FA-MPI: Fault-Aware MPI Specification and Concept of Operations

A Transactional Message Passing Interface & An Alternative Proposal to the MPI-3 Forum

Anthony Skjellum, Purushotham V. Bangalore
Department of Computer and Information Sciences
University of Alabama at Birmingham

Yoginder S. Dandass
Department of Computer Science and Engineering
Mississippi State University
{*tony,puri*}*@cis.uab.edu, yogi@cse.msstate.edu*

February 6, 2012 (Version 0.95)
UABCIS-TR-2012-011912 (DRAFT)

Abstract

This document offers an alternative approach to MPI-3 fault tolerance, and serves as a standalone proposal to the MPI-3 standard. It presents a "transactional model" that addresses detection, isolation, mitigation, and recovery accomplished with application-driven policies enabled by a small number of new MPI concepts and mechanisms. Identification with other fault-tolerant proposals/concepts is not emphasized, for reasons described herein. However, we utilize certain features of MPI-3 not related to fault tolerance in order to enable this model.

Fault-Aware MPI does not seek to be all encompassing, but rather supports non-blocking modes of 1-sided and 2-sided communication, and I/O. This proposal relies on existing MPI mechanisms where available in order to avoid introducing more than a few new concepts. This approach relies on applications building Fault Awareness into their calculations, and using application-specific strategies for both detection and recovery from faults, even if those were occasioned by an MPI-related fault. Since non-blocking communication, and overlapping of communication, computation, and I/O will grow more and more important on the march to exascale, this strategy is consonant with the likely evolution of scalable applications over the next decade, more so than with legacy applications that have emphasized blocking communication. Fault-Aware MPI leverages enhancements in MPI-3 that for the first time make such an "all non-blocking MPI program" achievable within the confines of standard APIs. Nonetheless, by the end of this paper, we show how to recover the blocking API fully in a layered library, without requiring MPI-3 itself to adopt a multiplicatively large number of new API calls.

If successful, Fault-Aware MPI will provide the following critical benefit to scalable programming: Programs written using the proposed approach, then deployed with a Fault-Aware MPI implementation will have a higher probability of running successfully to completion by surviving faults as compared to an equivalent MPI-3 programming running on a non-Fault-Aware MPI implementation on the same system and configuration/scale. The model described here provides guidance as to where Fault-Aware MPI implementations, I/O subsystems, related middleware and above all applications should make investments to enhance this fault-aware value, and how to retain portability, subject to the reality check that faults are different in kind and frequency on different systems, and on equivalent systems at varying scales and/or in distinct locales.

1 Introduction

Fault-Aware MPI-3 is proposed, and a style of MPI programming with these new features is described so that programs can work to achieve resilience to faults, and integrate nominal behavior with recovery behavior.

The following comprise the main contents of this paper:

- Identification of customers/audience for Fault-Aware MPI (aka a Fault-Tolerant MPI),
- Discussion of requirements on MPI applications that want to use a Fault-Aware MPI together with algorithmic-based fault tolerance[1] (ABFT), and/or checking-pointing (CPR) techniques, and/or N-mode Redundancy (NMR), and/or application-based recovery strategies to yield programs that are both Fault-Aware and *resilient* under classes of faults. The goal is for applications to run to completion[2] with significantly higher probability as compared to the equivalent MPI program running on the same MPI implementation without using Fault-Aware modes of operation. Fault-Aware MPI's goal is to enable smooth integration of user-based detection, isolation, mitigation, and recovery, with required MPI detection, isolation, mitigation, and user-driven mechanisms for recovery of MPI capability.
- Motivating Pseudo-code Examples
- Discussion of fault-free overhead associated with this approach, and granularity control so that programs can adjust in systematic ways (achieving a measure of resilience-portability) to various fault environments, where the relative rate of faults is different
- New API definitions: syntax, semantics and advice on how to use
- Discussion on options for implementation of the additional API
- Specific mechanisms based on `TryBlocks`[3] to provide transactional MPI behavior.
- The ability complete operations locally within a `TryBlock`, and to identify failures later at transactional commit (global). This means that `TryBlocks` can start and complete communication and have computation that depends on communication in a `TryBlock`.
- Discussion of implications on the MPI implementation associated with Fault-Awareness support

We note that leverage of transactional concepts appear throughout, and that "making MPI transactional" is a fundamental strategy behind Fault-Aware MPI.

In order to manage the complexity of this paper and to serve its timely role as a proposal to the MPI-3 Forum, we purposefully do not rely on the existing, rapidly evolving proposals, papers, discussions, and drafts for Fault Tolerant MPI from the MPI-3 Fault Tolerant Working Group (a group which has labored about four years on their approach, and whose efforts should not be discounted or diminished in any way by this write-up); furthermore, most recent papers (*e.g.,* [57])

[1] Examples include: RAID stripes, redundant computations and voting

[2] For programs that never terminate, but run in a periodic mode such as real-time programs, we define success for this effort if the programs continue to make progress in terms of delivering their desired functionality.

[3] To create transactions, we use persistent, nonblocking collective communication, a concept first discussed in MPI-2 Standardization and now being reconsidered in MPI-3.

in the Fault-Tolerant-MPI area are from people involved in the MPI-3 Working Group. Neither do we recount or align with the many previous projects in Fault Tolerant MPI including our own work in that area (however, we've cited as much of the entire body of work as we can find in the bibliography to establish what's been done up to now). Nor do we rely on the large literature of non-MPI fault tolerant message passing that itself deserves a large bibliography and appropriate credit. Rather, a fresh, standalone approach/einsatz has been assembled without implied linkages to detailed decisions made in those efforts. This is done so that potential users and MPI Forum members can read this paper, and decide on the efficacy of this work, making reference primarily to the MPI-2 standard plus certain new or proposed MPI-3 features: non-blocking collective communication, the idea of persistent collective communication and `MPI_Comm_create_group`). If this strategy proves of interest to users and/or to the MPI Forum, making associations to such existing work will become both important and necessary, and will be added in future where applicable.

While comparisons to other work are presently limited, we note four particular differences as compared to the working assumptions of the MPI Fault Tolerant Working Group. First, we do not limit ourselves to the fault model of "process failure." Because of this difference, we allow implementations to report other faults, and for applications to report other faults, as well as to cope with situations where a fault has occurred, but it is unmodeled; faults are mappable to operations as well as to communicators/groups. Second, we do not have the stated charter of supporting "all users" to enable them to make "all MPI-3 programs" Fault Tolerant. Because of this distinction, our approach is able to choose a specific approach to MPI programming with emphasis on non-blocking portions of the standard API, plus an augmented API that centers on transactional blocks (called `TryBlocks` throughout the remainder of this paper). Third, we work hard to avoid race conditions involving reliability of the system and subsequent decisions by the application. However, in a fourth area we are fully aligned: Fault-Aware MPI is a set of mechanisms, but, as with the Working Group, we do not provide policies for fault recovery, nor (semi-)automation. In our approach, the application must add a Fault-Awareness aspect in order to achieve a degree of resilience. Furthermore, in terms of nomenclature, prefer "Fault Aware" and "resilient," emphasizing that "Fault Tolerance" implies the entire lifecycle of detection, isolation, mitigation, and recovery, whereas the present approach is really about detection, isolation, and mitigation (both of MPI internally and the application), leaving recovery strategies strictly to the application.

Particularly, we discuss where the four pillars of Fault Tolerance are being handled in MPI and the MPI application (detection, isolation, mitigation, and recovery). MPI's role is restricted to helping detect faults and reporting them to applications (what previously would have been fail stops or "hangs") as well as being implemented in a way that will allow it to continue after certain faults have occurred. MPI also becomes responsible for helping an application to isolate faults within MPI once reported. The application also has the opportunity to report faults it detects to MPI. The application has also to cooperate with MPI to mitigate those faults, regardless of how detected, so that MPI can recover to a consistent internal state either by working with less resources, or by re-adding new resources (such as rebuilding a communicator). If MPI is unable to continue after a given fault and can report that to an application, that is also desired, although not the preferred outcome. An added goal is is to avoid forcing the application and MPI cooperatively to rebuild `MPI_COMM_WORLD`, or communicators peripheral to faults, after every fault.

Interestingly, this proposal reveals a BSP-like model [51] for using MPI (*cf*, [13]), while showing the value of asynchrony in the transactions ("epochs"), and supporting local progress and comple-

tion within transactions. It also reveals the value of strong asynchronous progress in MPI, even in the face of potential rollbacks. Furthermore, potential unscalabilities associated with the fault recovery of build-down hierarchies of communicators starting from `MPI_COMM_WORLD` become evident, suggesting the benefit of "build up" models, and use of inter-communicators during communication organization, and for recovery that avoids accidental complexity.

NB: This is an early draft, and so not all the formal references and review and vetting usual for a formal publication are provided in this version. We intend to publish this as an academic paper. Where needed, we refer anecdotally to related work in this early draft. The full paper will have many more references. Suggestions for references throughout are most welcome.

2 Terminology

The following terminology is used:

Fault A fault comprises a deviation of a system or subsystem from its specifications that causes a contract between the application and MPI or the system to be violated. This can include hardware errors, residual software faults[4], and be temporary[5], session-long, or permanent. Faults cause MPI programs to fail. In the context of this paper, application residual faults (aka "bugs") are not in scope, nor are QoS-type violations. Some faults are invisible. Others cause the system to produce incorrect results, others may simply cause deadlock.

Fault-Awareness Fault-Awareness for MPI means the addition of limited new functionality to support Fault-Aware applications, including the ability to detect and report certain Faults, and to be able to mitigate those faults so that the MPI implementation can continue, although not without explicit reorganization of the resources of the application, nor without interruption of normal application operation for a period of time.

A Fault-Aware MPI application is one that works with a Fault-Aware MPI implementation to detect faults, then reorganizing its use of resources with the MPI API that allows for creation of alternative MPI-based communication and I/O modes of operation with the remaining non-faulty resources (and perhaps new resources). We call such applications resilient to classes of faults of the system when they survive such faults and continue to provide some or all of their original service.

Resilience The ability to survive faults and continue providing a means for mitigating and recovering from a fault or faults, so as to continue to provide some or all service of the original application. The specific faults to which an application is resilient depends on the fault model of the environment, the ability to detect such faults, and the ability to mitigate and then recover from such faults. Not all faults can be modeled or managed by MPI or by an MPI application, so hierarchical recovery is recommended in order to support alternatives to recovery depending on the faults and the ability of MPI, the underlying system, and the application program to continue after the fault occurs. Feedback is used to reduce uncertainty in a complex parallel system, a common theme in systems and control.

[4]However, illegal calls to MPI produce an error message or fatal error, as appropriate, if detectable

[5]Such as single-event upsets in non-ECC memory

Fault Tolerance Fault Tolerance is Fault-Awareness that achieves ideal resilience for all faults that are relevant for the target system (exact fault model). Fault Tolerant programs run to completion as expected. This is a hypothetical, empyrean level of resilience that is impossible to achieve in practice, so we emphasize Fault-Awareness and resilience as practical alternatives in this document.

Fault-Free Overhead The cost, in the absence of a fault, compared to running without Fault-Awareness. This applies to overheads in the MPI implementation, and in the MPI application. By balancing the fault-free overhead, the cost/probability of recovery steps, and the mitigation strategy of the application, the application seeks resilience that minimizes its overall time to completion (or time-averaged throughput for non-terminating applications).

Iron Law for Fault-Awareness Fault-Awareness is only adoptable if it delivers a higher probability of success for a running application, as compared to running without Fault-Awareness. The time to recover from a fault is a fundamental time constant (τ_{Re}), as it the average fault rate R_{F} (inverse time) of the system. If the time to recovery is comparable to the inverse fault rate ($\tau_{\mathrm{Re}} \sim 1/R_{\mathrm{F}}$ then the methodology makes no progress, or may be worse than no fault wareness for short running programs ($T_{\mathrm{Run}} \ll 1/R_{\mathrm{F}}$).

3 Audience

It is essential to identify the potential users in order for this to be an effective alternative strategy and proposal to the MPI-3 Forum, and for there to be a good chance of acceptance in the community once implemented:

- Programmers who currently have or expect to have fault issues with their MPI applications.
- People writing new MPI-3 programs who want to make resilient programs for current and future architectures
- Developers of program transformation (source-to-source translators) who will take existing MPI-1 and MPI-2 code and "aspect it" for fault awareness using the proposed methodologies.[6]
- Users who want to retrofit their existing MPI programs for resilience by hand, who already have programs that are written substantially using non-blocking communication.
- People who already have checkpoint-restart in their applications, and want to add finer-grain recovery for additional fault models, perhaps to increase performance, or provide ability to port their applications to systems with diverse fault models and rates.
- Developers of parallel file systems who need resilient MPI functionality to help implement parts of their a) MPI implementation, and/or b) parallel file system.
- Current non-consumers of MPI in application spaces where fault-tolerance is a hard requirement.

Please note that we do not pretend to support all MPI programs, we seek to support MPI programs that use non-blocking communications in the first version of this document.

[6]Such program rewriting engines can clearly also help with "blocking" MPI code based on straightforward transformation to nonblocking + `WAIT` transformations.

4 Basic Requirements and Semantics

The requirements for Fault-Aware MPI are defined as follows:

1. MPI is extended to enable a transactional model, to allow a group of operations to be "tried" and then "committed" or "rolled back."

2. MPI is extended minimally to achieve these goals.

3. MPI implementations may perform extra work, in "Transactional blocks"[7] to help fault aware applications

4. Most of the onus for "resilience" lies with the application still, MPI helps where absolutely needed, and otherwise doesn't hinder. They must perform extra work at "commit" to ensure global error propagation within the surviving members of a communicator's group of tasks.

5. MPI Supports message passing, 1-sided, and I/O with the same kind of awareness mechanisms and error reporting approaches.

6. Programs written using non-blocking communication, I/O, and 1-sided operations are supported at a minimum (point to point and collective modes). Non-communicating, blocking operations like `MPI_Comm_rank()` are also allowed (local operations).

7. MPI programs can get normal, local completion and local error information within transactional blocks, and group-wide ("global to communicator scope") error information collectie when request, and in particular at the conclusion of transactional blocks.

8. No race conditions involving fault state or lack thereof is created in the new interface. A program never tests for "being in good shape" and then use that information in a subsequent call to make progress. Rather, programs attempt functions that may or may not succeed. Only a few "hardened" operations provide synchronized error guarantees, not the entire MPI API.

9. Error return codes may be augmented to support reporting of new classes of faults associated with operations.

10. MPI fault aware functionality will support timeouts to support application-directed granularity for trying (and failing) a transaction. There is no requirement to "wait forever," although that is allowed at an application's discretion.

11. MPI fault aware functionality shall include WAIT and TEST functionality augmented with specifiable timeouts.

12. In the limit of relatively few faults, applications can adapt this functionality provided here to reduce fault-free overhead, and when there are scenarios of relatively higher faults, they can likewise adapt. Adaptation at runtime should be possible.

13. The timeout granularity shall be no coarser than the unit `MPI_Wtick()`.

[7] Called `TryBlocks` in the proposal that follows.

When there are no faults on a communicator (respectively window, or file), then communication (respectively 1-sided operations, and I/O) operate as they do in MPI-3. There may be fault-free overhead in a practical implementation, in addition to the overhead of the operations described here. When used outside of the described transactional interface, there is no expectation of fault-friendly behavior in the MPI implementation.

The following semantics are offered regarding faults observed by MPI or reported to MPI by a user application:

1. Faults are conditions where the MPI-2 guarantee of reliability cannot be met by MPI. Task[8] failure, inability to communicate within a reasonable amount of time, and detectable communication errors can all raise faults within an MPI implementation. Furthermore, tasks are permitted to raise faults locally within an MPI implementation that have the same effect as faults detected by MPI.

2. A sufficient condition for a send/receive communication to be "active" is for MPI to have no fault condition associated with either the sender or receiver task in the group of the communicator associated with the operation. Otherwise, if errors are fatal, then a fatal error occurs. If errors return on that communicator, an error code is produced. Both intra-communicators and inter-communicators are supported.

3. A sufficient condition for a collective communication to be "active," is for MPI to have no fault conditions associated with the group of the associated communicator (as observed locally). Otherwise, if errors are fatal, then a fatal error occurs. If errors return on that communicator, an error code is produced. Both intra-communicators and inter-communicators are supported.

4. A sufficient condition for a 1-sided (point-to-point) operation[9] to be active is for MPI to have no fault condition associated with either the source or target window rank of the group associated with the Window. Otherwise, if errors are fatal, then a fatal error occurs. If errors return on that communicator, an error code is produced.

5. A sufficient condition for a non-collective I/O operation to be active is for MPI to have no fault condition associated with the local task initiating the operation nor with the File. Otherwise, if errors are fatal, then a fatal error occurs. If errors return on that communicator, an error code is produced.

6. A sufficient condition for a collective I/O operation to be active is for MPI to have no fault condition associated with with the group on which the current File is based.

7. An application task may raise a fault concerning itself or another rank or ranks ("discommend") in the group of a communicator, window, or file. MPI will trust this information, and use such fault information as if it were detected by MPI itself. An application will be able to specify at least one type of portable fault condition, and implementations may provide other fault conditions. After an application task specifies such a fault, local behavior with regard to the communicator, window, or File respectively will meet the requirements stated above. Discommendation is irreversible.

[8] We use "task" to connote an MPI-2-style process, and to allow for possible extension to other entities such as threads in other MPI-3 work. Read "process" everywhere we write "task" if you prefer to think in current terminology.

[9] Passive target mode needs further discussion.

"Active" communications are attempted. Otherwise either an error code is raised (for errors return), or the MPI task takes a fatal error. Both behaviors may be desirable in certain uses of the fault aware implementation, although we will commonly use errors return. Active communications will produce a request handle as an output, but are allowed to fail on return, if a fault is observed by MPI during their initiation.

The progress of faults between members of a communicator represents an opportunity for either passive or active progress of fault information. MPI implementations may choose to actively progress fault information between members of a process group, use piggybacking of fault information, or simply wait for specific fault-aware operations to be performed before sharing fault information within a group of tasks. The level of fault progression required by an MPI implementation is not defined here.

5 Fault-Aware Programming Model

Pseudo-code presented in this section shows a Fault-Aware MPI program accomplished with the mechanisms of Fault-Aware MPI. We have exposed hierarchical retry: soft, ABFT-enabled recovery, checkpoint/restart enabled-recovery, and application-restart (worst case), as four increasingly expensive levels of recovery based on fault environment experienced by the application.

Two parts: Communication instantiation, and communication mitigation, are detailed further after the discussion of the main program block. Likewise, it is presumed that any I/O block or CPR library also is implemented in a `TryBlock` and exploits `TryBlock` error state propagation and potentially communication recovery: a sequence of fault-aware/friendly barriers appear appropriate to support error propagation [15].

We don't need to delve into out-of-band mechanisms as part of this effort; we presume that they will appear under the covers in the implementation or as part of what users are doing on their own as part of recovery. For example, gossip between progress engines is a means for asynchronous error propagation as well, although active fault-progression is not mandated by Fault-Aware MPI. Piggybacking error state with the implementation or application's point-to-point communication appears useful too. Piggybacking and gossip are implementation choices for the MPI library and also for the user program. However, we don't do anything to specify or standardize those here. Other parts of MPI-3 do offer help with piggybacking, which is convenient for users who might wish to avail themselves of that capability.

5.1 A quick sketch of a Fault-Aware MPI Application

The pseudo-code below shows the main concepts of Fault-Aware MPI:

- Communication Instantiation
- I/O associated either with a restart or normal I/O read; protected by a TryBlock
- A sequence of TryBlock_start() ...TryBlock_waitall()
- Recovery after each Block, if needed
- Iteration over TryBlocks and Recovery until done
- Synchronized Deinitialization of Communication before program termination

```
MPI_Init()
...
/* Communication Instantiation with Fault Recovery; This is in a TryBlock */

loop_over_program_work /* process R total operations, N units of work per iteration */
{
   /* do any needed application I/O (initialization, restart, next data read...);
      This is in a TryBlock w/ retry loop  */

   /* Local Data Moves in Anticipation of Transaction */

   /* Transactional Steps through the program iteration to deliver its service */

   TryBlock_start(); /* synchronizing collective operation that admits or rejects the
                        start of the block */

   /* HERE: Non-blocking MPI calls and user local operations. There can
      also be local completions, fault injections based on ABFT discovery
      of problems with data in MPI messages, etc.  */

   TryBlock_waitall(); /* synchronizing collective operation that reports
                          consistent error state to all participants */

   /* Recovery Procedures, if Needed:
      a) move MPI to a new consistent state for communication
      b) reorganize how the application works so it can continue;
      can be ABFT or punt to restart at next iteration

      otherwise: local data moves (if any) at completion of the transaction
   */

   /* Do occasional checkpoints in a TryBlock; fail forward or backward,
      depending on application preference */

}

/* Synchronized Deinstantiation of Communication Is Ideally Done here;
   This is in a TryBlock */

MPI_Finalize();
```

A comparatively more detailed pseudo-code example is provided for this program organization in the next subsection.

Unrolling the work comprised in the `N` elements is appropriate, in order to control the amount of time in a transaction and support the limit $\tau_{\mathrm{Re}} \times R_{\mathrm{F}} \ll 1$; for instance, purely vector operations or data parallel operations fit this model. In this first version, we unroll just the middle of the application:

```
MPI_Init()
...
/* Communication Instantiation with Fault Recovery; This is in a TryBlock */
```

```
outer_loop_over_program_work /* process this loop R times */
{
   /* do any needed application I/O (initialization, restart, next data read...);
      This is in a TryBlock w/ retry loop  */

   unrolled_loop_over_program_work /* process this loop K times */
   {
      inner_loop_over_program_work /* process floor(N/K) or ceil(N/K) units of
                                      work per iteration, depending which iteration */
      {

         /* Local Data Moves in Anticipation of Transaction */

         /* Transactional Steps through the program iteration to delivery it service */

         TryBlock_start() /* synchronizing collective operation that admits
                             or rejects the start of the block */

         /* HERE: Your parallel application logic (communication, computation) */

         TryBlock_waitall(); /* synchronizing collective operation that reports
                                consistent error state to all participants
                                + free descriptors */

          /* Recovery Procedures, if Needed:
             a) move MPI to a new consistent state for communication
             b) reorganize how the application works so it can continue

              otherwise: local data moves (if any) at completion of the transaction
          */
      }

      /* Do occasional checkpoints in a TryBlock; fail forward or backward,
         depending on application preference */
   }
}

/* Synchronized Deinstantiation of Communication Is Ideally Done here;
   This is in a TryBlock */

MPI_Finalize();
```

If after `N` items are processed, additional communication is required, then this is accomplished with a two-level transaction; this version unrolls everything:

```
MPI_Init()
...
/* Communication Instantiation with Fault Recovery; This is in a TryBlock */

outer_loop_over_program_work /* process this loop R times */
{
```

```
   TryBlock_start(); /* synchronizing collective operation that admits or
                        rejects the start of the block [outer] */

   unrolled_loop_over_program_work /* process this loop K times */
   {
      /* do any needed application I/O (initialization, restart, next data read...);
         This is in a TryBlock */

      inner_loop_over_program_work /* process floor(N/K) or ceil(N/K) units
                                      of work per iteration, depending which iteration */
      {
         /* Local Data Moves in Anticipation of Transaction */

         /* Transactional Steps through the program iteration to deliver its service */

         TryBlock_start() /* synchronizing collective operation that admits
                             or rejects the start of the block */

         /* HERE: Your parallel application logic (communication, computation) */

         TryBlock_waitall(); /* synchronizing collective operation that
                                reports consistent error state to all participants
                                + free descriptors */

          /* Recovery Procedures, if Needed:
             a) move MPI to a new consistent state for communication
             b) reorganize how the application works so it can continue

             otherwise: local data moves (if any) at completion of the transaction
           */
      }

      /* Do occasional checkpoints in a TryBlock; fail forward or backward,
         depending on application preference [optional location] */
   }
   /* Additional Communication Operations Keyed to Outer Tryblock
      (e.g., norms and allreduces) */

   TryBlock_waitall();
   /* If successful, local Data Moves After Transaction + free descriptors

      Otherwise, do recovery */

   /* Do occasional checkpoints in a TryBlock; fail forward or backward,
      depending on application preference [optional location] */
}

/* Synchronized Deinstantiation of Communication Is Ideally Done here;
   This is in a TryBlock */

MPI_Finalize();
```

Here's another way to look at the multiple levels of transactions, emphasizing three phases of operations in the inner transaction, but omitting unrolling for simplicity:

```
MPI_Init()
...
/* Communication Instantiation with Fault Recovery */

loop_over_program_work /* R times needed to complete application;
                          process N units of work per iteration */
{

   /* Local Data Moves in Anticipation of Multi-Transaction */
   loop_over_multi_transaction
   {
      TryBlock_start(); /* synchronizing collective operation that
                           admits or rejects the start of the block [outer] */

      /* READ-CENTRIC: */
      TryBlock_start() /* synchronizing collective operation that
                          admits or rejects the start of the block */

      /* HERE: Non-blocking MPI calls and user local operations. */

      TryBlock_waitall(); /* synchronizing collective operation that reports
                             consistent error state to all participants
                             + free descriptors */

       /* Recovery Procedures, if Needed: punt to outer level */

      /* Local operations, and 1-sided, and communication only: */
      TryBlock_start(); /* synchronizing collective operation that
                          admits or rejects the start of the block */

      /* HERE: Non-blocking MPI calls and user local operations. */

      TryBlock_waitall(); /* synchronizing collective operation that
                             reports consistent error state to all participants
                             + free descriptors */

      /* Recovery Procedures if Needed: punt to outer level */

      /* Local Data Moves in Anticipation of Transaction */

      /* WRITE-CENTRIC: */
      TryBlock_start(); /* synchronizing collective operation that
                          admits or rejects the start of the block */

      /* HERE: Non-blocking MPI calls and user local operations. */

      TryBlock_waitall(); /* synchronizing collective operation that
                             reports consistent error state to all participants
                             + free descriptors */
```

```
      /* Recovery Procedures, if Needed: punt to outer level */

      TryBlock_waitall(); /* synchronizing collective operation that
                             admits or rejects the end of the block [outer] +
                             free descriptors  */

      /* Recovery Procedures if Needed:
         a) move MPI to a new consistent state for communication
         b) reorganize how the application works so it can continue
         c) use ABFT and/or CPR to get application back to a consistent state
         d) repeat outer transaction try block
      */
   }
   /*
        Success: local data moves (if any) at completion of the multi-transaction
   */
}

/* Synchronized Deinstantiation of Communication Is Ideally Done here */

MPI_Finalize();
```

Note that timeouts are for the individual `_start` and `_wait` operations, not for the durations of the sequence of operations in between the `_start` and `_wait`. We cover the concept of limiting the time of an entire block separately, toward the end of this paper.Providing a means for how to merge the last two `TryBlock_waitall` operations is also an optimization goal; we provide API for that below. The reader will also notice that the `TryBlock_start()` and `TryBlock_wait()` pseudo-operations are blocking. Non-blocking discussion involving these functions are described in the API section.

Meeting the Iron-Law limit of granularity remains important, so if this nested transaction experiences a significant probability of failures, and can't make progress, an unrolling concept would be needed (work on smaller amounts of data per transaction). As with timeouts, sizing this granularity is part of porting a Fault-Aware MPI application to a new platform.

5.2 Working with Nonblocking Synchronizations and Multi-level Transactions

Here are the main ideas we want to note about Fault-Aware MPI and multi-level parallel programs:

- We can do non-blocking synchronizations, we can hopefully overlap transactions with others, in the same parallel thread or in different parallel threads.
- multi-level transactions support both data parallel, nested parallel, and layered parallel codes.
- Our model is more general than BSP, which enforces blocking barriers.
- We know a lot about local completion within `TryBlocks`, we only enforce global errors when we choose (and then we mitigate, rollback or rollforward).

The `TryBlock` framework at two levels provides a general model of task and data parallel computation and communication. A loop over the outer `TryBlock` represents global program

progress. Internally, there can be several blocks within that outer block, each consisting of code blocks of data or task parallel code. The blocks can be run sequentially and in parallel within the outer TryBlock. Some of these groups of tasks will succeed, while others fail. The outer `TryBlock` "sifts" out success and failure. The application fails forward (partial success) when it can, and repeats work it has to when needed. For situations where all the groups of tasks interact strongly, the application retains the right to keep good work, or repeat all the work. Furthermore, sequentially repetitive work or parallel repetitive work in order to perform voting or local recovery is also doable within this framework. The amount of work attempted within the outer `TryBlock` can be adjusted in granularity via feedback obtained in the fault results (progress or lack thereof). This permits, in principle, for applications to be self-tuning with regard to the fault environment.[10]

6 Detailed Program Pseudo Code

Here we sketch a hierarchically recovering program, while omitting the details of communication instantiation, and communication recovery after a fault. Those follow in the next subsections.

6.1 Main Program Pseudo Code

Here is the pseudo-code for the main program structure.

```
main(int argc, char **argv)
{
   int code = 0;

   MPI_Comm comm;
   MPI_File file;
   MPI_Win win;
   int local_error;
   int sync_error;

   int N_requests;
   MPI_Status *statuses = 0;  /* will be an array of statuses */
   MPI_Request *requests = 0; /* will be an array of requests */

   int restart_needed = 0;
   int initialized = 0; /* these two variables reflect possible need for a restart phase */

   if((local_error = MPI_Init(&argc, &argv)) != MPI_SUCCESS)
   {
      /* this case is not discussed further in the first draft of Fault-Aware MPI */
   }
   MPI_Errhandler_set(MPI_COMM_WORLD,MPI_ERRORS_RETURN);

   do  /* Heavy-weight restart loop (could be check-point restart) */
   {
      /*
```

[10] Any self-tuning in a highly dynamic environment offers the potential for chaotic effects, so this requires further study in any real application scenario. Applications will have to filter the feedback to avoid overcorrecting or undercorrecting granularity of work.

```
   COMMUNICATION INSTANTIATION
   ---------------------------
   MPI_COMM_WORLD exists here, although it may have faults already:

   Build communicators needed for the application using MPI_Comm_split().

   We will document the fault-aware handling of this separately,
   because it could fail

   comm, file, and win are all created here if there are no failures

   We will deep dive this part of the application separately,
   it needs a recovery mode too.

   We may need to specify the implicit comm/group of Windows and Files to do
   certain operations.
*/

MPI_Errhandler_set(comm,MPI_ERRORS_RETURN);

if(!initialized || restart_needed)
{
   /* use MPI I/O or CPR library to load application state
      before continuing, set more_work_to_do_for_app;
      if this fails, repeat this loop */

   if(error) goto recovery_step;

   initialized = 1;
}

while(more_work_to_do_for_application)
{
   int block_entry = 0; /* tells code if we ever entered block */
   int soft_retry = 0;
   int soft_retry_max = 2;
       /* some application-dependent value (eg., try 2 for SEUs) */

   int recovery_mode = 0;

   do /* this is the soft retry loop.  An ABFT fault only was raised,
         simply redo the loop until we lose confidence in that
         recovery approach */
   {
‘       int numerrors = 0;
      int error_injection = 0;

      MPI_Timeout timeout;
      MPI_Timeout_set_ticks(&timeout,1000000);  /*specify timeout */
                     /* Ex: 1000000 time periods in units of MPI_Wtick() */
```

```
/* locally move data into common buffers for Try_Block: */

/* now try to enter the TryBlock: */
sync_error = MPI_TryBlock_start(comm, MPI_UNSIGNED_INT,
             local_error_injection, group_error_state, timeout, &req);
/* synchronous error outcome;
   a timeout of 0 = infinity */

if(sync_error == MPI_SUCCESS)
{
   block_entry = 1;

   local_error |= MPI_Operation1(comm |
                  window | file, &req[0]);
   ABFT_error_logic0(&error_injection);
      /* user defined logic for their application */

   local_error |= MPI_Operation1(comm |
                  window | file, &req[1]);
   ABFT_error_logic1(&error_injection);
      /* user defined logic for their application */
   ...
   local_error |= MPI_Operation1(comm |
                  window | file, &req[N_requests-1]);
   ABFT_error_logicN_1(&error_injection);
      /* user defined logic for their application */

   /* if we find an error in any operations, but
      local_error is not asserted by any operation,
      then we use fault injection to indicate the
      error noted locally; this is a user-defined
      value (0 is no-error): */

   int try_flag = 0;
   sync_error = MPI_TryBlock_waitall(req, &try_flag, &try_status,
                MPI_UNSIGNED_INT, &local_error_injection, group_error_state,
                timeout, N_requests, reqs, statuses, &numerrors, Error_indices);
   MPI_Request_free(req);
}
else
   block_entry = 0;

/* Recovery Begins Here
   (this is exemplary, not mandatory): */
switch(sync_error)
{
   case MPI_SUCCESS: /* no error */
       if(block_entry == 1)
       {
         /* use completed buffers; note that
             increment progress counter,
```

```
                        iterators, etc */
                    }
                    else
                    {
                       /* can't happen, unmodeled behavior */
                    }
                    break;

                case MPI_LOCAL_ERROR_INJECTION;
                     /* one or more processes
                        raised a non-zero error_injection */


                    soft_retry++;
                    break;

                case MPI_TIMEOUT:
                    /* can adapt timeout, can do soft retry if 0 or 1 */
                    if(block_entry == 0)
                    {
                    }
                    else
                    {
                    }
                    /* alternatively can treat timeout as a hard error,
                       such as if several backoffs have failed */

                    /* may or may not fall through here... */

NB: We need to make the nomenclature and names for all of these consistent!
                case MPI_TASK_FAULT:
                case MPI_COMMUNICATION_FAULT:
                case MPI_UNMODELED_FAULT:
                case MPI_MULTIPLE_FAULTS:  /*(see statuses)*/
                default:
                    /* recovery requires repair of a communicator,
                       and possible backtrack to a checkpoint */
                    recovery_mode = 1;
                    break;
            }

         } while((soft_retry &&
                 (soft_retry <soft_retry_max))
                 && !recovery_mode);

         /* we exceeded soft retry logic,
            resort to heavier-weight recovery */
         if((soft_retry >= soft_retry_max) || recovery_mode)
         {
            recovery_mode = 1;
            break;
```

```
         }

         /* periodically decide to do a checkpoint,
            using CPR library; if this fails,
            you can decide to do recovery, or keep going,
            depends on application and CPR policy */

      } /* end while(more_work_to_do_for_app) */

      if(!recovery_mode) /* application done */
      {
         code = 0;
         break;
      }

recovery_step:

       /*
         If this fails, may decide to restart application from scratch
         [fail stop + restart = most expensive restart]
       */
      if(recovery_mode >= recovery_mode_max)
      {
         code = 1;
         break;
      }

      /*
         COMMUNICATION MITIGATION
         ------------------------
         lost processes, spawn and readd, join to make new comm;
         or else resize for smaller world

         At least temporarily, comm is reorganized to support
         communication if this succeeds.

         If this fails, may decide to restart application from scratch
         [fail stop + restart = most expensive restart] */

         This logic will be detailed separately,
         because it too can fail.
       */

      /* At this point, we have nominally recovered comm;
         which is the communicator we need to proceed... */

      if(ABFT_can_recover_data())
      {
         error = do_ABFT_recover_data(comm);
         if(error) restart_needed = 1;
      }
```

```
      else
      {
         /* use CPR to reconstruct consistent
            state on next pass through */
         restart_needed = 1;
      }

   } while(restart_needed || more_work_to_do_for_app);

   /* We will use a non-fault aware termination for now: */
   MPI_Finalize();

   return code;
}
```

We now turn to presenting resiliency related to the instantiation and mitigation/recovery of communication.

6.2 Communication Instantiation

The theme of Fault-Aware MPI is to attempt operations and see what happens (locally, locally with ABFT, globally with synchronization of the then-surviving group). Supporting robust `MPI_Comm_split` is a key necessity.

```
   ...

   /*
      if this works as desired, MPI_Comm_split_sync() could be seen
      as roughly as 'ease of use' feature, not more nominally capable
      that TryBlock, except that we allow explicit timeouts in the new
      function.

      But, practically speaking, it does appear that a unified API is
      going to have more opportunities for success in faulty situations,
      compared to a two-level API which cannot fail forward inside the MPI
      implementation...
   */

#if NO_NEW_API_FOR_SPLIT_SUPPORTED

   /*
      You can consider code patth to be the rough pseudo-code for one way to
      implement an MPI_Comm_isplit_sync() just using TryBlock +
      MPI_Comm_create_group_sync():
   */

   MPI_Comm comm = MPI_COMM_WORLD;
   do
```

```
{
   TryBlock_start(comm,..., &try_reqest);
                  /* synchronizing collective operation
                     that admits or rejects the start of the block */

   MPI_Comm_isplit(comm, color, key, &newcomm, &req1); /* */

   error = TryBlock_waitall(); /* synchronizing collective operation that reports
                                  consistent error state to all participants */

   if(error != MPI_SUCCESS)
   {

      /* extract remaining group from info in outcome of _waitall(),
         put in remaining_group:

         Is _sync needed, or simply a new set of local error codes,
         and an implementation that survives classes of fault and returns a code?
       */

      error_comm_group_create = MPI_Comm_create_group_sync(comm, 0 /* coll tag */,
                                                remaining_group, timeout, &outcomm);

      switch(error_comm_group_create != MPI_SUCCESS)
      {
         case MPI_SUCCESS:
              break;
         case MPI_GROUP_SHRUNK:
             /* application dependent decision making, worst
                case is to raise failback=1

                otherwise
                a) adapt to given sizes
                b) grow communicators using a spawn+join strategy

                PS Don't forget MPI_Group_translate_ranks() if you need it.
             */
             break;

         default:
             /* unmodeled, unrecoverable state, we need to fail backwards */
             failback = 1;
             break;
      }
      else
```

```
            comm = outcomm;
       }

   } while((error != MPI_SUCCESS) && !failback); /* allow comm to keep shrinking */

#else /* this code path assumes existence of MPI_Comm_isplit_sync(): */

   /* fault-aware split produces the best communicators it can in each color/key: */
   MPI_Timeout timeout;
   ...
   error = MPI_Comm_split_sync(comm, color, key, timeout, &newcomm);

   switch(error)
   {
       case MPI_SUCCESS:
            break;

       case MPI_GROUPS_SHRUNK:
           /* application dependent decision making, worst
              case is to raise failback=1

              otherwise
              a) adapt to given sizes
              b) grow communicators using a spawn+join strategy

              we can also decide to try to repair MPI_COMM_WORLD
              if we need it later, but that's optional.
           */
           break;

       case MPI_TIMEOUT_OCCURRED:
       default:
           /* we can elect to try to repair MPI_COMM_WORLD,
              we can inspect sizes of groups for each color... */
           failback = 1;
           break;
   }

   if(!failback)
   {
       comm = newcomm; /* use this communicator */
   }
   else
   {
      /* restart MPI job; punt to scheduler (or retry script
```

```
        wrapping an mpiexec)! */
   }
#endif
```

Full API details are provided below in Section 7.

6.3 Communication Mitigation/Recovery

In this section, we cover the case where it was insufficient to build-down to a group we could split. We need to regrow the size of communication after a loss of (one or more) member(s) of the group of `MPI_COMM_WORLD`.

The a common use case for fault-free creation of a communicator is as follows:

- Spawn a new group (forms an intercommunicator), remote group has a maximum size
- *merge* the intercommunicator to form a new intracommunicator
- "Forget about the intercommunicator"; use the intracommunicator

The spawning functions are as follows:

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info,
                   int root, MPI_Comm comm, MPI_Comm *intercomm,
                   int array_of_errcodes[]);

int MPI_Comm_ispawn(char *command, char *argv[], int maxprocs, MPI_Info info,
                   int root, MPI_Comm comm, MPI_Comm *intercomm,
                   int array_of_errcodes[], MPI_Request *request);
```

and the more generalized forms:

```
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int root,
                            MPI_Comm comm, MPI_Comm *intercomm,
                            int array_of_errcodes[]);

int MPI_Comm_ispawn_multiple(int count, char *array_of_commands[],
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int root,
                            MPI_Comm comm, MPI_Comm *intercomm,
                            int array_of_errcodes[], MPI_Request *request);
```

and produce, as noted, an intercommunicator; the *root* rank has the arguments, a source of (additional) potential problems in the fault-aware scenario. The remote group is born equivalently to having been created with an `mpiexec`-type command, and is in principle an independent MPI

program at this point, except for its added intercommunicator relationship with the parent group. MPI-3 adds support for the non-blocking forms.

The function we most often want to apply after the spawn is the merging function, to get all communicating processes into a group:

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm);

int MPI_Intercomm_imerge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm,
                         MPI_Request *request);
```

This allows two groups to merge, but does nothing for the very interesting case of three or more groups. (The difficulty of merging $N \geq 3$ groups is well established, and in fact is a tricky piece of MPI-2 code. We don't address that concern in this paper. More API in this area is needed with or without faults.) Again, MPI-3 has added the non-blocking form needed for this proposal.

To begin with, we always assume that we can put a `TryBlock` around any MPI non-blocking function and retry it till we either get an acceptable result, or decide to backoff because MPI is unable to succeed in a reasonable way. Because spawns are able to return less resources than requested, we can start with this approach:

```
#ifdef NO_NEW_API_FOR_SPAWN_MERGE_SUPPORTED
   MPI_Timeout timeout;
   int failure_occurred = 0;
   do
   {

      MPI_TryBlock_start(comm);
      err1 = MPI_Comm_ispawn(...comm, intercomm,.... &req[0]);
      /* optionally do a local_wait() here to promote progress */
      err2 = MPI_Comm_intercomm_merge(...intercomm..., &req[1]);
      MPI_TryBlock_waitall();

      if(failure_occurred)
      {
         /* OK, if the input communicator has become inactive,
            we need to recover, and retry... */
         error = MPI_Comm_split_sync(comm, 0 /*color*/, 0 /*key*/,
                                     timeout, &newcomm);

         /* deal with an errors, choose new root for arguments, etc */

         MPI_Comm_free(comm);
         comm = newcomm;
       }
    } while(failure_occurred || !you_decide_to_punt_to_a_higher_level_of_recovery);


#else
```

```
    MPI_Comm_spawn_and_merge_sync(...);
    /* use unless you decide you can't get a bigger communicator */
#endif
```

If this succeeds, an intracommunicator of size greater than the original communicator is formed. However, the `TryBlock` alone not robust to faults in the input communicator, particularly not the loss of the root. We rely on recovery from the `MPI_Comm_split_sync()` function described above.

The full Fault-Aware API is covered in the next section.

7 Fault-Aware API Specification

In this draft, we use C-bindings to specify the proposed new syntax. Language-independent specifications will be provided in future updates.

7.1 Timeout API, Types

```
MPI_Timeout timeout;  /* a timeout non-opaque object,
                         measured in units of MPI_Wtick() */

int MPI_Timeout_set_ticks(MPI_Timeout *timeout, MPI_Aint ticks);
                           /* set in units of MPI_Wtick() */
int MPI_Timeout_set_seconds(MPI_Timeout *timeout, double &usec);
                           /* set in usec */

int MPI_Timeout_get_ticks(MPI_Timeout timeout, MPI_Aint &ticks);
                           /* set in units of MPI_Wtick() */
int MPI_Timeout_get_time(MPI_Timeout timeout, double &usec);
                           /* read in usec */
```

7.2 `TryBlock` API

The synchronizing forms of the `TryBlock` API for the beginning and ending of a `TryBlock` follows, where these allow combination of values in a gather-style[11]:

```
    int MPI_TryBlock_start(MPI_Comm comm, MPI_Datatype type,
                           void *local_error_injection, void *group_error_state,
                           MPI_Timeout timeout, MPI_Request *try_request);

    int MPI_TryBlock_waitall(MPI_Request try_request, int *try_flag,
                             MPI_Request *try_status,
                             MPI_Datatype type, void *local_error_injection,
                             void *group_error_state, MPI_Timeout timeout,
                             int count, MPI_Request array_of_requests[],
                             MPI_Status array_of_statuses[], int *numerrors,
```

[11] If this proposal is found to be promising, we will add an allreduce-style form of the wait operations too, which follows as a logical alternative to the allgather forms proposed.

```
                            int Error_indices[]);

int MPI_TryBlock_Waitsome(MPI_Request try_request, int *try_flag,
                          MPI_Request *try_status,
                          MPI_Datatype type, void *local_error_injection,
                          void *group_error_state, MPI_Timeout timeout,
                          MPI_Timeout timeout, int incount,
                          MPI_Request array_of_requests[], int *outcount,
                          int array_of_indices[], MPI_Status array_of_statuses[],
                          int *numerrors, int Error_indices[]);
```

Note to ourselves: Does a waitsome make sense? We want to finish the transaction or not. Testing can make perfect sense.

The nonblocking forms of these operations are defined as follows:

```
int MPI_TryBlock_istart(MPI_Comm comm, MPI_Datatype type,
                        void *local_error_injection, void *group_error_state,
                        MPI_Timeout timeout, MPI_Request *try_request);

int MPI_TryBlock_testall(MPI_Request try_request, int *try_flag,
                         MPI_Request *try_status,
                         MPI_Datatype type, void *local_error_injection,
                         void *group_error_state, MPI_Timeout timeout,
                         int count, MPI_Request array_of_requests[], int *flag,
                         MPI_Status array_of_statuses[], int *numerrors,
                         int Error_indices[]);

int MPI_TryBlock_testsome(MPI_Request try_request, int *try_flag,
                          MPI_Request *try_status,
                          MPI_Datatype type, void *local_error_injection,
                          void *group_error_state, MPI_Timeout timeout,
                          int incount, MPI_Request array_of_requests[],
                          int *outcount, int array_of_indices[],
                          MPI_Status array_of_statuses[], int *numerrors,
                          int Error_indices[]);
```

Furthermore, we provide a means to complete multiple nested transactions at once:

```
int MPI_TryBlock_multiple_waitall(int N_requests,
                          MPI_Request try_request_array[],
                          int try_flag_array[],
                          MPI_Request try_status_array[],
                          MPI_Datatype type,
                          void *local_error_injection,
                          void *group_error_state,
                          MPI_Timeout timeout,
                          int count,
```

```
                                MPI_Request array_of_requests[],
                                MPI_Status array_of_statuses[],
                                int *numerrors, int Error_indices[]);
```

This allows a program to do a sequence of `TryBlocks` and then reject all of them later.

For situations where failure rates are low, such strategies could help reduce overheads. Furthermore, when programs have nested parallelism, such as separate alternative tasks in different subgroups of a communicator, then the outer transaction is used to see the progress of these independent groups. Each can pass its own transaction, but all may not pass the outer transaction. At that point, application logic comes into play: you can retain the successful work and redo the other work, or have all processes backup... those decisions remain with the application.

We want to allow programs to complete communication inside a `TryBlock`. If there is no problem with it, then the requests will pass a subsequent `TryBlock` synchronization on those requests. Here the ability to test and wait on requests, for local completion, with timeouts is added to MPI, presented as follows:

```
int MPI_Test_local(MPI_Request *request, int *flag, MPI_Timeout timeout,
                   MPI_Status *status);

int MPI_Testall_local(int count, MPI_Request array_of_requests[], int *flag,
                      MPI_Timeout timeout, MPI_Status array_of_statuses[])

int MPI_Testsome_local(MPI_Timeout timeout, int incount,
                       MPI_Request array_of_requests[],
                       int *outcount, int array_of_indices[],
                       MPI_Status array_of_statuses[])

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

int MPI_Waitall_local(MPI_Timeout timeout, int count,
                      MPI_Request array_of_requests[],
                      MPI_Status array_of_statuses[]);

int MPI_Waitsome_local(MPI_Timeout timeout, int incount,
                       MPI_Request array_of_requests[], int *outcount,
                       int array_of_indices[], MPI_Status array_of_statuses[]);
```

However, unlike normal `test` and `wait`, these do not "destroy" the requests. The requests may be can waited or tested again for global error completion notification, or simply freed with `MPI_Request_free`. In this way, the local completions serve the same role as they would in the fault-free MPI model, but allow the further test at transaction commit time. Thus, single `TryBlocks` can mix communication initiation, local completion, and computation.[12]

These report on the completion of requests in the normal way; Fault-Aware MPI may provided enhanced ability to detect errors for both this and the non-timeout versions of `Wait` and `Test`.

[12]It remains the user's task to recover after a transaction fails, so it is presumed that all work in a transaction happens on temporary data (in each task), which is only moved after a transaction commits.

These are a superset of their MPI-2 counterparts. When a zero timeout is specified, they produce exactly the same behavior as their MPI-2 counterparts. Arguments, with the exception of the new timeout argument, exactly correspond to the MPI-2 counterparts as well.

The `try_request` represents the collective communication block. The other requests specified to _waitall, _testall, _waitsome, and _testsome represent operations started during the `TryBlock`'s scope. The `try_request` can be reused until it is explicitly requested to be freed using `MPI_Request_free()`.

Note: It is our intention to include `MProbe()` variants as well in a future draft. The current problem is that `MProbe()` chose not to work with requests, but rather invented a "message" as a concept. So, this operation will not mesh immediately with the approach given here, and investigation is needed on how to proceed, or if simply to provide an alternative solution for `MProbe` and recommend that to the Forum.

7.3 Additional TryBlock API

`TryBlocks` have the expectation that all collective operations within them will be issued by the user program (in consistent order per normal MPI rules), in order to maintain consistency in the fault-free case. A synchronized way of breaking free of this requirement is useful, but requires additional API. API to allow for testing of the failure of a `TryBlock` in the middle is useful:

```
/* group variants */
int MPI_TryBlock_group_sync(MPI_Group group, MPI_Request try_req,
                        MPI_Timeout timeout);

int MPI_TryBlock_group_isync(MPI_Group group, MPI_Request req,
                        MPI_Timeout timeout, MPI_Request *out_req);

/* comm variants */
int MPI_TryBlock_comm_sync(MPI_Comm sync_comm, MPI_Request try_req,
                        MPI_Timeout timeout);

int MPI_TryBlock_comm_isync(MPI_Request sync_comm, MPI_Request try_req,
                        MPI_Timeout timeout, MPI_Request *out_req);
```

In the group variants, this is a synchronizing collective over the group `group` which must be the same group as `comm` used to instantiate the `Tryblock`, a congruent group to that `comm`'s group, or a subset thereof. The request `req` is the request created by the previous `TryBlock_start` issued at the start of the block. By extension, for the case where the communicator-style syntax is used, the synchronization is over the `sync_comm`, which must be equivalent to the `TryBlock` request's communicator, or a subset thereof. For both variants, blocking and nonblocking variants are provided. The non-blocking form allows for a future sampling of global error state, while allowing for optimistic continuation of computation and communication prior to sampling that state... this is a highly desired mode of operation when the program operates in a relatively low fault regime within the granularity of its `TryBlock`.

These operations are more than a Barrier, they comprise a fault-aware barrier on the state of the relevant group's ability to continue communicating under the communicator `comm` that create `req`. It must be called loosely synchronously in all members of `group`. Furthermore, if the group is

empty, or contains only the local task's rank, this operation has the effect of returning only locally known errors at the time it is issued. Remember that all the fault-aware API does not destroy the relevant requests, they are still valid for further use in subsequent `_testall/_waitall` type operations.

The notion of a non-blocking synchronization is like a non-blocking barrier; you can ask for the answer later, so as to allow for the potential to overlap work (communication, computation, and/or I/O) with the operation. Since this is a pure overhead of fault-tolerance, hiding the cost of the operation appears valuable. It can be completed with a normal `wait`, or one of the newly proposed local waits, with no difference in functionality, since the completion is local, and the operation is required to be internally synchronizing.

You can only use these operations to make a negative decision about the state of the system. In other word, an error here is sufficient to allow a transaction to be collectively aborted by a group of MPI tasks.

7.4 Local Error-assertion API

Because ABFT or other application-oriented operations may detect errors in communication after local completion thereof, we provide a simple means for an application to mark a communication as bad, for future propagation when the transaction attempts to complete:

```
int MPI_Request_raise_error(MPI_Request request, int code);
```

We provide at least two error code values `MPI_Error_incorrect_results`, `MPI_Error_unknown`. This API is how an MPI task informs MPI that a subsequent `TryBlock` synchronization should do error propagation with regard to the particular MPI operation across the group involved.

Note that using `MPI_Cancel` on a request can produce an error locally or remotely, or both. Locally, this is a kind of recoverable error that is localized to the request, not to the communicator or group. In that sense, `MPI_Cancel` is both a request to abandon efforts to complete an operation, and a local error assertion on a request.

We should also support direct error assertion on a communicator, window, and/or file. The purpose of that would be to allow the user program to raise an assertion out-of-band (*e.g.,* I know that collective I/O on this communicator should be invalid). It would also allow local fault injection for experimental purposes in recovery, because otherwise it could prove difficult to test the actual resiliency of applications until faults occur, and not to be able to test behavior systematically prior to production mode deployment.

```
int MPI_Comm_raise_error(MPI_Comm comm, MPI_Error error);
int MPI_Win_raise_error(MPI_Win win, MPI_Error error);
int MPI_File_raise_error(MPI_File file, MPI_Error error);
```

Here, we at least have to allow for `MPI_Error_Invalidate_Collective`, `MPI_Error_User_assertion` and `MPI_Error_unknown`, all of which can then be interpreted later at the `TryBlock` completion (*e.g.,*, `_waitall`). As you will see next, the latter two are syntactic sugar, because we are going to allow the extraction of a reference to the underlying communicator of Windows and Files.

Generally speaking, we access to use the underlying ("hidden") communicator of Windows and Files in the same way as we would the communicator of a normal communication. This accessor is offered:

```
int MPI_File_get_comm(MPI_File file, MPI_Comm *comm);
int MPI_Win_get_comm(MPI_Win win, MPI_Comm *comm);
```

Communicators obtained in this way are just useful for error propagation; you aren't allowed to start using them for general MPI communication. However, they can be used legally as input to `MPI_Comm_isplit_sync`, `MPI_Comm_create_group_sync`, and other MPI operations where they form the template for further communication creation (such as for recovery). This states specific additional requirements on communicators embedded within Files and Windows within a Fault-Tolerant MPI implementation. You have to be able to extract a reference to them; and you may use them in communication-creation, window-creation, and file-open operations.

A further API allows a user-process to invalidate a single rank locally:

```
int MPI_Comm_rank_raise_error(MPI_Comm comm, int rank, int error);
int MPI_Comm_ranks_raise_error(MPI_Comm comm, int numranks, int rank_list[],
                               int list_of_errors[]);
```

This has the effect of locally invalidating point-to-point communication on the communicator to that rank, but not on any other copy of that communicator, window, or file. You have to raise them all individually. There is no inheritance of rank failure from this function. There is no guarantee that this local error will be propagated with progress-type semantics until the TryBlock synchronization step. Multiple calls to the operation have no effect, and the effect is irreversible. Choosing your own rank invalidates all point-to-point and collective communication on that communicator (locally only). Notice that at least four errors are provided: `MPI_Error_unknown`, `MPI_Error_rank_invalidate` (general error), `MPI_Error_task_down`, `MPI_Error_user_assertion`.

If the communicator is an intercommunicator, the meaning is changed to be a remote rank, and then you cannot self-invalidate using this operation.

If the goal is to invalidate all communication with a rank, for all groups and collective operations, then this operation represents a potentially heavier-weight (although still local) declaration to the MPI implementation:

```
int MPI_Comm_rank_raise_forall(MPI_Comm comm, int rank, int error);
int MPI_Comm_ranks_raise_forall(MPI_Comm comm, int numranks,
                                int rank_list[], int list_of_errors[]);
```

It uses the rank name in the specified communicator to look up the related task as it is used throughout the local MPI implementation, and marks all communication objects invalid. Although local, this is cross-cutting. This function is helpful as part of mitigation, isolation, and recovery, but may be controversial... it is also not absolutely required, since cooperating fault-aware programs will have to know about resource utilization and layering a lot more subtly than non-fault-aware programs. It is a way for a layered application to alert other layers with which it is not formally connected about errors it detects locally, so it comprises a kind of application-vertical assertion capability without requiring interrupt handlers.

7.5 Communication Recovery API

Because communication failures (whether from dead processes, bad links, high error rates, or other transient and permanent failure phenomena) may cause MPI or the application to mark as remote task in a group as bad, we need a straightforward means to continue communication with a smaller

communicator, and also provide a means for rebuilding communicators. Furthermore, we want to be able to reduce the complexity of managing unneeded global communicator state, depending on application structure, so that each recovery does not demand rebuilding all communicators in the system, particularly MPI_COMM_WORLD, unless the application needs to do so to meet its requirements.

7.5.1 Split and Group_Create

The MPI_Comm_create_group() function proposed elsewhere in MPI-3 is helpful in moving from a communicator with failed or marked-bad tasks, to a subset communicator base on a group not currently known to be bad. We employ the form that currently includes a *collective tag*, but we characterize our reliance on that feature, in case it is eliminated in future Forum discussions.

Here we discuss the limits to which we can push the currently proposed operation, and how we can merge that operation with TryBlock_[I]Sync(), to produce a recovery operation that is strictly "shrinking".

We propose this function set as minimal to supporting fault-recovery of intracommunicators:

```
/* no error synchronization over communicator (use in TryBlock) */
int MPI_Comm_icreate_group_local(MPI_Comm comm, int tag, MPI_Group group,
                                 MPI_Timeout timeout, MPI_Comm *outcomm,
                                 MPI_Request *request);

/* with error synchronization over communicator */
int MPI_Comm_create_group_sync(MPI_Comm comm, int tag, MPI_Group group,
                                 MPI_Timeout timeout, MPI_Comm *outcomm);
int MPI_Comm_icreate_group_sync(MPI_Comm comm, int tag, MPI_Group group,
                                 MPI_Timeout timeout, MPI_Comm *outcomm,
                                 MPI_Request *request);

/* use only in timeout-only scenario [for design orthogonality]: */
int MPI_Comm_create_group_local(MPI_Comm comm, int tag, MPI_Group group,
                                  MPI_Timeout timeout, MPI_Comm *outcomm);
```

Furthermore, we propose this new function set as central to fault-recovery of intracommunicators:

```
/* no error synchronization over communicator (use in TryBlock) */
int MPI_Comm_isplit_local(MPI_Comm comm, int color, int key,
                          MPI_Timeout timeout, MPI_Comm *newcomm,
                          MPI_Request *request);

/* with error synchronization over communicator */
int MPI_Comm_split_sync(MPI_Comm comm, int color, int key,
                        MPI_Timeout timeout, MPI_Comm *newcomm);
int MPI_Comm_isplit_sync(MPI_Comm comm, int color, int key,
                         MPI_Timeout timeout, MPI_Comm *newcomm,
                         MPI_Request *request);

/* use only in timeout-only scenario [for design orthogonality]: */
```

```
int MPI_Comm_split_local(MPI_Comm comm, int color, int key,
                         MPI_Timeout timeout, MPI_Comm *newcomm);
```

We expect that "splits" will be the workhorse intracommunicator recovery function family of most applications.

We are considering whether or not to add an `MPI_Comm_dup_sync` function. This seems to be unneeded, inasmuch as `split` functions can do everything except propagate info arguments.

7.5.2 DPM: Spawn and Spawn+Merge API

The following new API for creating intercommunicators is proposed[13]:

```
/* no error synchronization over communicator (use in TryBlock) */
int MPI_Comm_ispawn_local(char *command, char *argv[], int maxprocs, MPI_Info info,
                  int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *intercomm, MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request);

int MPI_Comm_ispawn_multiple_local(int count, char *array_of_commands[],
                           char* *array_of_argv[], int array_of_maxprocs[],
                           MPI_Info array_of_info[], int numroots,
                           int list_of_roots[], MPI_Comm comm, MPI_Comm *intercomm,
                           int array_of_errcodes[], MPI_Request *request);

/* with error synchronization over communicator */
int MPI_Comm_ispawn_sync(char *command, char *argv[], int maxprocs, MPI_Info info,
                  int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *intercomm, MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request);

int MPI_Comm_ispawn_multiple_sync(int count, char *array_of_commands[],
                           char* *array_of_argv[], int array_of_maxprocs[],
                           MPI_Info array_of_info[], int numroot,
                           int list_of_roots[], MPI_Comm comm, MPI_Comm *intercomm,
                           int array_of_errcodes[], MPI_Request *request);

/* use only in timeout-only scenario [for design orthogonality]: */
int MPI_Comm_spawn_local(char *command, char *argv[], int maxprocs, MPI_Info info,
                  int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *intercomm, MPI_Timeout timeout, int array_of_errcodes[]);

int MPI_Comm_spawn_multiple_local(int count, char *array_of_commands[],
```

[13]We note in the next several paragraphs that we have added certain API "for design orthogonality." This API doesn't assist with Fault Tolerance in the preferred means described in this proposal, so it can be dispensed with if orthogonality in design isn't considered essential compared to economy of the total API size of MPI. The functions are useful in the case where timeout-based errors are all the application wants to detect; they are not hard to provide given the other variants to be implemented.

```
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int numroots,
                            int list_of_roots[], MPI_Comm comm, MPI_Comm *intercomm,
                            MPI_Timeout timeout, int array_of_errcodes[]);
```

These make the best inter-communicator they can, providing you with lots of error information if they fail. This allows a number of roots to be specified (a superset of the previous functionality), all of which must offer identical input arguments if provided. In particular, the origin group is allowed to shrink, and the MPI implementation may choose to replicate input state from the given roots in order to support recovery if faults occur during spawning.

The following new API for creating intracommunicators is proposed:

```
/* no error synchronization over communicator (use in TryBlock) */
int MPI_Comm_ispawn_merge_local(char *command, char *argv[], int maxprocs,
                  MPI_Info info, int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *outcomm, MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request);

int MPI_Comm_ispawn_merge_multiple_local(int count, char *array_of_commands[],
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int numroots,
                            int list_of_roots[],MPI_Comm comm, MPI_Comm *outcomm,
                            int array_of_errcodes[], MPI_Request *request);

/* with error synchronization over communicator */
int MPI_Comm_ispawn_merge_sync(char *command, char *argv[], int maxprocs,
                  MPI_Info info, int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *outcomm, MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request);

int MPI_Comm_ispawn_merge_multiple_sync(int count, char *array_of_commands[],
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int numroots,
                            int list_of_roots[], MPI_Comm comm, MPI_Comm *outcomm,
                            int array_of_errcodes[], MPI_Request *request);

/* use only in timeout-only scenario [for design orthogonality]: */
int MPI_Comm_spawn_merge_local(char *command, char *argv[], int maxprocs,
                  MPI_Info info, int numroots, int list_of_roots[], MPI_Comm comm,
                  MPI_Comm *outcomm, MPI_Timeout timeout, int array_of_errcodes[]);

int MPI_Comm_spawn_merge_multiple_local(int count, char *array_of_commands[],
                            char* *array_of_argv[], int array_of_maxprocs[],
                            MPI_Info array_of_info[], int numroot,
                            int list_of_roots[], MPI_Comm comm, MPI_Comm *outcomm,
                            MPI_Timeout timeout, int array_of_errcodes[]);
```

With these functions, the remote process group created will not have a valid intercommunicator parent associated with it, they will be part of a bigger `MPI_COMM_WORLD` when created.

Finally, it should be noted here that an implicit requirement of Fault-Aware MPI is an execution environment that allows for the using of dynamic process management, and that many scheduler-oriented environments have forbidden the use of this functionality in practice. However, preallocation of the maximum number of replacement processes is one example of how this could be accomplished within a static framework.

7.6 Collective Communication: Inter-Communicator

Inter-communicators offer several advantages over intra-communicators in cases where there are faults (see [49] for discussion of inter-communicators). For the inter-communicators supported by MPI-2, both point-to-point and collective operations are possible. Then it depends on what `TryBlock` should be used to contain and support inter-communicator operations. A suitable inter-communicator is one whose local and remote groups are the same as or a superset of all the operations in the `TryBlock`. Extending `TryBlock`s to inter-communicators puts extra work on the implementation but not the standard.

"Active" communication on an inter-communicator is equivalent as for an intra-communicator. Point-to-point operations are attempted so long as there is no local discommendation of the remote rank involves, or a local discommendation of the communicator itself (provided we allow a local error to be raised on a communicator or our own rank). Collective communication is "active" in a process, so long as there is no fault on the communicator. Since collective operations are bilateral, it is plausible that one direction could complete but not the other. Such errors may not be adequately mapped with a single request object used for the non-blocking variant of such collectives. A collective is marked failed if any process raises an error on it, or if MPI detects an error on it.

7.7 One-Sided Communication

To be added.

7.8 I/O

To be added.

7.9 Communicator, 1-Sided, and File Management

To be added.

8 Communicators and Scalability : Less Implies More

Two concepts are considered in this section, both designed to improve the scalability of MPI applications in the face of faults:

- hierarchical communication and communicators (developed fully in [81]),
- further generalization of fault-aware process spawning and communicator construction.

For the former, we point out that the communicator as a flat virtual topology is suboptimal; and ways programs can minimally be changed to help with fault recovery even without introducing a lot of new API and implementation functionality. For the latter, we suggest additional APIs that will avoid creating communicators we really never need, thereby avoiding excess overheads in faulty environments.

8.1 Working With Hierarchical Communication to Simplify Recovery

In a reliable communication model, with static process management, the `MPI_COMM_WORLD` approach to communication initiation, and subsequent subdivisions are perfectly adequate. With the addition of MPI-2 Dynamic Process Management, programs in environments that permit resource reallocation (typically non-batch-scheduler environments), can spawn additional "worlds," and these can "join" or merge (with effort) via intercommunicator logic. IMPI, which prescribed interoperable MPI-1 implementations between vendors, worked with a mechanism to unite `MPI_COMM_WORLD`s of separate implementations, but has yet to be extended to MPI-2, and probably needs to be started from scratch after so long an hiatus.

The existence of `MPI_COMM_WORLD` in a faulty environment represents a risk to the parallel program, as well as a programmatic (and possibly performance) benefit in the absence of faults. Here is a concrete example: take the world, as given by N processes in `MPI_COMM_WORLD` and partition it into a cartesian topology of shape $P \times Q = N$. This is commonly done with linear algebra and other solvers, and is well known. In such systems, there are typically $1 + P + Q$ total communicators representing the whole group (the world), the logical process rows, and the logical process columns. Each individual process participates in three communicators, including `MPI_COMM_WORLD`. In data parallel codes, local computation is followed by row, column operations (e.g., broadcast or gather), followed sometimes by global reductions. Two-level transactions allow for independent data-parallel subgroups to continue when one faults, but requires recovery of faulty groups, and creation of a revised communicator to reintroduce global reductions. This allows the tasks in each subgroup to continue to make progress until they need a global synchronization, after which the application can either fail forward or fail backward, depending on its recovery requirements. In a fail-backward scenario, it may be possible for the intermediate work done by the good groups to be held while the faulty group is resized. In the end, $P \times Q$ processes are involved in any recovery of a single faulty process, since the global communicator must be recovered.

However, if the global reduction or gather is the only operation that the group needs, and there is no direct need for a communicator over all the groups. Instead, local groups of P (or Q) processes choose to do reductions, followed by Q (or P) processes doing orthogonal reductions. This eliminates the need for the global communicator. When a fault occurs, only the two communicators (involving $P + Q - 1$ processes) need to be fixed. What is more, a second concurrent fault at most doubles this participation. Since recovery is expected to be expensive, and additional faults can occur at any time, shortening the recovery window is highly desirable.

The role of Fault-Aware MPI is first to point out such best practices, and to allow disused communicators such as `MPI_COMM_WORLD` to remain unrecovered after faults begin. Only if the application chooses to recover the global communicator should this be required. Meanwhile, active point-to-point communication remains viable on subsets of `MPI_COMM_WORLD` that have not sustained a fault. Second, Fault-Aware MPI could propose a specific deinitialization of a communicator such as `MPI_COMM_WORLD` so that the resources associated with it can be discarded. Third, we could propose additional functionality to allow for sparse communicators to be the immediate result

of MPI_Spawn, or MPI_Init variants, so that a global communicator need never be assembled for applications that do not require it. Because synchronization at initialization and spawning is most likely superlinear in complexity, reductions to only the groups needed appears promising. A specific allowance for freeing MPI_COMM_WORLD might be useful, but programs could just decide not to use it, fault or no fault, once they set up communication... in an optimal world, individual spawned groups would be woven together with new inter-communicator logic that is easy to use. By creating communication topology initially, we avoid state, and we never have to pay for that state, or try to recover it.

Furthermore, we could consider supporting hierarchical communicators (limited communication topologies) equivalent to composing row_comm $\bigotimes$ col_comm[14]. Such a direct-product / hierarchical communicator would have constrained topological connectivity in it, but the objects could address collective communication as if they were the original global communicator. Collective operations would be unchanged, except that implementation choices would be constrained. They would inherently have two or more hops for delivering messages if employed with point-to-point operations (trading latency for recovery performance). Furthermore, such constrained communications, if defined properly, would be functionally transparent to global virtual topology communicators, and so could be passed to layered libraries, and also serve in the creation of Windows and Files. At present, this is an experimental concept, and one that can clearly be created at first glance via a layered library. Formal specification of such communicators is not offered here[15].

For now, the recommendation that Fault-Aware MPI offers is that programs such as those on cartesian topologies should be reorganized to avoid utilizing global communicators, but rather trade latency (two successive collective operations, one per logical cartesian dimension), for globality of state.

8.2 Building Just the Right Groups When Spawning

We also point out that it is possible to spawn programs more efficiently if we do not require them to join a single group, as it may be easier to build smaller groups when faults are occurring. MPI_Comm_ispawn() variants, including the fault-aware versions offered above, allow a group of processes to form a single intercommunicator per spawn (successively), and we provided optional merge forms so that the operation produces the newly desired intracommunicator. In a spawn+split API, the existing communicator (surviving) group and new tasks are color-keyed to one or more intercommunicators from the beginning (with the surviving group on the local side, and new processes on the remote side from the perspective of the invoking application). In the spawn+merge+split API, the existing communicator (surviving) group and new tasks are color-keyed into a set of independent intra-communicators. There is both an MPI_COMM_WORLD and available parent intercommunicator for new tasks. Processes from the original (surviving) group can be dispersed as chosen into the new group. In this latter model, the MPI_COMM_WORLD of the new tasks is the output

[14]Informally, you can think in a specific process in terms of T-shaped, L-shaped, or +-shaped process collections, depending on the position of logical process (p, q) in the logical grid $P \times Q$. All $P \times Q$ processes call the collective operation loosely synchronously, but they do so with their hierarchical communicator, instead of with a group-wide communicator. We actually prefer to think of these as tensor-product communicators; they deserve their own development.

[15]This type of structure could also be used with a hidden redundant dimension of tasks, in order to support extensions to Fault-Aware MPI for automated redundancy, voting, and recovery. However, that is a topic for another paper; see [81].

of the split, and there is no parent intercommunicator. Here are two examples (one from each kind of API):

```
/* these APIs have with error synchronization over communicator
   (others omitted for brevity)  */
int MPI_Comm_ispawn_split_sync(char *command, char *argv[],
                  MPI_Info info, int numroots, int list_of_roots[], MPI_Comm comm,
                  int my_color, int my_key, int numcolors, int colors[],
                  int max_tasks_per_color[], MPI_Comm *intercomm,
                  MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request); /* my_key/my_color set local comm */

int MPI_Comm_ispawn_merge_split_sync(char *command, char *argv[],
                  MPI_Info info, int numroots, int list_of_roots[], MPI_Comm comm,
                  int my_color, int my_key, int numcolors, int colors[],
                  int max_tasks_per_color[], int MPI_Comm *outcomm,
                  MPI_Timeout timeout, int array_of_errcodes[],
                  MPI_Request *request);
```

If this overall proposal is found meritorious, we will enumerate this API fully. It follows logically from the API examples here, and those above. We emphasize "build up" and "avoid bigger groups than needed" as the right approaches in future message-passing system group instantiation and management. The next subsection also considers this idea.

8.3 Joining Groups However Spawned

An ancillary function for connecting process pairs defined MPI-2, which relies on Berkeley sockets is `MPI_Comm_join()`:

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm);
```

This "odd-ball" function violates MPI's design rule about underlying transports by specifying Berkeley sockets as the means for rendezvous between two processes. It nonetheless exists, and poses an option for fault-tolerance. This function may be a good idea (its name is an anachronism as well), but not scalable enough.

Given a set of G groups (each with its own intra-communicator), we would like to create a set of H new communicators, possibly $H = 1$ (complete merger). A single MPI implementation might or might be able to help us do this, unless the tasks involved in those processes all contain elements of each group. It is particularly the case that `MPI_Comm_Join` circumvents this problem by specifying how rendezvous will be achieved between two MPI-2 processes. Such an operation would also have to have helpful fault characteristics to be part of Fault-Aware MPI.

Here are two hypothetical forms for such an operation:

```
/* timeout only (use in TryBlock): */
int MPI_Comm_isplit_multi_local(int numcomms, MPI_Comm *incomms, int num_outcomms,
                   int my_color_array[], int my_key_array[], MPI_Timeout timeout,
                   MPI_Comm *outcomms, int colltag, MPI_Request *request);
```

```
/* timeout only (sync version): */
int MPI_Comm_isplit_multi_sync(int numcomms, MPI_Comm *incomms, int num_outcomms,
                    int my_color_array[], int my_key_array[], MPI_Timeout timeout,
                    MPI_Comm *outcomms, int colltag, MPI_Request *request);
```

Note: These are not proposed, they are just examples of what could be done.

`MPI_Comm_isplit_multi_*()` would allow each task to input a number of its intracommunicators, together with a request for how many output intracommunicators to which it would like to belong (zero or more). For each of those output communicators, it would specify its desired color and key. As always, each key sorts the task's rank stably in the respective output communicator. Colors tell us how many output communicators the task will belong to on output; the also `ndest_comms` variable sets the required length of the `outcomms` array locally. The added generality is that processes could be combined that were not originally in a shared group. The MPI implementation has to be able to make a spanning tree of all represented tasks, and then build the groups as needed. The argument `colltag` – a collective tag – is provided to help keep such operations from mismatching across multiple invocations.

In the fault-aware case, the following additional capabilities would have to be present, to make this useful:

- Only point-to-point activity across the composition of groups representing all the communicators would be required
- Output groups would contain the processes not established to be faulting at the last synchronization only

Network partitioning remains a problem in all such algorithms. Reportable error conditions will be reported, such as the inability to reach certain processes. However, the network partition possibility is not unique to this function.

Note that you might need `MPI_Comm_join+MPI_Comm_merge` to create a spanning tree in some cases with independent groups, or something better we have yet to define instead of this, for cases where independent spawns were performed. So we only have part of the solution here.

The question is whether this is a good operation for MPI to offer, or is it too heavy weight. If not, is there a lighter way to connect G independent groups, so that you don't have to build them down from a bigger group that was otherwise unneeded, and subject to a higher probability of faults.

An alternative that is far more tractable is a spoke-and-hub graph model, with a central `group` connected via intercommunicators to other groups that it may have previously spawned. In that case, we simply need a better `MPI_Intercomm_merge`. This seems obvious, so we won't pursue it further at this time, it is simply an intercommunicator merge that takes several intercommunicators involving a common group instead of one intercommunicator. The fault-aware version of this proposal builds the intracommunicator that is the best it can from all of these tasks.

Allowing G disconnected groups or groups that we must assume because of possible faults to be effectively disconnected, is useful important and represents a useful building block for group communication "build up." For such a situation, we have to require the MPI implementation itself to know how to reach agreement about how to provide a common communication object. That appears the most interesting feature to support, allowing pieces and parts of the MPI application to come and go, and using the MPI implementation as a bulletin board (or tuple space) for

rendezvous/recombination of application components. It is also beyond the scope of the current proposal.

Finally, allowing smaller groups of communicating processors to form direct-product communicators described above by build up appears to be attractive. if P groups of size Q were independently created, and protocol for establishing a peer among each group were available to create a parent of size $L \geq Q$ with membership from all P groups, then we could work towards forming the Q communicators of size P representing the orthogonal dimensions without ever having a complete $P \times Q$ size communicator in any process. Rather, reach process would have two communicators representing the first and second cartesian group relationship, and a third direct-product communicator, that uses two-levels of communication. By consensus, programs would use hierarchical communicators interchangeably with communicators involving an actual group of size $P \times Q$ with possible latency and copy tradeoffs for data transfers, depending on how much the underlying MPI implementation and transport is able to mask the hierarchical nature by replicating needed state implicitly below the user-program level.

9 Further Topics

Other ideas that the author are convinced to be important, or supportive for this proposal, but that are not covered above, are mentioned here.

9.1 Operation in Multithreaded Parallel Applications

Threaded MPI programs do not pose any specific new difficult, as compared to single-threaded MPI programs. As usual, communication on a single communicator has to serialize so as to meet MPI ordering requirements.

However, it is possible to pose interesting non-blocking operations on a communicator group in an alternative thread. For instance, one could pose a user-level heart-beat or gossip engine working on a duplicated communicator of the application's communicator.

9.2 Gossip Protocols in the Progress Engine and In User Threads

It is worth noting that the propagation of errors can be done proactively, or passively, just as MPI can be proactive in communication progress, or relatively passive, waiting till wait and test to produce message progress.

Gossip protocols can be turned on during time blocks, involving the progress engines of a communicator; in this way, errors reach their destinations sooner than they would if they had to wait for MPI `TryBlock` synchronization.

9.3 End-to-End Timeouts for Transactions

`TryBlock`s do not have end-to-end timeouts associated with them. Conceivably, we could put a timeout alarm into the start of `TryBlock`, and raise an error when the time runs out. However, we are not recommending asynchronous handlers in this proposal. Instead, we offer these possibilities.

Programs written with Fault-Aware MPI are nominally nonblocking within the transactional blocks:

```
double alpha, beta, max_time = XXX;
...

TryBlock_start(..., &try_request)
alpha = MPI_Wtime();

/* MPI operations and local work here */

beta = MPI_Wtime();
if((beta - alpha) > max_time)
{
   MPI_Request_raise_error(try_request, MPI_USER_LOCAL_TIMEOUT);
}
TryBlock_waitall(...try_request...) /* has its own timeout */
```

The program would be free to check "delta T" throughout the `TryBlock` to see if the operations were progressing, particularly, if `_test_local` and `_wait_local` operations are being used (they can have local timeouts too of course, but we want to pose a total upper-bound on latency through the `TryBlock`. This approach is time polling, and depending on how long some operations take, may not provide the granularity of timeout control desired by the application.

If a maximum "delta T," measured across all the surviving tasks of the communicator is desired, then additional API is needed, so that the "allreduce with maximum" can be performed in the `_waitall`. We defer this for now.

9.4 The role and behavior of `MPI_Cancel`

The role and behavior of `MPI_Cancel` is quite simple in this proposal. Cancellation is a local request, which raises a local condition on the request involved, which may still complete, or result in a cancellation notice being posted on the status. We provide the same guarantee as MPI-3 offers for cancellation in the non-fault situation. However, two things are worth noting: `_waitall`s can raise a cancellation notification on requests globally (think of it as a soft error), and the globalization means that MPI may have the opportunity to "clean up a cancellation more efficiently" than in a nominal MPI program. Whether that is so or not, the Fault-Aware MPI programmer is free to use `MPI_Cancel` but may not see much benefit in doing so, given that there is no guarantee of active cancellation in MPI. Further discussion on this subject is clearly warranted.

9.5 "Recovering" Blocking API Operations

It is straightforward[16] to wrap the non-blocking APIs, using the new mode of testing provided. In this way, fine granuality local testing for completion can be achieved, while providing a hidden request that is managed strictly by a layered library approach to the transactional commit. This is sketched as follows:

[16]But why would you do this? It might help transform existing blocking MPI-1 and MPI-2 programs to use Fault-Aware MPI, and it might help some program transformation engines. But, the best programs will not insist on blocking each operation, but allow operations to progress concurrently, and for MPI to do as much as it can in parallel with the application tasks/threads/processes.

```
int MPI_Send_In_TryBlock(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Timeout timeout, MPI_Request tryrequest)
{
    int error;

    MPI_Request *request = malloc(MPI_Request *)sizeof(MPI_Request);
    /* ADD: Keep a copy of the request as an info item or linked list associated
       with tryrequest, or as a request tied to comm used with transrequest ;
       we can refine this;  bottom line, we will extract those, and then use at
       the TryBlock_wait(); This is where the tryrequest comes in...
       no new MPI API needed;

       NB. DESCRIBE WHAT TO DO HERE WITH NESTED TRANSACTIONS; WHERE TO
       ATTACH REQUEST?
    */

    error = MPI_Isend(buf, count, datatype, dest, tag, comm, request);
    if(error == MPI_SUCCESS)
    {
       error = MPI_Wait_local(timeout, request, MPI_STATUS_IGNORE);
    }
    else
    {
       /* even though an error happened, the request was made,
          and will be part of transactional commit */
    }
    return(error);
}
```

The point is to demonstrate that we can move local completion back to the level of a single operation. Only local errors are required to be detected here, and this operation may succeed, only to fail at commit. For the fault-free case, this operation might have slightly higher overhead, but is otherwise semantically equivalent. Even though a user can do this, he or she might not wish to do so, because this means that there is a completion with the operation in the `TryBlock`, rather than an aggregation of completion over several operations, which ideally will be more efficient... the point is, the user can provide a blocking form of MPI if desired. What is more, we would standardize these operations if desired; the authors simply do not recommend this as required.

This Operation can be layered on existing functionality of MPI-3 plus the proposed functionality here. Therefore, users can create blocking operations if desired, without requiring the MPI Standard to expand multiplicatively to augment all blocking API. That means that this proposal is no less general than those that directly support the blocking APIs in Fault-Tolerant-mode.

10 Future Work

The introduction of transactional semantics and supporting syntax into MPI, together with identification of the non-blocking APIs of MPI as the most easy to integrate into a Fault-Aware framework

opens the door for further potential progress. In particular, the use of transactional concepts in MPI I/O subsystems and distributed shared memory subsystems offers to support better recovery than is currently envisioned. We intend to explore transactional distributed shared memory, and transactional parallel file systems used to backend MPI I/O as ways to enhance recovery.

We've introduced a number of possible directions to enhance MPI, always with attention to fault-awareness, but in other areas directly in support of scalability. For example, hierarchical (aka tensor-product or direct product) communicators described above are being developed further separately [81].

In short, the Fault-Aware MPI API described here poses/reveals middle-out requirements both on the user application, and on the underlying services that an MPI implementation will use, including parallel I/O and filesystems (*cf*, [82]). In order to achieve effective detection, isolation, mitigation, and recovery, we observe the benefits of transactional concepts throughout the parallel system, not just in the way the MPI application is programmed itself for resilience.

11 Summary/Conclusions

In this proposal, we have offered a Fault-Aware MPI specification, including examples. This model offers MPI-3 applications and implementations with a realizable implementation of resilient programs, and can provide mitigation and recovery, together with application-specific or general algorithmic fault tolerance, and/or checkpoint restart, and/or other redundancy and recovery schemes and concepts to achieve practical program execution in faulty parallel environments. New features elsewhere in MPI-3 make this model feasible. The main idea is to use transactional blocks, and to support rollback of some or all operations at a transactional commit. Applications are given broad leeway on how to use the simple mechanisms provided, but these mechanisms support data parallel, task parallel, and multi-level parallel (layered) programs. Communication (1-sided, 2-sided) as well as I/O operations of MPI are included in this proposal, but we restrict support to non-blocking communicating operations. Relatively few new APIs are required, and the only new data type introduced is a timeout. We also show how to reintroduce blocking functionality through a layered library approach.

Fault-Aware MPI programs will be factorable into nominal operation, and fault mode operation, so that aspect-oriented approaches based on source-to-source translators could be used to separate concerns for application developers and fault-tolerance developers in large-scale software. This supports our long-held view that a single MPI program must be usable in multiple fault environments, so that the accidental complexity of managing the faults must be separable from the main MPI code. Otherwise, important aspects of portability are lost, or strongly diminished.

It is important to conclude with a reality check. There is no "free lunch": some faults (*e.g.*, a power outage) are going to take out the entire HPC system or partition on which an application is running, rendering moot the efforts to make a Fault-Aware parallel application fail through an unmodeled fault while still running via hierarchical recovery such as CPR[17]. There are "situations" where we might hope for the ability to recover using Fault-Aware MPI and otherwise, but are not necessarily entitled to do so.

[17]The recovery for this particular case is restart in the job queue after system reboot, which is just another level of hierarchical recovery.

12 Acknowledgements

The authors acknowledge Dr. Joshua Hursey of Oak Ridge National Laboratory, for motivating us to write up this proposal.

References

[1] *2004 IEEE International Conference on Cluster Computing (CLUSTER 2004), September 20-23 2004, San Diego, California, USA*. IEEE Computer Society, 2004.

[2] *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*. IEEE Computer Society, 2005.

[3] *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*. IEEE, 2007.

[4] *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*. IEEE, 2011.

[5] ACM. *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. ACM Press, 2006.

[6] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *HPDC*, 1999.

[7] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[8] Thara Angskun, Graham E. Fagg, George Bosilca, Jelena Pjesivac-Grbovic, and Jack Dongarra. Scalable fault tolerant protocol for parallel runtime environments. In Mohr et al. [72], pages 141–149.

[9] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, and Mark A. Taylor. Network fault tolerance in la-MPI. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 344–351. Springer, 2003.

[10] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *IPDPS*. IEEE Computer Society, 2004.

[11] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[12] Rajanikanth Batchu, Anthony Skjellum, Zhenqian Cui, Murali Beddhu, Jothi P. Neelamegam, Yoginder S. Dandass, and Manoj Apte. MPI/FT(tm): Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *CCGRID*, pages 26–33. IEEE Computer Society, 2001.

[13] Rob H. Bisseling. *Parallel scientific computation - a structured approach using BSP and MPI.* Oxford University Press, 2004.

[14] Douglas M. Blough and Peng Liu. Fimd-MPI: A tool for injecting faults into MPI applications. In *IPDPS*, pages 241–. IEEE Computer Society, 2000.

[15] George Bosilca, January 2012. Personal Communication.

[16] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *SC*, pages 1–18, 2002.

[17] Aurelien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC*, page 25. ACM, 2003.

[18] Aurelien Bouteiller, Boris Collin, Thomas Hérault, Pierre Lemarinier, and Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant MPI. In *IPDPS* [2].

[19] Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V project: A multiprotocol automatic fault-tolerant mpi. *IJHPCA*, 20(3):319–333, 2006.

[20] Aurelien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *CLUSTER*, pages 242–250. IEEE Computer Society, 2003.

[21] Greg Bronevetsky. Transactional messages. URL: http://svn.mpi-forum.org/trac2/mpi-forum-web/wiki/Transactional-Msg.

[22] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in application-level fault-tolerant MPI. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *ICS*, pages 234–243. ACM, 2003.

[23] Darius Buntinas. Scalable distributed consensus to support MPI fault tolerance. In Cotronis et al. [30], pages 325–328.

[24] Darius Buntinas, Camille Coti, Thomas Hérault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Comp. Syst.*, 24(1):73–84, 2008.

[25] Alejandro Calderón, Félix García Carballeira, Jesús Carretero, José María Pérez, and Luis Miguel Sánchez. A fault tolerant MPI-IO implementation using the expand parallel file system. In *PDP*, pages 274–281. IEEE Computer Society, 2005.

[26] Alejandro Calderón, Félix García Carballeira, Florin Isaila, Rainer Keller, and Alexander Schulz. Fault tolerant file models for MPI-IO parallel file systems. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 153–160. Springer, 2007.

[27] Franck Cappello. Fault tolerance for petascale systems: Current knowledge, challenges and opportunities. In Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, page 2. Springer, 2008.

[28] Sayantan Chakravorty, Celso L. Mendes, and Laxmikant V. Kalé. Proactive fault tolerance in MPI applications via task migration. In Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *HiPC*, volume 4297 of *Lecture Notes in Computer Science*, pages 485–496. Springer, 2006.

[29] Camille Coti, Thomas Hérault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. MPI tools and performance studies - blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC* [5], page 127.

[30] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011.

[31] Charng da Lu and Daniel A. Reed. Assessing fault sensitivity in MPI applications. In *SC*, page 37. IEEE Computer Society, 2004.

[32] David Dewolfs, Jan Broeckhove, Vaidy S. Sunderam, and Graham E. Fagg. FT-MPI, fault-tolerant metacomputing and generic name services: A case study. In Mohr et al. [72], pages 133–140.

[33] David Dewolfs, Dawid Kurzyniec, Vaidy S. Sunderam, Jan Broeckhove, Tom Dhaene, and Graham E. Fagg. Applicability of generic naming services and fault-tolerant metacomputing with FT-MPI. In Martino et al. [70], pages 268–275.

[34] Jack Dongarra. Fault tolerance in message passing and in action. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, page 6. Springer, 2004.

[35] Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*, volume 1908 of *Lecture Notes in Computer Science*. Springer, 2000.

[36] Angelo Duarte, Dolores Rexachs, and Emilio Luque. An intelligent management of fault tolerance in cluster using RADICMPI. In Mohr et al. [72], pages 150–157.

[37] Graham E. Fagg, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack Dongarra. Scalable fault tolerant MPI: Extending the recovery algorithm. In Martino et al. [70], pages 67–75.

[38] Graham E. Fagg, Antonin Bukovsky, and Jack Dongarra. Fault tolerant MPI for the harness meta-computing system. In Vassil N. Alexandrov, Jack Dongarra, Benjoe A. Juliano, René S. Renner, and Chih Jeng Kenneth Tan, editors, *International Conference on Computational Science (1)*, volume 2073 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2001.

[39] Graham E. Fagg, Antonin Bukovsky, and Jack Dongarra. Harness and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[40] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In Dongarra et al. [35], pages 346–353.

[41] Graham E. Fagg and Jack Dongarra. Harness fault tolerant MPI design, usage and performance issues. *Future Generation Comp. Syst.*, 18(8):1127–1142, 2002.

[42] Graham E. Fagg and Jack Dongarra. Building and using a fault-tolerant mpi implementation. *IJHPCA*, 18(3):353–361, 2004.

[43] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault-management in P2P-MPI. *International Journal of Parallel Programming*, 37(5):433–461, 2009.

[44] Stéphane Genaud and Choopan Rattanapoka. Fault management in P2P-MPI. In Christophe Cérin and Kuan-Ching Li, editors, *GPC*, volume 4459 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2007.

[45] Stéphane Genaud and Choopan Rattanapoka. Evaluation of replication and fault detection in P2P-MPI. In *IPDPS*, pages 1–8. IEEE, 2009.

[46] Alexandre D. Gonçalves, Matheus Bersot, André Bulcão, Cristina Boeres, Lúcia Maria de A. Drummond, and Vinod E. F. Rebello. Fault tolerance in an industrial seismic processing application for multicore clusters. In Cotronis et al. [30], pages 264–271.

[47] Richard L. Graham. Approaches for parallel applications fault tolerance. In Mohr et al. [72], page 2.

[48] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. URL: www.mcs.anl.gov/ lusk/papers/fault-tolerance.pdf. Accessed: January 21, 2012.

[49] William Gropp and Ewing Lusk. Fault tolerance in Message Passing Interface programs. *Int. J. High Perform. Comput. Appl.*, 18:363–372, August 2004.

[50] Hyuck Han, Hyungsoo Jung, Jai Wug Kim, Jongpil Lee, Youngjin Yu, Shin Gyu Kim, and Heon Young Yeom. Shield: A fault-tolerant MPI for an infiniband cluster. In Michael Gerndt and Dieter Kranzlmüller, editors, *HPCC*, volume 4208 of *Lecture Notes in Computer Science*, pages 874–883. Springer, 2006.

[51] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[52] William Hoarau, Pierre Lemarinier, Thomas Hérault, Eric Rodriguez, Sébastien Tixeuil, and Franck Cappello. FAIL-MPI: How fault-tolerant is fault-tolerant MPI? In *CLUSTER*. IEEE, 2006.

[53] Benoit Hudzia and Serge G. Petiton. Reliable multicast fault tolerant MPI in the grid environment. *CoRR*, abs/cs/0608114, 2006.

[54] Joshua Hursey and Richard L. Graham. Building a fault tolerant MPI application: A ring communication example. In *IPDPS Workshops* [4], pages 1549–1556.

[55] Joshua Hursey and Richard L. Graham. Preserving collective performance across process failure for a fault tolerant MPI. In *IPDPS Workshops* [4], pages 1208–1215.

[56] Joshua Hursey and Richard L. Graham. Analyzing fault aware collective performance in a process fault tolerant mpi. *Parallel Computing*, 38(1-2):15–25, 2012.

[57] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In Cotronis et al. [30], pages 329–332.

[58] Joshua Hursey, Thomas Naughton, Geoffroy Vallée, and Richard L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant mpi. In Cotronis et al. [30], pages 255–263.

[59] Joshua Hursey, Jeffrey M. Squyres, Timothy Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open MPI. In *IPDPS* [3], pages 1–8.

[60] Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. Abaris: An adaptable fault detection/recovery component framework for MPIs. In *IPDPS* [3], pages 1–8.

[61] Hyungsoo Jung, Dongin Shin, Hyuck Han, Jai Wug Kim, Heon Young Yeom, and Jongsuk Lee. Design and implementation of multiple fault-tolerant MPI over myrinet (m^3). In *SC*, page 32. IEEE Computer Society, 2005.

[62] Dawid Kurzyniec and Vaidy S. Sunderam. Combining FT-MPI with H2O: Fault-tolerant MPI across administrative boundaries. In *IPDPS* [2].

[63] Inseon Lee, Heon Young Yeom, Taesoon Park, and Hyoung-Woo Park. A lightweight message logging scheme for fault tolerant MPI. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *PPAM*, volume 3019 of *Lecture Notes in Computer Science*, pages 397–404. Springer, 2003.

[64] Pierre Lemarinier, Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, and Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER* [1], pages 115–124.

[65] Pierre Lemarinier, Aurelien Bouteiller, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. *IJHPCN*, 2(2/3/4):146–155, 2004.

[66] Xu Liu, Bibo Tu, Jianfeng Zhan, and Dan Meng. A fast-start, fault-tolerant MPI launcher on dawning supercomputers. In *PDCAT*, pages 263–266. IEEE Computer Society, 2008.

[67] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, and Paraskevas Evripidou. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.

[68] André Luckow and Bettina Schnor. Migol: A fault-tolerant service framework for MPI applications in the grid. In Martino et al. [70], pages 258–267.

[69] André Luckow and Bettina Schnor. Migol: A fault-tolerant service framework for MPI applications in the grid. *Future Generation Comp. Syst.*, 24(2):142–152, 2008.

[70] Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*. Springer, 2005.

[71] Mauro Migliardi, Vaidy S. Sunderam, and Arrigo Frisiani. A simple, fault tolerant naming space for the harness metacomputing system. In Dongarra et al. [35], pages 152–159.

[72] Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*. Springer, 2006.

[73] José Carlos Mouriño, María J. Martín, Patricia González, and Ramon Doallo. Fault-tolerant solutions for a MPI compute intensive application. In *PDP*, pages 246–253. IEEE Computer Society, 2007.

[74] M. Vivekananda Reddy and Sanjay Chaudhary. Scheduling in grid: Rescheduling MPI applications using a fault-tolerant MPI implementation. In *COMSWARE*. IEEE, 2007.

[75] Gabriel Rodríguez, Patricia González, María J. Martín, and Juan Touriño. Enhancing fault-tolerance of large-scale MPI scientific applications. In Victor E. Malyshkin, editor, *PaCT*, volume 4671 of *Lecture Notes in Computer Science*, pages 153–161. Springer, 2007.

[76] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant V. Kalé, and Franck Cappello. On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 567–578. Springer, 2011.

[77] Anton Selikhov and C. Germai. A channel memory based fault tolerance for MPI applications. *Future Generation Comp. Syst.*, 21(4):709–715, 2005.

[78] Anton Selikhov and Cécile Germain. CMDE: A channel memory based dynamic environment for fault-tolerant message passing based on MPICH-V architecture. In Victor E. Malyshkin, editor, *PaCT*, volume 2763 of *Lecture Notes in Computer Science*, pages 528–537. Springer, 2003.

[79] A. David Selvakumar, P. M. Sobha, G. C. Ravindra, and R. Pitchiah. Design, implementation and performance of fault-tolerant message passing interface (MPI). In David A. Bader and Ashfaq A. Khokhar, editors, *ISCA PDCS*, pages 145–150. ISCA, 2004.

[80] Galen M. Shipman, Richard L. Graham, and George Bosilca. Network fault tolerance in open MPI. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 868–878. Springer, 2007.

[81] Anthony Skjellum. Hierarchical Communicators and Groups for MPI-3: Fault-Awareness, Dynamic Process Management, Transparent-Redundancy, and Exascale. Technical Report UABCIS-TR-2012-020612, University of Alabama at Birmingham, Computer and Information Sciences, February 2012. URL: TBD.

[82] Anthony Skjellum. Middle-out transactional requirements on exascale parallel middleware, storage, and services. Technical Report UABCIS-TR-2012-020312, University of Alabama at Birmingham, Computer and Information Sciences, February 2012. URL: TBD.

[83] Rajagopal Subramaniyan, Vikas Aggarwal, Adam Jacobs, and Alan D. George. FEMPI: A lightweight fault-tolerant MPI for embedded cluster systems. In Hamid R. Arabnia, editor, *ESA*, pages 3–9. CSREA Press, 2006.

[84] Jyothish Varma, Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Scalable, fault tolerant membership for MPI tasks on hpc systems. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 219–228. ACM, 2006.

[85] Abhinav Vishnu, Prachi Gupta, Amith R. Mamidala, and Dhabaleswar K. Panda. Scalable systems software - a software based approach for providing network fault tolerance in clusters with udapl interface: MPI level design and performance evaluation. In *SC* [5], page 85.

[86] John Paul Walters and Vipin Chaudhary. A scalable asynchronous replication-based strategy for fault tolerant MPI applications. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2007.

[87] John Paul Walters and Vipin Chaudhary. Replication-based fault tolerance for MPI applications. *IEEE Trans. Parallel Distrib. Syst.*, 20(7):997–1010, 2009.

[88] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *IPDPS* [3], pages 1–10.

[89] Zhiyuan Wang, Xuejun Yang, and Yun Zhou. MMPI: A scalable fault tolerance mechanism for MPI large scale parallel computing. In *CIT*, pages 1251–1256. IEEE Computer Society, 2010.

[90] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *CLUSTER* [1], pages 93–103.