynamic data is periodically checkpointed. Both the log and the checkpointed data are backed up in a buddy process. After a failure, the ULFM functionalities are used to repair the communication engine, and warm spare processes are used to replace the failed ones. During the recovery, survivor processes roll back to the last dynamic checkpoint. On the other hand, replacement processes re-execute the initialization phase of the code; however, MPI communication calls are intercepted and their results are obtained directly from the log. After the initialization phase, the dynamic state of the replacement processes is recovered from the last dynamic checkpoint, and the execution resumes.

Bland et al. [7] study the performance gain of using a non-shrinking resilience strategy instead of traditional checkpoint/restart on an iterative refinement stencil code using the Monte Carlo Communication Kernel, showing that the total time to completion can improve by as much as 75% using ULFM. The use of the ULFM functionalities avoids the interruption of the application, allowing the checkpointed data to be kept in memory of the neighbor processes. Exploiting the particularities of this code, at the end of each iteration the checkpointed data is sent to a neighbor process, and processes check for failures using the `MPIX_Comm_agree` routine.

Gamell et al. [24] propose a local backward recovery proposal for stencil-based applications. In this case, taking into account the particular characteristics of the target applications, only the failed processes have to roll back. The experimental results prove that the local recovery obtains important performance benefits as compared to the global recovery. This work involved non-trivial refactoring of the message passing capability to keep one-to-one communication alive for the survived processes.

The previous proposals consider the particular characteristics of the applications to simplify the recovery process. A customized solution allows reducing the recovery overhead upon failure—for example, simplifying the detection of failures by checking the status of the execution in specific points or recovering the application data by means of its properties as an alternative to checkpointing. However, this also restricts their applicability, making them not suitable to be generally applied to any MPI application

On the other hand, CRAFT (Checkpoint-Restart and Automatic Fault Tolerance) [45] builds upon ULFM to provide a generic library for application-level checkpointing and dynamic process recovery not restricted to a particular type of application. It provides multithreaded multi-level coordinated checkpointing. It supports both non-shrinking recovery, spawning new replacement processes to take over, rolling back all processes to a previous state, and shrinking recovery, which relies on the user for the recovery procedure.

In the same vein, Fenix [23] is a library that facilitates the implementation of resilient MPI applications. It uses ULFM to detect failures and recover communicators, and implements, with the help of the user, an application-aware implicitly coordinated diskless checkpointing. It follows a non-shrinking global backward model: in case of failure, failed processes are re-spawned and all processes go back to the last checkpoint. This idea was extended to the specification of application programming interfaces [38] of the Fenix library, which accommodates abstraction of MPI fault tolerance and shields the users calling ULFM interfaces by hiding all capabilities inside the MPI profiling (PMPI) layer.

A local rollback protocol that can be generally applied to single program, multiple data (SPMD) applications is proposed in [35]. It combines the ComPiler for Portable Checkpointing (CPPC) tool, message logging, and ULFM. The application code is automatically instrumented to add fault tolerance support. After a failure, replacement processes are re-spawned to take over the failed ones. During the recovery, only the replacement processes are rollbacked to the last checkpoint, while consistency and further progress in the execution is achieved through a two-level message logging process. Point-to-point communications are logged by the Open MPI VProtocol [10] component at the library level. On the other hand, collective communications are optimally logged at the application level, thereby decoupling the logging protocol from the particular collective implementation. Besides, the spatially coordinated checkpointing protocol applied by CPPC reduces the log size, the log memory requirements, and overall the resilience impact on the applications.

Failure detection	Stop & restart		Application aborted due to failures
	Resilience		Notification of failures to all live processes[1]
Reconstruction	Stop & restart		Relaunching of the application
	Resilience	Shrinking	Agreement about failed processes
			Shrinking of the communicator
			Load rebalancing
		Non-shrinking	Agreement about failed processes
			Shrinking of the communicator
			Re-spawning of failed processes[2]
			Communicator reconstruction
Restart	Stop & restart	Global backward	Recomputation from the last checkpoint
	Resilience	Global backward	Recomputation from the last checkpoint
		Local backward	Recomputation of failed tasks
		Local forward	Recomputation is not needed

[1] In some cases, it is enough to notify a subset of live processes

[2] Or activation of warm spare processes

Table 3: Recovery operations in traditional stop & restart solutions vs ULFM proposals.

6. Resilient vs. Stop and Restart Solutions

The previous section reviews the application-driven solutions exploiting the ULFM functionalities that have been proposed in the last years. In order to give a global overview of the performance benefits that can be obtained in fault tolerance techniques exploiting the ULFM constructs, this section compares traditional stop & restart and resilient strategies to cope with faults in the applications. Table 3 provides a schematic overview of the alternative recovery operations to tolerate failures in resilient solutions using ULFM, comparing them with the actions performed in stop & restart approaches. The next paragraphs provide some insights into the performance of both strategies, focusing on the recovery operations involving ULFM. Although the main objective of this paper is providing a survey of the ULFM capabilities and it is not an experimental work, we have included in this section some results from our previous works [36, 35] for illustrative purposes. Note that these results do not intend to experimentally evaluate the overheads of ULFM, but highlight the potential of the ULFM approach.

In the failure detection stage, stop & restart techniques cause the entire application to abort when one or several processes fail, while the ULFM resilience constructs enable failure notification to some or all the remaining live processes without global cancellation of the application. Besides, the existence of a well-defined propagation mechanism (i.e., communication revocation), exposed through the ULFM API, allows for highly optimized implementations, as proposed in [8]. Such implementations take advantage of underlying MPI capabilities and the structure of applications

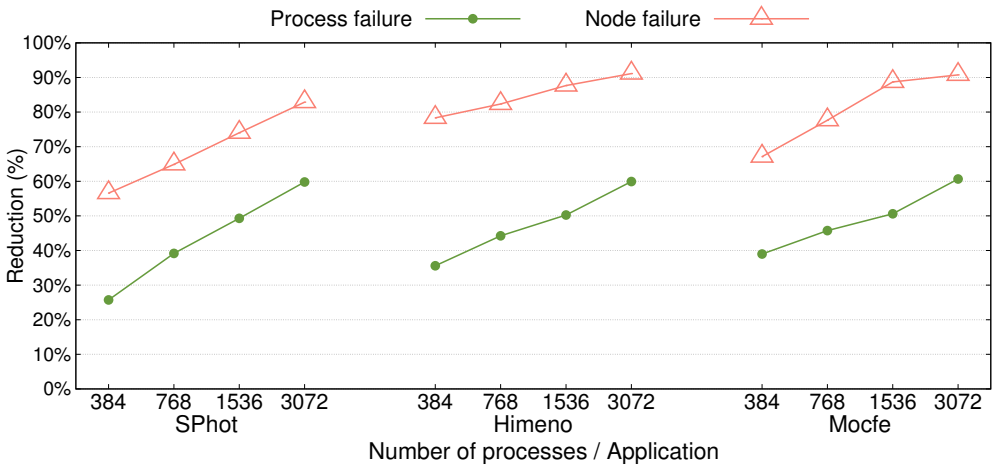


Figure 9: Percentage of reduction in the global failure detection time when using ULFM with respect to the stop & restart solution for one-process failure and for a full-node failure.

to improve the speed at which process faults are detected and to deliver a fast and reliable multicast using the same high-speed interconnect as the MPI library itself. Thus, the failure propagation constructs in ULFM enable faster global failure detection. To illustrate this, we compared a stop & restart solution to an equivalent resilient approach when scaling out three applications with different checkpoint file sizes and communication patterns. The Advanced Simulation and Computing (ASC) Sequoia Benchmark SPhot [2] is a physics package that implements a Monte Carlo Scalar PHOTon transport code, the Himeno [29] benchmark is a Poisson equation solver using the Jacobi method, and MOCFE-Bone [53] simulates the main procedures in a 3D method of characteristics (MOC) code for the numerical solution of the steady-state neutron transport equation. Figure 9 represents the percentage reduction in the global detection times achieved using ULFM with respect to a stop & restart solution using MPI default detection mechanisms. On average, ULFM reduces by 47% and 79% the global detection times for one-process failure and for a full-node failure respectively, and the benefit increases as the applications scale out. More details about the applications and the experimental environment can be found in [36].

Once a failure is detected, the communication environment needs to be reconstructed. In the case of stop & restart techniques, the entire application needs to be relaunched, all processes need to be restarted, and the MPI communication capabilities set up (time spent in the `MPI_Init` routine). On the other hand, resilience strategies allow further communication between the live processes by identifying the failed ones and excluding them from the MPI communicators (using the ULFM routine `MPIX_Comm_shrink`). Then, the application continues with a lesser number of processes (shrinking approach), or the failed processes are replaced (non-shrinking approach). In the first case, the computational load must be rebalanced among the survivor processes. In the second case, the communicator must be reconstructed to include the replacement processes (from spares, or dynamically added to the allocation).

Figure 10 illustrates the reduction in reconstruction time that can be achieved when using a non-shrinking resilience strategy instead of a stop & restart one. The benefits obtained with the resilience approach are reduced when

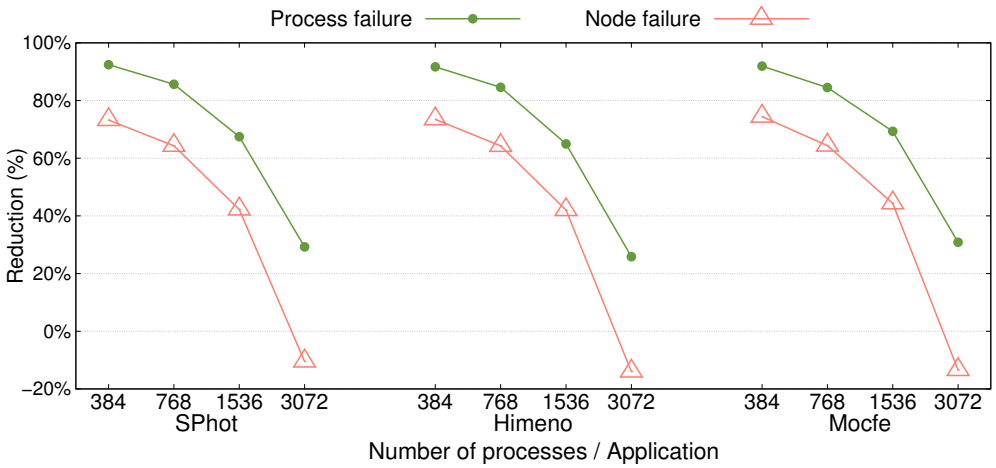


Figure 10: Percentage of reduction in the reconstruction operations time when using ULFM with respect to the stop & restart solution.

increasing the number of running processes and the number of failed ones. Most of the reconstruction overhead is due to the cost of spawning new processes to replace the failed ones and the reconfiguration of the communicator so that each spare takes over a failed process. As shown in Figure 11, the shrinking operation is negligible, while the cost of the spawn and reconfiguration increases when scaling out and when increasing the number of failed processes. Such suboptimal behavior is, at least in part, due to the lack of optimization in the underlying spawn mechanisms used by MPI implementations. It must be noted that the dynamic processing capabilities of MPI are rarely necessary outside resilience (at least in the context of HPC), and even in such rare cases, the extremely small number of potential users lead MPI implementors to provide workable but suboptimal solutions. As with most MPI features, a growing number of potential users has a positive impact on the willingness of MPI implementors to put effort into design and to improve the needed features. However, this is not the only solution to reduce the overhead of restoring all processes. The overhead due to the re-spawning can also be reduced by out-of-band management of spare processes, either by allocating spare processes during initialization or by asynchronously paying the cost of maintaining a set of spares during execution. There are two options in the management of spare processes: the use of *warm* or *hot* spare processes. Warm spares are kept in standby until the occurrence of a failure, whereas hot spares imply the replication of the application processes, that is, they perform active execution. The use of hot spares requires substantially more hardware resources which, depending on the replication factor, might rapidly become prohibitively expensive.

Lastly, the application's state needs to be recovered in order to reach a consistent point from which the execution can be resumed. Traditional stop & restart relies on global rollback checkpointing: the application state is recovered from the last consistent checkpoint, repeating the computation from that point to the failure, and then resuming the execution. A global rollback can also be applied to a resilience approach after handling the failure using the ULFM constructs. However, in many instances, the failure has a localized scope and its impact is restricted to a subset of the resources being used. The usage of the ULFM constructs avoids the complete loss of the state in the processes

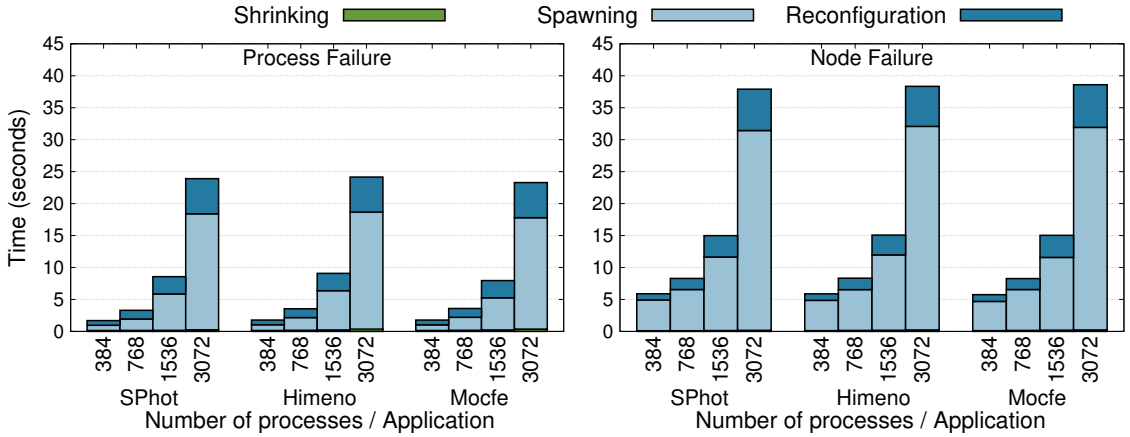


Figure 11: Time of the reconstructing step in a non-shrinking resilience solution.

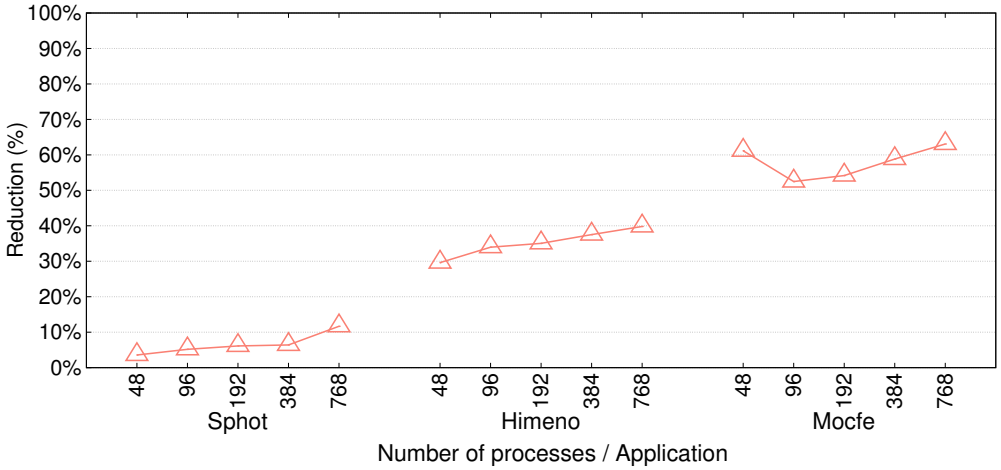


Figure 12: Percentage of reduction in the restart times when using local rollback with respect to the global rollback recovery.

not affected by the failure. Therefore, enabling the implementation of more efficient strategies to recover a consistent application state from which the execution can resume. The usage of local rollback recoveries, in which only a subset of the processes rollback to a previously saved state, introduces both performance and energy saving benefits. For illustrative purposes, Figure 12 shows the performance improvement obtained by using the local rollback protocol build over ULFM in [35]. It reports the percentage reduction in the restart times when one process in the application is recovered using the local rollback protocol instead of a global rollback. In the local rollback protocol, only the processes affected by the failure rollback and repeat computation, while consistency and progress in the execution are achieved through message logging. At the cost of logging, the recomputation performed by the recovering processes results in a more efficient execution of the computation: no communications waits are introduced, received messages are rapidly available, and unnecessary message emissions from past states of the computation are skipped. Therefore, contributing to reduce the overall failure overhead. An extended evaluation of the local rollback protocol and more

details about the message logging cost are available at [35, 34]. Additionally, in certain applications [41, 42, 47, 43, 1], forward recoveries can be implemented, avoiding the recomputation state altogether, by building a new application state from which the execution can resume.

7. Concluding Remarks

Due to the increase in the number of computational resources in IT infrastructures, failures become the norm, and, therefore, the need for resiliency at the programming level surges for long-lasting and large scale applications. Thus, the lack of resilience support in MPI becomes a major handicap for the adoption of MPI as a communication infrastructure outside the HPC niche.

Though the research studies and attempts to incorporate fault tolerance into the MPI standard go back almost two decades, the most auspicious active project towards this end is nowadays ULFM. It supports a variety of fault tolerance models, and with its low-level API provides a complete set of basic constructs for building resilient algorithms.

The first fully fledged implementation of the ULFM extensions was available in Open MPI since 2012, and new releases have been regularly issued since. The convenience of this implementation has led to its broad adoption, at least in the research community, in a large range of domains and scenarios. Moreover, several efforts have also been made to integrate and use ULFM into libraries and frameworks that help users leverage resilience capabilities on their programs.

This paper describes the resilience constructs provided by the ULFM interface and reviews a wide variety of fault tolerance solutions for MPI applications using ULFM, pointing out trends and issues expected in the next computing milestones. As a concluding remark, it could be stated that ULFM provides the necessary support for application-driven recovery—a portable approach that can work both at the scales expected from future exascale platforms, and in more volatile settings such as cloud computing.

Acknowledgments

This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2016-75845-P) and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research (ref. ED431C 2017/04). This research was also supported by the National Science Foundation of the United States under award NSF-SI2 #1664142, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Ali, M. M., Strazdins, P. E., Harding, B., and Hegland, M. Complex Scientific Applications Made Fault-Tolerant with the Sparse Grid Combination Technique. *The International Journal of High Performance Computing Applications 30*, 3 (2016), 335–359.

[2] ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks/`. Last accessed: July 2019.

[3] Ashraf, R. A., Hukerikar, S., and Engelmann, C. Shrink or Substitute: Handling Process Failures in HPC Systems Using In-Situ Recovery. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2018), pp. 178–185.

[4] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[5] Batchu, R., Neelamegam, J. P., , Beddhu, M., Skjellum, A., Dandass, Y., and Apte, M. MPI/FT/sup TM/: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid* (2001), pp. 26–33.

[6] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., and Dongarra, J. Post-Failure Recovery of MPI Communication Capability: Design and Rationale. *The International Journal of High Performance Computing Applications 27*, 3 (2013), 244–254.

[7] Bland, W., Raffenetti, K., and Balaji, P. Simplifying the Recovery Model of User-Level Failure Mitigation. In *Workshop on Exascale MPI at Supercomputing Conference* (2014), IEEE, pp. 20–25.

[8] Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., and Dongarra, J. A Failure Detector for HPC Platforms. *The International Journal of High Performance Computing Applications 32* (01-2018 2018), 139–158.

[9] Bosilca, G., Bouteiller, A., Herault, T., and Robert, Y. Fault-Tolerance for High Performance and Distributed Computing: Theory and Practice. `http://fault-tolerance.org/2018/11/09/sc18/`, November 2018. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18) Tutorial.

[10] Bouteiller, A., Bosilca, G., and Dongarra, J. Redesigning the Message Logging Model for High Performance. *Concurrency and Computation: Practice and Experience 22*, 16 (2010), 2196–2211.

[11] Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., and Cappello, F. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *The International Journal of High Performance Computing Applications 20*, 3 (2006), 319–333.

[12] Cantwell, C. D., and Nielsen, A. S. A Minimally Intrusive Low-Memory Approach to Resilience for Existing Transient Solvers. *Journal of Scientific Computing 78*, 1 (2019), 565–581.

[13] Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., and Snir, M. Toward Exascale Resilience: 2014 Update. *Supercomputing Frontiers and Innovations 1*, 1 (2014), 5–28.

[14] Castain, R. H., Hursey, J., Bouteiller, A., and Solt, D. PMIx: Process Management for Exascale Environments. *Parallel Computing 79* (2018), 9 – 29.

[15] Chakraborty, S., Laguna, I., Emani, M., Mohror, K., Panda, D. K., Schulz, M., and Subramoni, H. EReinit: Scalable and Efficient Fault-Tolerance for Bulk-Synchronous MPI Applications. *Concurrency and Computation: Practice and Experience 0*, 0 (2018), e4863.

[16] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not. 40*, 10 (Oct. 2005), 519–538.

[17] Di Martino, C., Kalbarczyk, Z., and Iyer, R. Measuring the Resiliency of Extreme-Scale Computing Environments. In *Principles of Performance and Reliability Modeling and Evaluation*. Springer, 2016, pp. 609–655.

[18] Dongarra, J., Herault, T., and Robert, Y. Fault Tolerance Techniques for High-Performance Computing. In *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015, pp. 3–85.

[19] Emani, M., Laguna, I., Mohror, K., Sultana, N., and Skjellum, A. Checkpointable MPI: A Transparent Fault-Tolerance Approach for MPI. Tech. Rep. LLNL-CONF-739586, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[20] Fagg, G., and Dongarra, J. FT-MPI: Fault tolerant MPI, Supporting Dynamic Applications in a Dynamic World. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2000), 346–353.

[21] Fanfarillo, A., Garain, S. K., Balsara, D., and Nagle, D. Resilient Computational Applications using CoArray Fortran. *Parallel Computing 81* (2019), 58 – 67.

[22] Fang, A., Laguna, I., Sato, K., Islam, T., and Mohror, K. Fault Tolerance Assistant (FTA): An Exception Handling Programming Model for MPI Applications. Tech. Rep. LLNL-TR-692704, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[23] Gamell, M., Katz, D. S., Kolla, H., Chen, J., Klasky, S., and Parashar, M. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)* (2014), pp. 895–906.

[24] Gamell, M., Teranishi, K., Mayo, J., Kolla, H., Heroux, M., Chen, J., and Parashar, M. Modeling and Simulating Multiple Failure Masking Enabled by Local Recovery for Stencil-based Applications at Extreme Scales. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[25] Hamouda, S. S., Herta, B., Milthorpe, J., Grove, D., and Tardieu, O. Resilient X10 over MPI User Level Failure Mitigation. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10* (2016), X10 2016, pp. 18–23.

[26] Hassani, A., Skjellum, A., Bangalore, P. V., and Brightwell, R. Practical Resilient Cases for FA-MPI, a Transactional Fault-Tolerant MPI. In *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI'15)* (2015).

[27] Hassani, A., Skjellum, A., and Brightwell, R. Design and Evaluation of FA-MPI, a Transactional Resilience Scheme for Non-blocking MPI. In *International Conference on Dependable Systems and Networks* (2014), pp. 750–755.

[28] Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., and Dongarra, J. Practical Scalable Consensus for Pseudo-synchronous Distributed Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 31:1–31:12.

[29] Himeno Benchmark. `http://accc.riken.jp/en/supercom/himenobmt/`. Last accessed: July 2019.

[30] Laguna, I., Gamblin, T., Mohror, K., Schulz, M., Pritchard, H., and Davis, N. A Global Exception Fault Tolerance Model for MPI. In *Workshop on Exascale MPI at Supercomputing Conference (ExaMPI)* (2014).

[31] Laguna, I., Richards, D. F., Gamblin, T., Schulz, M., and de Supinski, B. R. Evaluating User-Level Fault Tolerance for MPI Applications. In *European MPI Users' Group Meeting* (2014), ACM, p. 57.

[32] Laguna, I., Richards, D. F., Gamblin, T., Schulz, M., de Supinski, B. R., Mohror, K., and Pritchard, H. Evaluating and Extending User-Level Fault Tolerance in MPI Applications. *The International Journal of High Performance Computing Applications 30*, 3 (2016), 305–319.

[33] Lemarinier, P., Hasanov, K., Venugopal, S., and Katrinis, K. Architecting Malleable MPI Applications for Priority-Driven Adaptive Scheduling. In *Proceedings of the 23rd European MPI Users' Group Meeting* (2016), ACM, pp. 74–81.

[34] Losada, N., Bosilca, G., and Bouteiller, A. Asynchronous Receiver-Driven Replay for Local Rollback of MPI Applications. In *2019 IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)* (2019).

[35] Losada, N., Bosilca, G., Bouteiller, A., González, P., and Martín, M. J. Local Rollback for Resilient MPI Applications with Application-Level Checkpointing and Message Logging. *Future Generation Computer Systems 91* (2019), 450–464.

[36] Losada, N., Martín, M. J., and González, P. Assessing Resilient versus Stop-and-Restart Fault-Tolerant Solutions in MPI Applications. *The Journal of Supercomputing 73*, 1 (2017), 316–329.

[37] Louca, S., Neophytou, N., Lachanas, A., and Evripidou, P. MPI-FT: Portable Fault Tolerance Scheme for MPI. *Parallel Processing Letters 10*, 4 (2000), 371–382.

[38] Marc Gamell, Rob F. Van der Wijngaart, K. T., and Parashar, M. Specification of Fenix MPI Fault Tolerance Library. Tech. Rep. SAND2016-10522, Sandia National Laboratories, USA, 2016.

[39] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System. In *Conference on High Performance Computing Networking, Storage and Analysis, SC* (2010), IEEE, pp. 1–11.

[40] Numrich, R. W., and Reid, J. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum 17*, 2 (Aug. 1998), 1–31.

[41] Pauli, S., Arbenz, P., and Schwab, C. Intrinsic Fault Tolerance of Multilevel Monte Carlo Methods. *Journal of Parallel and Distributed Computing 84* (2015), 24–36.

[42] Pauli, S., Kohler, M., and Arbenz, P. A Fault Tolerant Implementation of Multi-Level Monte Carlo Methods. *Parallel Computing: Acceler-*

ating Computational Science and Engineering 25 (2014), 471.

[43] Rizzi, F., Morris, K., Sargsyan, K., Mycek, P., Safta, C., Debusschere, B., LeMaitre, O., and Knio, O. ULFM-MPI Implementation of a Resilient Task-Based Partial Differential Equations Preconditioner. In *Workshop on Fault-Tolerance for HPC at Extreme Scale* (2016), ACM, pp. 19–26.

[44] Sato, K., Moody, A., Mohror, K., Gamblin, T., de Supinski, B. R., Maruyama, N., and Matsuoka, S. FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *International Parallel and Distributed Processing Symposium* (2014), IEEE, pp. 1225–1234.

[45] Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., and Wellein, G. CRAFT: A Library for Easier Application-Level Checkpoint/Restart and Automatic Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems 30*, 3 (March 2019), 501–514.

[46] Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al. Addressing Failures in Exascale Computing. *The International Journal of High Performance Computing Applications 28*, 2 (2014), 129–173.

[47] Strazdins, P. E., Ali, M. M., and Debusschere, B. Application Fault Tolerance for Shrinking Resources via the Sparse Grid Combination Technique. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International* (2016), IEEE, pp. 1232–1238.

[48] Subramaniyan, R., Aggarwal, V., Jacobs, A., and George, A. FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems. In *14th Annual European Symposium on Algorithms* (2006), pp. 3–9.

[49] Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. NR-MPI: a Non-stop and Fault Resilient MPI. In *International Conference on Parallel and Distributed Systems* (2013), IEEE, pp. 190–199.

[50] Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. NR-MPI: A Non-Stop and Fault Resilient MPI Supporting Programmer Defined Data Backup and Restore for E-Scale Super Computing Systems. *Supercomputing Frontiers and Innovations 3*, 1 (2016), 4–12.

[51] Takefusa, A., Ikegami, T., Nakada, H., Takano, R., Tozawa, T., and Tanaka, Y. Scalable and Highly Available Fault Resilient Programming Middleware for Exascale Computing. In *Proceedings of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)(Technical Poster)* (2014).

[52] Teranishi, K., and Heroux, M. A. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *European MPI Users' Group Meeting* (2014), ACM, p. 51.

[53] Wolters, E., and Smith, M. MOCFE-Bone: the 3D MOC Mini-Application for Exascale Research. Tech. Rep. ANL/NE-12/59, Argonne National Lab.(ANL), Argonne, IL (United States), 2013.

[54] Yoo, A. B., Jette, M. A., and Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing* (2003), pp. 44–60.