SAND2015-10877C

# Practical Resilient Cases for FA-MPI, A Transactional Fault-Tolerant MPI

Amin Hassani
University of Alabama at Birmingham
CH 115, 1300 University Blvd
Birmingham, AL, 35205
ahassani@cis.uab.edu

Anthony Skjellum
Auburn University
3101 Shelby Center
Auburn, AL 36849
skjellum@auburn.edu

Purushotham V. Bangalore
University of Alabama at Birmingham
CH 115, 1300 University Blvd
Birmingham, AL, 35205
puri@cis.uab.edu

Ron Brightwell
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-1319
rbbrigh@sandia.gov

## ABSTRACT

MPI is insufficient when confronting failures. FA-MPI (Fault-Aware MPI) provides extensions to the MPI standard designed to enable data-parallel applications to achieve resilience without sacrificing scalability. FA-MPI introduces transactions as a novel extension to the MPI message-passing model. Transactions support failure detection, isolation, mitigation, and recovery via application-driven policies. To achieve maximum achievable performance of modern machines, overlapping communication and I/O with computation through non-blocking operations is of growing importance. Therefore, we emphasize fault-tolerant, non-blocking communication operations plus a set of nestable lightweight transactional *TryBlock* API extensions able to exploit system and application hierarchy. This strategy enables applications to *run to completion* with higher probability than nominally. We modified two proxy applications—MiniFE and LULESH—by adding FA-MPI semantics to them. Finally we present performance and overhead results for 1K MPI processes.

## 1. INTRODUCTION

Resilience is one of the most significant challenges facing large-scale scientific computing. Machines continue to grow in scale in order to achieve ever higher levels of compute performance but without significant improvements in hardware reliability. Such platforms will be prone to increasing rates of failure. It is unclear whether existing strategies for tolerating such faults and recovering from failures will continue to be effective. A study [18] on systems at the Los Alamos National Laboratory concluded that failure rates in supercomputers were linearly correlated with the number of sockets in the subject systems. However, in addition to scale, environmental issues such as cosmic rays played a role in failure rate. In contrast to the concept of standard message passing and other portable program constructs used today, applications need to adapt to different types and rates of faults on various systems, and even adjust to to different scales on the same system at the same location.

MPI [15] was designed to achieve high performance portability without regard to fault-tolerance in the application or application programmer interface (API) level. The MPI model posits a reliable communication layer in which processes are always able to communicate once they become known (virtual all-to-all connectivity). Any failure management is considered either or both the responsibility of the application and/or the underlying MPI library implementation itself. Except for registration of an error handler per-communicator (vs. per-function or other granularity), resource exhaustion, and invalid function argument detection, the MPI programming interface offers no fault-management capability or semantics. As a result, process failure typically causes either a hang or an abort of the parallel application; practical workarounds formally violate the MPI standard.

Here, we present a new standalone approach for fault-tolerance in MPI called Fault-Aware MPI (or FA-MPI) [19, 10]. The key goal of FA-MPI is to extend MPI minimally in order to support a lightweight transactional model for fault-awareness[1]. FA-MPI is based on per-transaction fault-awareness as opposed to per-operation schemes such as User-Level Failure Mitigation (ULFM) [5]. In distributed environments, ad-hoc and non-transactional approaches for reliability are known to encounter a complexity barrier in implementation [4]. However, we hypothesize that transactions provide a straightforward solution for reliability and consistency in such systems. This approach is similar to the Bulk


*ExaMPI2015,* November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-3998-8/15/11$15.00.
DOI: http://dx.doi.org/10.1145/2831129.2831130.

[1]Throughout the paper we use fault-tolerance and fault-awareness interchangeably. Notionally, the concept of fault-awareness is the first step toward resilience and, ultimately, fault-tolerance.

Synchronous Parallel (BSP) [22] model for data parallelism, in which synchronicity is reached in "epochs" and barrier synchronization allows an application to enter a known consistent state. Parallel and distributed environments are different in the degree (granularity) of coupling between components, but tight coupling of parallel systems does not prevent the use of transactional concepts. In fact, coordinated checkpoint-restart, the most prevalent fault-tolerant approach, is akin to a giant transaction. Data is saved in reliable storage at successive intervals and reloaded upon failure. However, transactions at a finer granularity at the application level and in parallel message passing systems have not been explored thoroughly.

Fault-awareness makes the application literally "aware" of failures, and, consequently, fault-tolerance can be achieved through the implementation of available approaches for recovery on a given platform, including hierarchical, multi-level, and hybrid checkpoint/restart, message logging, and Algorithm-Based Fault Tolerance (ABFT); some of these may come with portable APIs, others may be system-specific. Flexibility of this approach to fault-awareness allows an application to tolerate different types and rates of faults in the memory and application hierarchy. FA-MPI allows the granularity of fault-awareness to be configured by enlarging/reducing transaction duration and length (size of execution unit within a transaction) and hierarchically nesting transactions[2].

To achieve resiliency, reduction in instantaneous performance is unavoidable because any fault-tolerant method has to have some minimal fault-free overhead. FA-MPI imposes such an overhead on applications; they will run slightly slower in a zero-fault equivalent operational mode than the comparable program running without benefit of the fault tolerance features in place. However, with fault-free overhead imposed, applications will be able to reach completion (Or run indefinitely for periodic programs that do not terminate per se.), whereas fault-free performance ceases at the first fault/failure. This performance trade-off is especially important given future extreme scale machines with higher projected failure rates. The FA-MPI framework contemplates a programming approach in which the application decides the level of effort it expends in detection vs. the potential for missed detections, as well as the approach that trade-off are managed systematically through tuning the fault-free overhead. Controlling overhead will be accomplished by setting the granularity and hierarchy of synchronization. Applications will be able to decide whether to expend additional fault-free overhead in return for faster recovery or vice versa. In FA-MPI, most of the fault-free overhead resides in transactional operations for starting and ending a transaction, specifically transaction *commit*, which is a synchronizing collective call. The transaction commit performs similarly to an `MPI_ALLREDUCE` function computing on one integer per rank in the fault-free case. However, nominal MPI operations need to have minimal or no overhead in a practical implementation. One of the main features of FA-MPI is that failure is not detected/mitigated/isolated/recovered per operation (in contrast to ULFM).

[2]Implicitly, we accept that a fault-aware MPI program may use different, non-portable solutions to do detection and recovery, but those will best be isolated sections of an application. FA-MPI constructs will provide the portable framework and help with separation of concerns.

The main contribution of this paper is the design and evaluation of FA-MPI in the development of fault tolerant applications. We have modified MiniFE and LULESH to use FA-MPI and evaluated the performance of these applications along with the overhead imposed by FA-MPI. We also updated query for failure and failure recovery API and their semantics from the previous work [10].

The remainder of this paper is organized as follows. In the next section, background information on resilience in MPI is discussed. Section 3 describes basic design of FA-MPI, along with our design rationale. In section 4 we present the design of two mini-applications equipped with FA-MPI. Section 5 consists of performance results that illustrate the fault-free and faulty cost of FA-MPI on HPC applications. Our conclusions and a summary of future work are covered in section 6.

## 2. BACKGROUND AND RELATED WORK

Many studies have revealed the impact of fault-tolerance in MPI based on various methodologies such as checkpoint-restart [17], message logging [2] and replication [7, 8]. Certain approaches require intervention of the application while others are application-transparent. However, a successful, scalable fault-tolerant environment requires the engagement of hardware and many layers of the software stack for failure management, and the message passing middleware plays an important role in this process. Adding fault-tolerance support to the MPI Standard has been an important topic in the past few years. The MPI Forum's Fault-Tolerant Working Group (FTWG) has been developing and considering proposals that extend the MPI interface and provide fault-tolerant semantics. Two recent proposals offered are Run-Through Stabilization (RTS) [13], which was rejected by the MPI forum, and the more recently, User-Level Failure Mitigation (ULFM) [5], which is the being considered for standardization in MPI 4.0. Notably, several studies have found insufficiency in ULFM [1, 20] for data-parallel applications.

The goal of ULFM is to add fault-tolerant support only for "process failure" (transient failures are masked as (mapped to) process failures) and allows implementation of a wide range of fault-tolerant techniques on top of ULFM. It adds a "minimal" set of API extensions to MPI and is suitable for libraries providing fault-tolerance rather than user applications. However, our hypothesis is that a "more general" approach (as opposed to "minimal") is in fact more suitable for handling diverse fault models. ULFM strives to provide fault-tolerant functionality for the entire MPI specification; that goal is evidently overwhelming because MPI was not designed with fault-tolerance in mind.

It is important to consider that the HPC community needs to research and evaluate multiple approaches and reach best practices with fault-tolerant MPI candidates before finalization of fault-tolerant approaches in a future MPI standard. In this paper, we present a new approach that is significantly different from either of the abovementioned proposals. We have separately presented the need for changes in ULFM to us to adopt it for use with FA-MPI in [11] (as one particular example). A common, lower-level set of primitives that enable multiple fault-tolerant approaches would be a potential direction for a future MPI standard to embrace.

Containment Domains [6] are a resilience scheme for parallel environments with transactional application-level semantics. A program is divided into hierarchical structures called

Containment Domains (CDs). Each CD has four components: preserve, body, detect, and recover, which conforms to a simple transactional concept for error detection and recovery. CDs achieve fault-tolerance through *eventual consistency* [6], which means that if failure should be detected, the CD will be retried (recovered) locally without global coordination preventing the use of any communication inside a CD and restricting the types of addressable fault models. *Relaxed CDs* allow communication inside CDs, which forces logging of incoming messages. The relaxed approach can be effective in conjunction with FA-MPI for local/global coordination of recovery. However, CD is not a fault-tolerant Design for MPI and it assumes a fault-tolerant MPI.

Multi-level and hierarchical coordinated checkpointing schemes [3, 16] have been studied recently. These methods take advantage of system hierarchy for data recovery. These methods are suitable to be used with hierarchical and coordinated semantics of FA-MPI.

For several decades, transactions have been used to control concurrency and consistency in systems with shared-resources [4]. They can also be used to control reliability. Distributed consistency is achieved in a mutual agreement protocol between all participants to either *commit* (accept) or *rollback* (reject) a series of changes. From the viewpoint of a message passing system, a transactional commit operation ensures consistency of several "communication" and/or "computation" operations. Basic transactions (flat transactions) consist of two calls: `BEGIN_WORK` and `COMMIT_WORK`. Flat transactions are enough to achieve ACID (Atomicity, Consistency, Isolation, and Durability) reliability, but to control performance and scalability at finer levels other models of transactions like nested transactions, multi-level transactions, chained transactions, save points, and sagas [4] have been used. For example, nested transactions can be used to localize the effects of failures and subsequent recovery to smaller regions. Nested transactions can take advantage of system hierarchy in two ways: Memory hierarchy for saving correct data for future recovery and fault model hierarchy from soft errors to multiple cascading hard failures.

## 2.1 Non-Blocking Communication

The path to Exascale and the need for scalable and dependable applications and libraries motivate the use of (restriction to) non-blocking communication operations in message passing systems (at least for the purposes of revealing and focusing an effective fault tolerant message passing model, rather than coping with the added/accidental complexities of blocking legacy semantics under failures). These also have proven to be the best semantics to achieve overlapping computation, communication, and I/O. Non-blocking semantics are also important aspects for fault-tolerance in MPI. Parallel system components are coupled horizontally through the synchronization operations. In addition, this problem intensifies when fault couples different layers in a component. As an example, an MPI communication operation might hang because of a process failure in a remote process. In this situation, forward progress will be impossible because of the coupling between user thread and communication thread. However, in non-blocking semantics, a remote failure, will not halt the user thread (unless `MPI_WAIT` or a related function is called for completion of operation).

Non-blocking MPI operations are amenable to a fault-tolerant solution because they provide a request handle for tracking the status of the pending or progressing operation. In a blocking model, by way of contrast, any resource or reference to an operations is freed after returning from the call. This means that there is no available mechanism to track the status of blocking operations except via an error code on return, synchronous/local with the user thread. However, returning from a blocking operation like `MPI_SEND` only guarantees that the data buffer can be used and there is no guarantee of data transfer to remote node, nor any assured local failure notification. This motivates FA-MPI to bypass blocking APIs completely and to support only non-blocking API in our current design. Non-communicating blocking operations like `MPI_COMM_RANK` are, however, supported. The majority of current communicator creation functions are blocking and pose a problem during initialization and recovery. But, future versions of MPI will likely incorporate non-blocking versions of communicator creation functions.

# 3. FA-MPI DESIGN

In this section, we describe the design of FA-MPI at the API level and we provide rationale for our design decisions.

## 3.1 TryBlocks

As mentioned above, FA-MPI first restricts MPI to the non-blocking model of communication, then extends MPI with a transactional model designed to allow a series of operations to be "tried" and then "committed" when all operations succeed, or else be "rolled backward" or "rolled forward" when some operations fail. The TryBlock is the fundamental building block of the FA-MPI model. TryBlock operations model a transaction block within which several non-blocking communication, computation, and/or I/O operations are executed.

A TryBlock starts with $\texttt{MPI\_TRYBLOCK\_START}_{\texttt{FA-MPI}}$[3] completes with $\texttt{MPI\_TRYBLOCK\_FINISH}_{\texttt{FA-MPI}}$. $\texttt{MPI\_TRYBLOCK\_START}_{\texttt{FA-MPI}}$ is a locally collective operation[4] that binds a communicator to the TryBlock's request handle. Any communicators (including the communicators associated with MPI window and file objects) used inside a TryBlock should be an improper subset of TryBlock's communicator's group. Violation of this requirement may not result in any error, but such a choice will not guarantee a successful fault-aware mechanism since the transaction commit will ignore processes that are not in the TryBlock's communicator's group. It is not erroneous to use TryBlock with a failed communicator (*e.g.,* that has failed MPI ranks) because TryBlocks may be needed for recovery procedures.

FA-MPI provides three different types of transaction levels: Local, group-wise, and an in-between mode. A local transaction is defined by the flag $\texttt{MPI\_TRYBLOCK\_LOCAL}_{\texttt{FA-MPI}}$ given as an argument to $\texttt{MPI\_TRYBLOCK\_START}_{\texttt{FA-MPI}}$. In this mode, failures are not synchronized among the ranks in the TryBlock's communicator at the end of the TryBlock. This option can be used by approaches such as containment domains as mentioned earlier. In this case, TryBlocks can be

[3] An updated list of introduced API is presented in the appendix A

[4] A locally collective operation is an operation that is called on all (healthy) ranks of a communicator, but there might not be any synchronization or communication between ranks.

retried locally and failures are detected and corrected locally without any group-wise synchronization. However, a majority of data parallel applications need synchronization of some sort and any fault-tolerant API needs certain levels of synchronization. A second mode defined by flag MPI_TRYBLOCK_GLOBAL$_{\text{FA-MPI}}$ allows a TryBlock to synchronize the failures detected at the end of the TryBlock. This is the most important case because it allows the application to synchronize and ensure a healthy state of MPI before continuing. A third mode is one "between" the first two modes and is defined by the flag MPI_TRYBLOCK_WITH_COLLECTIVE$_{\text{FA-MPI}}$. This flag notifies a TryBlock that one of the operations given to the TryBlock will be a synchronizing collective operation such as MPI_ALLREDUCE. Collectives such as MPI_ALLREDUCE, will fail (or hang) at all ranks if a process has failed before the operations have started. Hence with a certain probability, if all operations inside a TryBlock succeed, a TryBlock can presume that everything was fine. Otherwise, if one of these collectives should return a failure that denotes a group-wise failure, failure information can be synchronized.

A TryBlock transaction is completed (committed) by MPI_TRYBLOCK_FINISH$_{\text{FA-MPI}}$, which is a non-blocking function. This function's semantics are complex because of its responsibilities. Its synchronization behavior is defined by the flag in MPI_TRYBLOCK_START$_{\text{FA-MPI}}$. The TryBlock finish function is given an array of request handles that were called after the corresponding TryBlock start. It uses the request handles given to it to detect and propagate failure information. The idea that both application and MPI need to communicate regarding faults leads to the design of TryBlocks based on exchanging MPI operations' request handles. The TryBlock finish acts like an MPI_WAITALL at first and waits for its given requests either to complete successfully, fail, or time out. Then, it builds a list of locally detected faults. If the synchronization flag is set, the operation will synchronize this list with other ranks in the TryBlock's group using a consensus algorithm. Once this synchronization is done, every participating rank will have the same failure information and can make a unified failure recovery decision. At the end of the transaction, ranks decide consistently to accept (completely or partially) or reject the transaction by examining group-wise returned failures. TryBlock completion determines faulty and failed objects, requests, processes, and failures associated with each. This group-wise knowledge allows applications to achieve a consistent recovery procedure.

The motivation behind choosing request handles for MPI_TRYBLOCK_FINISH$_{\text{FA-MPI}}$ and not their MPI_STATUS is the need to check the status of operations in progress and not just operations that already have completed. FA-MPI thereby supports per-operation granularity for faults (without requiring per-operation testing as the operations are issued).

The non-blocking nature of MPI_TRYBLOCK_FINISH$_{\text{FA-MPI}}$ allows multiple TryBlocks to be executed simultaneously. An outer TryBlock sifts out failed and successful TryBlocks. Its request handle can be used to be waited or tested on. The nested property of TryBlocks is required to achieve scalability through application of data and task parallelism for smaller communicator groups, various memory hierarchies, and different fault models. The most important part of a transaction is the commit and this has the most complex implementation. Any implementation should strive to contain fault-free overhead. However, conceptually, this overhead cannot be less than a normal MPI_ALLREDUCE computing over one integer. Since synchronizing failure information is important, higher overhead can be tolerated when failure hits the system.

## 3.2 Failure Notification

Failures can be revealed to the user after the TryBlock's finish call. "Query for failure" is a mechanism for the user to retrieve information about local and group-wise failures. In this model, FA-MPI provides three basic APIs to query for error information. However, in the future, more APIs may be added to cope with different schemes. Although this API is not finalized, our current conceptual model offers a good starting point for further investigation and illustrates what we are working to achieve in the complete specification.

In order to retrieve failed requests in a scalable manner, MPI_GET_FAILED_REQUESTS$_{\text{FA-MPI}}$ is introduced. This function returns an array of indices to failed requests given to MPI_TRYBLOCK_FINISH$_{\text{FA-MPI}}$. This is a scalable approach if an application has a vast number of requests and is only looking for failed ones. However, since (per long-standing convention) the MPI implementation should not allocate memory internally and return it to the application, an allocated array of integers along with the size of array (max) will be given to the function. The *count* parameter is an output value indicating the number of failed requests. This function is designed to capture a wide variety of query functions for different types of error codes. Multiple error codes can be queried to help with different recovery procedures. MPI_ERRCODES_ANY$_{\text{FA-MPI}}$ is defined to act as a wildcard for all error codes discovered in the TryBlock.

To retrieve the list of failed communicators, we can use the MPI_GET_FAILED_COMMUNICATORS$_{\text{FA-MPI}}$ function to acquire such a list. We should note that in nominal MPI applications, different layers of an application might not have handles to specific communicators, but they can have the requests associated with them (such as a TryBlock request). This function can return communicators, which have ranks with specific error codes raised on them. The most common use of this function can be used to retrieve communicators that are failed through process failure (error code MPI_ERR_PROC_FAILED$_{\text{ULFM}}$). Libraries can be provided to handle communicator recovery through these functions. FA-MPI provides more informative error codes than just MPI_ERR_PROC_FAILED$_{\text{ULFM}}$. For example, if soft-faults can be distinguished from hard-faults, this knowledge can help with recovery procedures, as discussed below in section 5.

To obtain the list of failed ranks (based on error codes) in a communicator, MPI_GET_FAILED_GROUP$_{\text{FA-MPI}}$ can be used to get these failed ranks as a group. Different error codes can be combined, to retrieve a broader list of ranks with problems. This information might be useful for certain failure predictors that can explore certain behaviors of specific ranks in a communicator. Or, it might be used to tune timeout values for different collective or point-to-point operations.

Unlike the *revoking* approach used in ULFM, our approach is such that failure notification behavior should not be exposed to the user. It should be MPI's responsibility to handle resource management, memory de-allocation and request completion problems that occur in the presence of failures such as a process crash. In FA-MPI, these problems are addressed by semantics that are defined for underlying implementations of TryBlocks and its behavior in the presence of faults.

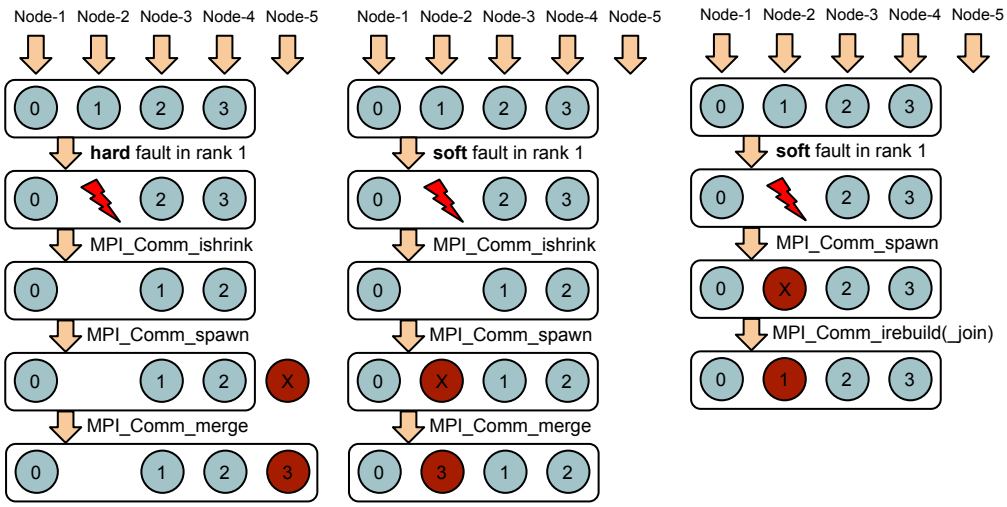



Figure 1: Different types of communicator recovery using `MPI_COMM_SPAWN`. (a) After a hard failure, the communicator has been shrunk, a new process is spawned on a new node and most of the processes alter their rank. (b) After a soft failure, communicator has been shrunk and a process is spawned on the node of failed rank. Most of processes again lose their rank. (c). After a soft fault, a new process is spawned by the root on the node of failed rank and their original, pre-fault ranks are preserved in all processes.

## 3.3 Recovery

Applications can decide to resolve problems in MPI objects (communicators, windows, and files) after notification of local and group-wise failures. For instance, a failed communicator is unable to perform certain collective operations like `MPI_ALLREDUCE`. Certain models of failure such as process crash will cause the communicator to become unusable. To replace this communicator with a healthy one, we suggest a new non-blocking operation $\texttt{MPI\_COMM\_ISHRINK}_{\texttt{FA-MPI}}$ that creates a new communicator with only the healthy ranks (excluding failed ranks). This function is the non-blocking version of $\texttt{MPI\_COMM\_SHRINK}_{\texttt{ULFM}}$ that is introduced in ULFM with one specific difference. In order to use $\texttt{MPI\_COMM\_ISHRINK}_{\texttt{FA-MPI}}$ on a communicator, a synchronizing TryBlock should have been called on this communicator or one of its parent's communicators (containing an improper superset of the group). So before shrinking the communicator, the state of failed ranks have been synchronized and there will be no need for calling an agreement operation inside the $\texttt{MPI\_COMM\_ISHRINK}_{\texttt{FA-MPI}}$. However, users might need to call shrink inside a TryBlock to ensure that no new failure has appeared, otherwise the shrink operation itself will need to be retried. FA-MPI (unlike ULFM) does not require the strict semantics for shrink (that is, FA-MPI does not require a consensus inside the shrink).

FA-MPI maintains the original, single-assignment properties of MPI objects and so repairing a communicator is not compatible with our design. This operation is suitable for applications that can tolerate loss of processes and continuing recovery based on the smaller communicator. Master-Worker style applications are suitable for this purpose. However, the majority of data-parallel applications cannot continue execution with their failed ranks and failed ranks needs to be replaced by healthy ones. In order to grow the shrunken communicator to the original size, new processes can be spawned to form an intercommunicator. Then, this intercommunicator can be merged into an intracommunicator of the original size. Figure 1a and 1b shows the result of this operation when new process is spawned on a different node as failed rank's node or on the same node as failed rank respectively. But, the problem with this method of recovery is that an MPI process' rank in the new communicator will differ in general from its rank in the failed communicator. This can cause significant drawbacks in post-recovery. For example, if checkpoint-restart is done locally on the local disks of each node, any change in rank of a process is undesirable because the rank's data will be on a different node. Also this will cause the communicator to lose its optimized topology and can cause performance degradation in future collective and point-to-point operations.

To address this obstacle, we designed a set of new functions that rebuilds a communicator without processes losing their pre-fault ranks. This function gets a group of newly spawned or already extant processes to fill up the failed ranks in the failed communicator. $\texttt{MPI\_COMM\_IREBUILD}_{\texttt{FA-MPI}}$ is designed for this purpose. The processes in the new group needs to call $\texttt{MPI\_COMM\_IREBUILD\_JOIN}_{\texttt{FA-MPI}}$ to join the new communicator that is being created. This allows the new processes to involve themselves into multiple communicators if needed. An example is when `MPI_COMM_WORLD` is decomposed into a 2D dimensional array of communicators and a newly created rank needs to participate in multiple communicators. The current implementation of rebuild semantics scales linearly. Figure 1c represents the result of such operation for rebuilding a communicator.

FA-MPI considers recovery as a block of computation and communication that can be handled inside a TryBlock even in the presence of faults. It provides an environment for a successful multi-level recovery. Partial soft retry, complete soft retry, roll-back, roll-forward, and checkpoint/restart can

be utilized based on an application's state, recovery decisions and policies.

# 4. APPLICATIONS OF FA-MPI

There are two aspects of recovery for an application that uses a resilient MPI. First, there is the recovery of the MPI layer after a failure. In order to have a resilient application without the need for restarting the MPI layer, we must be able to restore the state of MPI after a failure. Once this step completes successfully, the application's data also needs to be restored. Checkpoint-restart schemes are needed for most HPC applications to recover their data from the last stable point. Recovery models in this category have been well studied in the past. The purpose of this paper is not to emphasize any particular data recovery model or evaluation of such models and the recovery of the MPI layer itself from failed state is a main objective. In this paper, we demonstrate how two mini applications can be augmented with FA-MPI semantics. In next section we present the performance results for both of these mini applications.

We have designed a simple fault-tolerant library for communicator recovery using FA-MPI semantics. Algorithm 1 shows both initialization and recovery semantics for both shrink and rebuild recovery. Most HPC applications cannot tolerate loss of processes and after processes have been lost they have to replaced by healthy processes. In other to achieve this goal, there are two options: Either new processes can be spawned on demand after failure or some processes can be reserved at the initialization point of MPI in a separate communicator for future replacement of failed ranks. For our fault-tolerant library we designed the former. One of the advantages of spawning processes on demand is the ability to spawn a process in the same machine as the failed process. Of course, this only applies if it is a soft failure such as segmentation fault and it can be detected as soft failure. MPI Implementations that utilize dæmons for process management (like Open MPI) have the capability to detect when a process has failed because of a segmentation fault.

If keeping healthy ranks intact in the new communicator is not desired, then after shrinking the communicator, new processes are spawned to form a new intercommunicator. This intercommunicator is merged into a bigger communicator that is is used as the new world communicator. Based on semantics of `MPI_INTERCOMM_MERGE`, processes in the remote group (newly spawned processes) will have either the highest (or lowest) ranks. Figure 1a shows this behavior. For keeping the ranks in new communicator the same as the failed communicator, $\texttt{MPI\_COMM\_IREBUILD}_{\texttt{FA-MPI}}$ is called in all alive ranks in the original communicator and $\texttt{MPI\_COMM\_IREBUILD\_JOIN}_{\texttt{FA-MPI}}$ is called in newly spawned processes. This causes newly spawned processes to replace failed ranks in the original communicator.

In order to spawn processes in the current implementation there is a need for a healthy communicator (a communicator that has been "shrunken"). However, in future improvements to the implementation, there will be no need for a healthy communicator (achieved through shrink) because only the root can spawn processes using `MPI_COMM_SELF` as the communicator and $\texttt{MPI\_COMM\_IREBUILD(\_JOIN)}_{\texttt{FA-MPI}}$ can synchronize the new processes information to other alive processes.

The first mini application we used is MiniFE (also known as HPCCG) from the Mantevo mini-application suite [12]. MiniFE performs a finite element generation through a standard conjugate gradient algorithm that mimics a large production size application of such a capability. It uses 3D box domains and general sparse matrix data formats. It is written in C++ and has the ability to use OpenMP, CUDA, Qthreads, ... programming models. However, in this study we focused on its MPI aspects without regard to any types of thread level parallelization or computational offload engine that might be employed.

**Algorithm 1** Fault-Tolerant Library

```
1: procedure INITIALIZE_MPI
2:     if I_am_spawned then
3:         parent ← MPI_COMM_GET_PARENT()
4:         if rebuild_recovery then
5:             world ← MPI_COMM_IREBUILD_JOIN(parent)
6:         else if shrink_recovery then
7:             world ← MPI_INTERCOMM_MERGE(parent)
8:         end if
9:     else
10:        world ← MPI_COMM_IDUP(mpi_comm_world)
11:    end if
12: end procedure
13: procedure RECOVER_MPI(tbreq)
14:    fcomm ←MPI_GET_FAILED_COMMUNICATORS(tbreq)
   ▷ fcomm should be equal to world
15:    fgroup ←MPI_GET_FAILED_GROUP(tbreq,fcomm)
16:    spawn_size ← MPI_GROUP_SIZE(fgroup)
17:    shcomm ← MPI_COMM_ISHRINK(fcomm)  ▷ shrink is
   not necessary when doing rebuild recovery.
18:    spcomm ← MPI_COMM_SPAWN(shcomm,spawn_size)
19:    if rebuild_recovery then
20:        world ← MPI_COMM_IREBUILD(spcomm)
21:    else if shrink_recovery then
22:        world ← MPI_INTERCOMM_MERGE(spcomm)
23:    end if
24: end procedure
```

Algorithm 2 shows a high level description of a resilient MiniFE based on FA-MPI's semantics of failure detection and recovery. The algorithm starts by duplicating `MPI_COMM_WORLD` because[5] this global communicator cannot be freed and substituted later (once a fault occurs). In the initialization phases of MiniFE, a matrix is created and there are a few group-wise and point-to-point MPI communication operations involved, but for simplicity we did not add fault-awareness for these initialization procedures (mainly because they are short and are not part of the the main body of the algorithm). Upon a process failure in the main loop of the algorithm, using the fault-tolerant library the "world" communicator is replaced with a healthy one using either shrink/merge or rebuild/join mechanism.

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [14] is a simplified proxy application for studying co-design in large scale systems. Its code is in C++ and also has OpenMP capabilities. We have added fault-awareness capability to LULESH's code using

[5]An important part of improving MPI's readiness for fault tolerance in the near future will be to remove such legacy data structures by default in favor of smarter choices for application initialization and communication rendezvous.

**Algorithm 2** Resilient MiniFE

```
1: procedure MAIN(argc,argv)
2:     INITIALIZE_MPI()
3:     my_box, global_box ← CREATE_BOX()
4:     mesh ←CREATE_MESH(my_box,global_box)
5:     A ←CREATE_MATRIX(mesh)
6:     x_0 ←INITIAL_GUESS(mesh)
7:     for i ← 0; norm > tol and i < max_i; i ← i + 1 do
8:         b_i ←CREATE_B(x_{i−1})
9:         tbreq ←MPI_TRYBLOCK_START(world,global_flag)
10:        x_i ←SOLVE(A, b_i)                    ▷ fault injection
11:        MPI_TRYBLOCK_FINISH(tbreq, A.requests)
12:        rc ←MPI_WAITALL_LOCAL(tbreq, timeout)
13:        if rc = timeout then
14:            goto 11
15:        else if rc = found_errors then
16:            rc2 ← RECOVER_MPI()
17:            if rc2 = success then
18:                RECOVER_DATA()
19:            else
20:                MPI_ABORT()
21:            end if
22:        end if
23:    end for
24: end procedure
25: procedure SOLVE(A,b)
26:    for all neighbor n in A.neighbors do
27:        A.requests+ = MPI_ISEND(A.ext_bufs[n],n)
28:        A.requests+ = MPI_IRECV(A.ext_bufr[n],n)
29:        MATRIX_VECTOR_COMPUTATIONS()
30:        MPI_WAITALL_LOCAL(A.requests)
31:    end for
32: end procedure
```

**Algorithm 3** Resilient LULESH

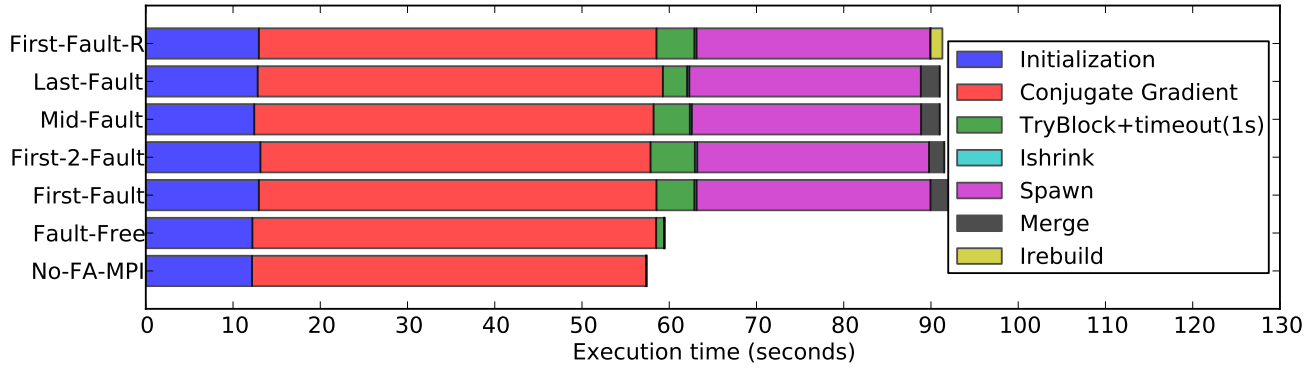
```
1: procedure MAIN(argc,argv)
2:     INITIALIZE_MPI()
3:     tbreq ←MPI_TRYBLOCK_START(world,global_flag)
4:     D ←INITIALIZE_LULESH_DATA()
5:     for i ← 0; norm > tol and i < max_i; i ← i + 1 do
6:         D.requests ←MPI_IALLREDUCE()
7:         MPI_TRYBLOCK_FINISH(tbreq, D.requests)
8:         rc ←MPI_WAITALL_LOCAL(tbreq, timeout)
9:         if rc = timeout then
10:            goto 7
11:        else if rc = found_errors then
12:            rc2 ← RECOVER_MPI()
13:            if rc2 = success then
14:                RECOVER_DATA()
15:            else
16:                MPI_ABORT()
17:            end if
18:        end if
19:        tbreq ←MPI_TRYBLOCK_START(world,global_flag)
20:        SOLVE(D)                              ▷ fault injection
21:    end for
22: end procedure
23: procedure SOLVE(D)
24:    for all i in 1 : 4 do
25:        tbreq ←MPI_TRYBLOCK_START(world,local_flag)
26:        s_reqs ← POST_RECVS(D)
27:        r_reqs ← POST_SENDS(D)
28:        MPI_TRYBLOCK_FINISH(tbreq, s_reqs + r_reqs)
29:        rc ←MPI_WAITALL_LOCAL(tbreq, timeout)
30:        COMPUTE(D)
31:    end for
32: end procedure
```

TryBlocks. LULESH's communication model is bulk synchronous. Each iteration of the main algorithm starts with an `MPI_ALLREDUCE` followed by several communication and computation functions. Inside each iteration four sequences of send and receive are initiated. LULESH's code was not entirely non-blocking, so the first step was to convert all communication functions to their non-blocking counterparts. For each blocks of send and receive we added a local TryBlock to detect failures. Since the TryBlock is local, it has no synchronization, but it is necessary in order to prevent deadlocks. So, totally tow levels of TryBlocks are used in LULESH, one global TryBlock for each iteration to enforce a consensus for each iteration and four local TryBlocks for each block of send and receive. There is no recovery procedure at the second level and a process failure will be handled only at the first level. This choice is for simplicity only. Algorithm 3 shows a simple overview of the FA-MPI added capability to LULESH's code base.

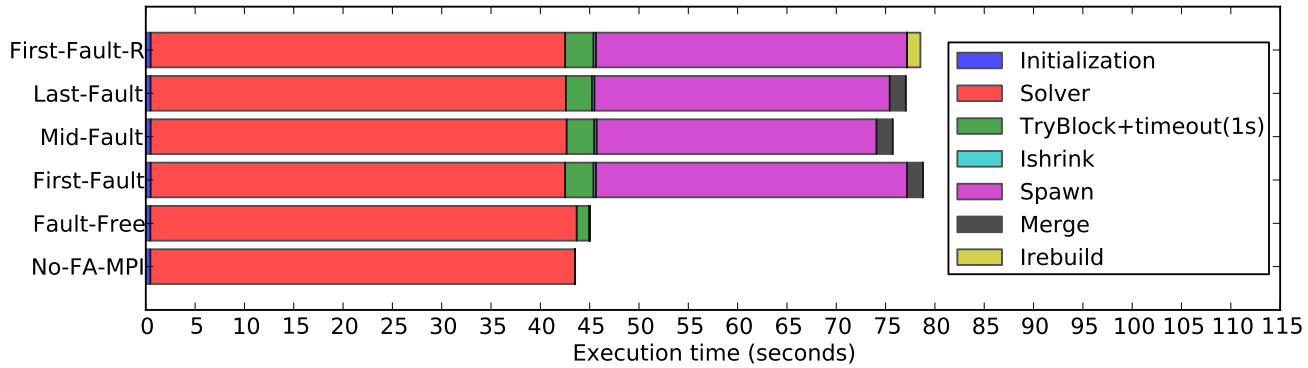
# 5. EXPERIMENTS

We prototyped FA-MPI using a development trunk of Open MPI [9] (version 1.8.4rc5). The current prototype of FA-MPI uses MPI dæmons for failure detection. Once a process failure is detected by a dæmon, it will be propagated to all other dæmons through the head node. In the future, such a choice might be considered only as a backup/secondary approach since FA-MPI will propagate errors using semantics implemented in TryBlocks. These experiments were performed on the Stampede cluster from the Extreme Science and Engineering Discovery Environment (XSEDE) resources [21]. Each node of Stampede incorporates one Intel Xeon E5-2680 at 2.7GHz with 16 cores. There are 2GB memory per core and the interconnect is FDR InfiniBand.

We performed experiments with different fault-models for MiniFE. We did experiments for 1,024 MPI processes. In these experiments, based on the size of the matrix, each rank has about 1GB of local data. The first experiment (*No FA-MPI*) is on MiniFE without any added resilience support. This version uses blocking point-to-point operations for data exchange (MiniFE does not uses any collective operation when computing the conjugate gradient and all communications are done through point-to-point operations). Then for *Fault-Free* experiment, we converted all blocking operations to their non-blocking counterparts. We then added TryBlock semantics to the main loop. We used the synchronizing version of TryBlock with $\texttt{MPI\_TRYBLOCK\_GLOBAL}_{\texttt{FA-MPI}}$ for each iteration of the loop in order to show the maximum overhead that is introduced by FA-MPI. In the last four experiments, along with addition of resilience to MiniFE, we injected one or two process faults at different times during execution. Process faults were injected by artificially creating a segmentation fault in the execution path of application. We injected one process failure for the beginning, end, and middle iterations of the main loop.




(a) Resilient MiniFE on 1,024 MPI processes with global matrix size $nx = 2,000, ny = 1,000, nz = 1,000$ *double* floating point and *64-bit integer* for local and global ordinals and 200 iterations.




(b) Resilient LULESH on 1,000 MPI processes and global domain size $50^3$, 200 iterations.

Figure 2: a) Resilient MiniFE and b) Resilient LULESH. *No FA-MPI*: There are no FA-MPI APIs added to MiniFE. *Fault-Free*: FA-MPI API was added, but no fault was injected. *First-Fault*: One process failure was injected at the first iteration. *First-2-Fault*: Two parallel process failures were injected at the first iteration. *Mid-Fault*: One process failure was injected at the 100th iteration. *Last-Fault*: One process failure was injected at the 198th iteration *First-Fault-R*: One process failure was injected at the first iteration and `MPI_COMM_IREBUILD(_JOIN)`$_{\texttt{FA-MPI}}$ for recovery.

We injected failure for two processes at the same time for experiment *First-2-Fault*. In order to be able to compare the timings we ran the application for exactly 200 iterations even for faulty situations that needed to recover from a past checkpoint. One extra node was reserved when submitting a job to the cluster's queue in order to spawn a new process (efficiently and without overpopulating some nodes). Results in *First-Fault-R* shows performance of rebuild/join communicator instead of shrink/merge. We performed similar experiments for Resilient LULESH.

There are a few points to highlight regarding the results of these experiments. As the *Fault-Free* case shows, the fault-free overhead introduced by FA-MPI semantics is insignificant. Our view is that applications can accept such small overheads in order to avoid failing globally in the presence of faults (such as a process crash) rather than restarting the whole application from a checkpoint. There is a small increase in the total overhead of TryBlocks for faulty cases in comparison to fault-free cases for three reasons. The first reason is the fact that detecting a failure takes a long time. Also, because we propagate the failure internally using Open MPI's dæmons (which uses TCP/IP communication), time is added to the failure detection. In large systems, it might take a few seconds for a failure to be detected and propagated. The second reason, can be a change in topology of ranks in the communicator. Since for some of these experiments we used shrinking and spawning of new processes on a different node, the changes in ranks may result in a changes in topology (where the processes reside) and this can have affect on the performance of collective operations if they are optimized based on the physical positioning of MPI processes. The third reason is the lingering communication requests that may remain in the InfiniBand HCA after a failure. For example, if a point-to-point communication is instantiated in rank $P$ for rank $Q$, and $Q$ fails and its failure is detected by $P$ later, then FA-MPI cancels the operation if it has not yet finished. Canceling an operation is expensive and in some implementations (such as Open-MPI) canceling for a send operation is not implemented. We expect these problems to be alleviated in future versions of Open MPI. Canceling operations are needed for cleaning up the MPI state after communication failure.

We noticed that the execution time of the spawn operation is extremely high (it takes several seconds to complete). This high execution time may be because of an unoptimized implementation or because of the need for some global syn-

chronization among all the processes in the MPI instance through its dæmons using TCP/IP connections. As can be seen, the time required for shrinking or rebuilding a communicator is insignificant.

## 6. CONCLUSIONS AND FUTURE WORK

FA-MPI comprises a set of extension APIs and semantics to the MPI standard that together enable fault-awareness via a transactional model when restricted to non-blocking MPI communication operations. FA-MPI detects, disseminates, and notifies applications of failures and assists with isolation, mitigation, and recovery procedures. Applications using FA-MPI will run to completion with higher probability than with non-fault-aware MPI. To achieve acceptable performance, scalability, and resiliency, we employ configurable, lightweight transactions combined with non-blocking operations. FA-MPI introduces TryBlocks as the main fault-aware components for MPI.

This work is motivated by the fact that there is a decided need for resilience research in MPI that is practical. Revealing different approaches for a fault-tolerant MPI in the HPC community will be valuable and help achieve consensus; the community should not rush to standardize an attractive concept or hypothetical approach without comparing, contrasting, testing, and determining practical benefits. Several viable approaches need to be investigated and best practices need to be created and optimized and only then added to future MPI standards, of which we offer FA-MPI as one potential candidate. The HPC community still does not know to what degree and of what type future failures in massively parallel supercomputers will prove to be and the door must be left open for future unexpected revelations in such large scale systems. Studying viable alternatives to support resilience will help the HPC community best to cope with such future uncertainty.

FA-MPI adds little overhead to MPI applications such as a MiniFE or LULESH. Although our prototype of FA-MPI is in the research phase and is not fully optimized, future implementations will improved the performance of FA-MPI semantics. FA-MPI's approach offers controllable overhead in order to tune the application on a given platform, configuration, scalability, and fault environment.

Finally, there is a need for experimenting with FA-MPI on different applications other than MiniFE and LULESH and broadening use cases of FA-MPI and discovering its shortcoming. FA-MPI semantics allow libraries to be build to help with resilience of applications. Incorporating different checkpoint-restart approaches to an application enhanced with FA-MPI can be a future research direction of this research as well.

## 7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant CCF-1239962 and Sandia National Laboratories under grant PO-1330625. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. This work were also partially supported by Computer Science and Software Engineering department of Auburn University.

# APPENDIX

## A. FA-MPI API

```
int MPI_TryBlock_start(MPI_Comm comm, int flag,
    MPI_Request* tb_request);

int MPI_TryBlock_finish(MPI_Request tb_request,
    int count, MPI_Request array_of_requests[],
    MPI_Timeout timeout);

int MPI_Timeout_set_seconds(MPI_Timeout* timeout,
    double seconds);

int MPI_Timeout_get_seconds(MPI_Timeout timeout,
    double* seconds);

int MPI_Request_raise_error(MPI_Request request,
    int error_code);

int MPI_Wait_local(MPI_Request* request,
    MPI_Status *status, MPI_Timeout timeout);

int MPI_Waitall_local(int count,
    MPI_Request *request, MPI_Status *status,
    MPI_Timeout timeout);

int MPI_Waitany_local(int count,
    MPI_Request *request, int* index,
    MPI_Status *status, MPI_Timeout timeout);

int MPI_Waitsome_local(int incount,
    MPI_Request requests[], int *outcount,
    int indices[], MPI_Status statuses[],
    MPI_Timeout timeout);

int MPI_Get_failed_requests(MPI_Request tb_request,
    int error_codes_count,
    int array_of_error_codes[],
    int max_requests_count, int array_of_indices[],
    int* count);

int MPI_Get_failed_communicators(
    MPI_Request tb_request, int error_codes_count,
    int array_of_error_codes[],
    int max_comunicators_count,
    MPI_Comm array_of_communicators[], int* count);

int MPI_Get_failed_group(MPI_Request tb_request,
    int error_codes_count,
    int array_of_error_codes[], MPI_Comm comm,
    MPI_Group* fgroup);

int MPI_Comm_ishrink(MPI_Comm comm,
    MPI_Comm* newcomm, MPI_Request* request);

int MPI_Comm_irebuild(MPI_Comm comm, int root,
    MPI_Comm aux_intercomm, int N_ranks, int ranks[],
    int is_local, MPI_Comm* newcomm,
    MPI_Request* request);

int MPI_Comm_irebuild_join(MPI_Comm aux_intercomm,
    MPI_Comm* newcomm, MPI_Request* request);
```

## References

[1] M. Ali, J. Southern, P. Strazdins, and B. Harding. Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1169–1178, May 2014.

[2] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu. MPI/FT: a model-based approach to low-overhead fault

tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, Oct. 2004.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[4] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981. Cited by 1091.

[5] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, Aug. 2013.

[6] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

[7] K. Ferreira, J. Stearley, J. H. Laros,III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[8] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78, 2012.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] A. Hassani, A. Skjellum, and R. Brightwell. Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 750–755, June 2014.

[11] A. Hassani, A. Skjellum, R. Brightwell, and P. V. Bangalore. Comparing, Contrasting, Generalizing, and Integrating Two Current Designs for Fault-Tolerant MPI. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 63:63–63:68, New York, NY, USA, 2014. ACM. 00000.

[12] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sept. 2009.

[13] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In *Recent Advances in the Message Passing Interface*, pages 329–332. Springer, 2011.

[14] I. Karlin, J. Keasler, and R. Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, Livermore, CA, Aug. 2013.

[15] Message Passing Interface Forum. MPI: a message-passing interface standard version 3.0. Technical report, Sept. 2012.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[17] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[18] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78:012022, July 2007.

[19] A. Skjellum and P. V. Bangalore. FA-MPI: fault-aware MPI specification and concept of operations. Technical Report UABCIS-TR-2012-011912, University of Alabama at Birmingham, May 2012.

[20] K. Teranishi and M. A. Heroux. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM. 00001.

[21] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkens-Diehr. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering*, 16(5):62–74, Sept. 2014.

[22] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.