# A high-level C++ approach to manage local errors, asynchrony and faults in an MPI application

1st Christian Engwer
*Applied Mathematics*
*University of Münster*
*Orleansring 10*
*48149 Münster, Germany*

2nd Mirco Altenbernd
*IANS*
*University of Stuttgart*
*Allmandring 5b*
*70569 Stuttgart, Germany*

3rd Nils-Arne Dreier
*Applied Mathematics*
*University of Münster*
*Orleansring 10*
*48149 Münster, Germany*

4th Dominik Göddeke
*IANS*
*University of Stuttgart*
*Allmandring 5b*
*70569 Stuttgart, Germany*

*Abstract*—C++ advocates exceptions as the preferred way to handle unexpected behaviour of an implementation in the code. This does not integrate well with the error handling of MPI, which more or less always results in program termination in case of MPI failures. In particular, a local C++ exception can currently lead to a deadlock due to unfinished communication requests on remote hosts. At the same time, future MPI implementations are expected to include an API to continue computations even after a hard fault (node loss), i.e. the worst possible unexpected behaviour.

In this paper we present an approach that adds extended exception propagation support to C++ MPI programs. Our technique allows to propagate local exceptions to remote hosts to avoid deadlocks, and to map MPI failures on remote hosts to local exceptions. A use case of particular interest are asynchronous 'local failure local recovery' resilience approaches. Our prototype implementation uses MPI-3.0 features only. In addition we present a dedicated implementation, which integrates seamlessly with MPI-ULFM, i.e. the most prominent proposal for extending MPI towards fault tolerance.

Our implementation is available at https://gitlab.dune-project.org/christi/test-mpi-exceptions.

*Keywords*-C++, ULFM, Exceptions, Fault-tolerance

## I. INTRODUCTION

C++ programs comprise a wide range of target machines with many different architectures. Despite careful debugging, it is always possible that a program behaves unexpectedly in some way. This is particularly important in the case of software frameworks, which we are mostly interested in: Packages like DUNE [1], deal.II [2] and Trilinos [3] have a broad user base for PDE-related computations, and all use C++ template metaprogramming to relieve the user of the burden to implement common features over and over again. Furthermore parallel software frameworks often hide at least coarse-grained parallelism from their users. In parallel numerical algorithms, unexpected behaviour can occur quite frequently: A solver could diverge, the input of a component (e.g. the mesher) could be inappropriate for another component (e.g. the discretiser), etc.

A well-written code should detect unexpected behaviour and provide the user with a possibility to react appropriately in their own code, instead of simply terminating with some error code. For C++, *exceptions* are the recommended method to handle this. With well placed exceptions and corresponding *try-catch* blocks, it is possible to accomplish a more robust program behaviour. This holds both for framework developers and framework users.

For large-scale computations, MPI ('message passing interface') is the de-facto standard for coarse-grained communication. The current MPI specification [4] does not define any way to propagate exceptions from one so-called rank (process) to another. In the case of unexpected behaviour within the MPI layer itself, MPI programs simply terminate, maybe after a time-out. This is a design decision that unfortunately implies a severe disadvantage in C++, when combined with the ideally asynchronous progress of computation and communication: An exception that is thrown locally by some rank can currently lead to a communication deadlock, or ultimately even to undesired program termination. Even though exceptions are technically an illegal use of the MPI standard (a peer no longer participates in a communication), it undesirably conflicts with the C++ concept of error handling.

With increasing degrees of parallelism and architectural complexity, it becomes even harder for framework developers to predict eventual misbehaviour and to provide appropriate infrastructure. When not resorting to C++ exceptions or synchronous global checkpoint-restart techniques, it is possible (albeit cumbersome and undesirable) with basic MPI features that a local process communicates its information to other participants to restore the functionality of an algorithm. This does not hold however for failures within the MPI layer itself: The reliability of hardware is expected to become a non-negligible problem on upcoming extreme-scale systems [5], [6], and predictions of the Mean-Time-Between-Failure (i.e. the expected time span between

two failures) hint at this problem to become the norm rather than the exception. This makes it necessary to include support for failure propagation and thus fault-mitigation and fault-tolerance, for the full range from locally recoverable 'irregularities', the unrecoverable failure of complete ranks and their associated data, and anything in between.

*Contribution:* We are convinced that any kind of unexpected behaviour in MPI-C++ programs should be treated and is treatable in the same way, i.e. through C++ exceptions. In this paper, we present a possible approach along with a prototype implementation to realise this claim. We follow C++11 techniques, e.g. use future-like abstractions to handle asynchronous communication.

*State of the art:* Current implementations of the MPI standard like MPICH, OpenMPI or IntelMPI do not provide an easy-to-use way to propagate unexpected behaviour from one process/rank to another: The standard does not mandate this, not even for failures like a node loss within the MPI layer itself. Current MPI implementations thus typically terminate (or deadlock) in such a situation. The most prominent proposal which suggests a suitable extension to the MPI standard currently is *User-Level Failure Mitigation (ULFM)* [7], [8]. It allows users to define a workaround for the node loss scenario, e.g. clear the broken communicator and create a new one with a reduced number of processors, or include some spare nodes. This extension will provide a good solution for most of the arising problems, but it is still far away from being available on current HPC systems: ULFM might be included in the standard only from MPI-4 onwards. The approach we suggest works without ULFM, but includes a dedicated code path for any future MPI version that includes ULFM. We describe our ULFM integration in section III-C.

Furthermore the C++ API for MPI was dropped from MPI-3 since it offered no real advantage over the C bindings, instead of being a simple wrapper layer. MPI users coding in C++ are still using the C bindings, writing there own C++ interface/layer or using existing interfaces like Boost.MPI [9]. Although the Boost.MPI documentation states that the library '[...] provides an alternative C++ interface to MPI that better supports modern C++ development styles, including complete support for user-defined data types and C++ Standard Library types and arbitrary function objects for collective algorithms...', there is currently no support for neither exception propagation nor fault-tolerance.

*Use cases for our technique:* Due to the current discrepancy of supported software on different machines, which most of the time do not have MPI implementations with ULFM support, one goal of this work is to provide a flexible C++ interface for unified exception propagation. The infrastructure we provide allows to manage all kinds of local misbehaviour and most faults at the MPI level, in a C++ conforming way. Our approach is future-proof in the sense that it should easily enable switching to an ULFM-enhanced MPI version without the necessity to substantially change the user code, once such an MPI library becomes available. Switching to such an MPI deployment furthermore extends the type of faults which can be handled.

Our currently implemented prototype interface handles all faults which we describe in section II-A if the MPI installation supports ULFM. Otherwise only soft faults and thus exception propagation are supported.

We particularly focus on the quite general problem of propagating *a local error to other MPI ranks*. This general problem is relevant in different scenarios and fault-tolerance concepts:

1) *Local failure local recovery* (LFLR) [10] techniques are based on recomputation of lost information, rather than resorting to a global checkpoint. For instance, Huber et al. [11] or Göddeke et al. [12] pursue this idea for multigrid. If a node crashes, the lost approximation is recomputed, which mandates to re-establish the (lost) communicator.

2) In some scenarios it can be possible that failures are repairable in a sufficient way for the local computation but nevertheless would lead to bad behaviour on a more global level. This necessitates a *local repair and (semi-) global action*. In this case one would need some hierarchical escalation strategy to propagate the error to the neighbourhood in an efficient way, without impacting unaffected processes too much. A prominent example are Krylov-type solvers, where a small local inconsistency can lead to a globally skewed Krylov space and thus to deteriorated convergence rates or even convergence to a wrong global solution. This is in some sense comparable to the LFLR concept: A local repair and a global reset of the solver is sufficient to maintain a good convergence behaviour without a global rollback. This concept can drastically reduce the amount of necessary communication and possibly provides more efficient recovery for highly parallel systems.

3) In the worst case we need a *global roll-back* within the whole communicator. This means that we have to send a signal from possibly one rank to all ranks in the communicator, stop the current operation and go back to the state of the last checkpoint.

## II. Background

On the way to exascale computing many new challenges are arising, and it is still unknown what problems we exactly have to expect. Consensus exists that the number of processes will increase by a factor of 10 to 100. At the same time it is anticipated that the Mean-Time-Between-Failure (MTBF) is decreasing. This is a direct effect of the increased number of processes, since studies predict that the MTBF is proportional to the number of processes [13],

[14]. In addition systems reliability might become increasingly important because one approach to increase energy efficiency is to lower the core-voltage, which in turn leads to an increasing probability of soft faults (i.e. bit-flips) [15].

### A. Faults and Failures

Elliot et al. [16] introduce a widely used terminology and taxonomy to differentiate between fault types. These range from occasional or recurrent bit-flips which can have no effect or may lead to permanent faulty computation, up to losing complete nodes, and anything in between. For this work we categorise them roughly into two types: *Hard failures* lead to the crash of a process, and *soft failures* possibly lead to unexpected behaviour but without interrupting computation or communication. Nevertheless the obtained results after a soft failure may be faulty. In the following we shortly describe the characteristics of these failures. We do not differentiate between *failure* and *fault*.

*Soft failure:* We categorise a failure as a soft failure if afterwards the process is still capable of throwing an exception in some way. They can occur either directly due to a C++ runtime error, a numerical failure within an algorithm (like division by zero) or more generally a detected misbehaviour (e.g. solver divergence) and a related user-defined exception. In addition, the process must be able to communicate this exception afterwards. We will not further categorise or differentiate this type of failure. Neither we will talk about detection mechanism or possibilities to repair the effect of such failures. This is a user-level task and thus coherent with the idea behind the proposed ULFM extension. We are interested in providing additional functionality for the user to handle such circumstances in a problem-specific and thus more efficient fashion, rather than a black-box solution like global checkpointing.

*Hard failure:* A hard failure on the other hand leads to the loss of a part of a communicator, i.e. a process or a whole node. Within an MPI communication this can result in a deadlock due to open MPI requests. These failures are a main motivation behind the design of the ULFM extension [7]. If a hard failure occurs it is not straight forward to continue the computation. The default way to handle such faults is a rollback to a previous checkpoint, which will be more and more expensive with increasing parallelism not only because of recomputation but also because of communication [13], [17]–[19]. In addition the communicator has to be re-established with replacement processes, or the application has to be repartitioned and/or load-balanced.

### B. ULFM

*User Level Failure Mitigation* (ULFM) is proposed to be 'a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures'[1]. If a hard failure occurs in current versions of OpenMPI and MPICH, the runtime tries to terminate all processes and ends the computation. The idea of the ULFM proposal is instead to return an error code to the user, which enables to define an approach to repair the computation, e.g. by freeing all other processes and the faulty communicator, followed by setting up a new communicator. Alternatively it is an option to shrink the communicator so that computation can be continued with less participants. We emphasise again that the actual reaction is problem-specific. To this end, ULFM proposes a set of new essential MPI functions, for instance:

- `MPI_Comm_revoke`
  This function signals the revocation of the communicator to all ranks; further MPI calls (except of the two following) within the communicator will fail with an error code of class `MPI_ERR_COMM_REVOKED`.
- `MPI_Comm_agree`
  This function provides functionality for agreeing on further proceeding after a failure between ranks within a communicator. A bit-wise `AND` operation is performed over an integer.
- `MPI_Comm_shrink`
  A new communicator is created excluding all failed ranks.
- Hard-failure detection
  Hard-failures are detected by ULFM. In that case all communication involving failed ranks is terminated with an error of class `MPI_ERR_PROC_FAILURE` or `MPI_ERR_PROC_FAILURE_PENDING`.

Several other features especially for file I/O and one-sided communication are provided by ULFM, and we refer to the current specification for details[2].

ULFM is not included in the MPI-3 standard but is proposed for MPI 4.0[3]. A prototype implementation in OpenMPI exists, but is based on the outdated version 1.7.x of OpenMPI. This makes it hard to deploy it on current HPC systems and to make it available for a large user-base before it is finally integrated into the standard. A dedicated version of a ULFM-extended OpenMPI implementation is available on Edison, a large-scale Cray XC30 machine [4]. This indicates that it could become available in more production environments in the near future.

Some first implementations within MPICH [20] exist, but they are unoptimised yet and more or less only a proof of concept. In particular, it is not possible to shrink a revoked communicator[5], which is required in our ULFM-based implementation. Therefore we cannot present results

---

[1]http://fault-tolerance.org/downloads/20161115-tutorialSC16-handson.pdf, slide 7

[2]http://fault-tolerance.org/ulfm/ulfm-specification

[3]http://mpi-forum.org/mpi-40

[4]http://fault-tolerance.org/2017/03/26/running-on-edison

[5]https://github.com/pmodels/mpich/issues/2198

with MPICH version 3.3a2.

## III. INTERFACE AND TECHNICAL DETAILS

This section describes the user interface of our proposal. It implements the future paradigm, which was introduced to C++ with C++11 to handle asynchronous tasks. Furthermore it uses exceptions to indicate errors like it is advocated by the C++ standard. Early experiments (not covered in this paper) indicate that our approach can already be used to define algorithm-specific LFLR techniques to increase the fault-tolerance of algorithms.

For the implementation of the user interface we distinguish two cases: with and without ULFM support of the underlying MPI implementation. We refer the non-ULFM implementation as the *Black-Channel* approach since we create an additional communicator for error communication, which is not used in the fault-free scenario. In this case it is only possible to detect soft faults. If the ULFM extension is available, the interface will adapt and can detect hard faults as well.

The code is available at https://gitlab.dune-project.org/christi/test-mpi-exceptions and contains also a range of small examples with exception propagation.

### A. User interface

Figure 1 shows a class diagram of the user interface, and we describe the semantic of each class in the following paragraphs. Listing 1 shows an example of using the user interface.

*Instance:* Every MPI program needs to call `MPI_Init` at the beginning of the program and `MPI_Finalize` at the end. We implement this into a singleton class `Instance` to ensure the proper structure of the program. The constructor checks if MPI is already initialised, if it is not the case `MPI_Init` is called. `MPI_Finalize` is only called in the destructor if `MPI_Init` was called in the constructor of the respective object. The `Instance` class also provides access to the `comm_world` communicator.

*Comm:* The class `Comm` manages a communicator and provides functions to duplicate the communicator, to generate new sub-communicators and to issue communication calls like `send` and `recv`. Collective communication is also feasible but might lead to a memory leak in the failure case (cf. section IV-B). We exemplarily implemented the `all_reduce` functionality but the class is easily extendable to every non-blocking communication method. Furthermore `rank` number and `size` of the communicator can be determined by calling the respective methods. Instances of this class cannot be copied, since this class represents a one-to-one relation to MPI communicators. For duplication the interface provides a dedicated `duplicate` method. Intracommunicators are not supported yet.

*Future:* Communication requests are wrapped in the class `Future` which implements the asynchronous programming concept of using *future-type objects*. An non-blocking communication can be initiated by calling the respective methods of the `Comm` object, which returns a `Future` object. The user calls the method `wait` to ensure that the communication is completed (i.e. the buffer of the communication can be reused). The `wait` method may throw one of the following exceptions:

*Propagated:* One rank can signal an error to all remote ranks by calling the method `signal_error`, which takes an error code as an argument. In this case all remote ranks throw an exception of type `Propagated_exception`, when they call `wait` of a `Future` object or if they are already waiting. The rank itself throws a `Propagated_exception` within the method `signal_error`. The `Propagated_exception` objects contains information about which ranks (possibly several) have signaled an error and with which error code. Reacting to these exceptions does not require to revoke and set up a new communicator.

*Corrupted communicator:* The `Comm` object detects in the destructor whether it gets destructed during stack unwinding due to a thrown exception by using `std::uncaught_exception`. This incident is interpreted as an unrecoverable error within the communicator. It is propagated to all remote ranks, which will throw a `Corrupted_comm_exception` when they call `wait` or are already waiting on a `Future` object. This exceptions should not be caught within the scope of the `Comm` object to ensure a consistent state on all ranks.

*MPI errors:* In the case of any MPI error that cannot be assigned to one of the previous exceptions we throw an `MPI_error_exception`. It inherits from `std::exception` and contains the respective error code.

### B. Black channel implementation

We now present the implementation based on the MPI-3.0 standard (without ULFM). The constructor of the `Comm` object duplicates the MPI communicator by calling `MPI_Comm_dup`. The new communicator is called `comm_err` and is stored in the `Comm` object. It is used for failure related communication. In `comm_err` we create a non-blocking receive operation via `MPI_Irecv` and store the pending request in `err_req`. The duplication of the communicator is made to not block a communication tag.

The function `signal_error` issues a matching `MPI_Issend` for `err_req` to all other ranks and cancels its own `err_req`. It uses the non-blocking operation since it is possible that two ranks simultaneously propagate errors: In that case a blocking operation may deadlock since it is not ensured that there is a matching `recv`. Once all error messages have been send or a rank receives an error message, it calls `MPI_Barrier` to wait for all ranks being in the
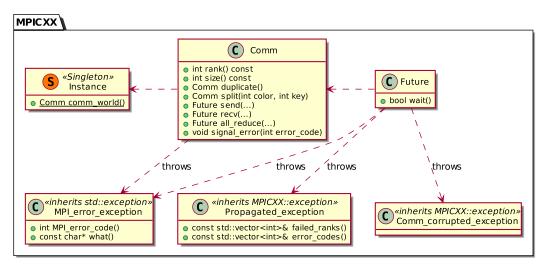
Figure 1: Class diagram of the user interface

Listing 1: Minimal user interface (2 processors)

```
// initializing mpi via a singleton class
const MPICXX::Instance& mpi = MPICXX::initialize(argc, argv);
try { // try-catch: corrupted communicator exceptions
    MPICXX::Comm& comm = mpi.comm_world(); // get a reference to the mpi communicator
    try { // try-catch: remote/propagated exceptions
        try { // try-catch: local exceptions
            int answer = 0;
            MPICXX::Future f; // creating a future object to store the communication
            if (comm.rank() == 0) { // rank 0
                answer = 42;
                // set up a send from rank 0 to rank 1
                f = comm.send(&answer, 1, MPI_INT, 1);
            }
            if (comm.rank() == 1) { // rank 1
                // set up a receive on rank 1 from rank 0
                f = comm.recv(&answer, 1, MPI_INT, 0);
            }
            f.wait(); // wait for communication to be finished
        } catch(std::exception& e) { // catch local exception
            comm.signal_error(666); // propagate an exception with code 666 to all ranks
        }
    } catch(MPICXX::Propagated_exception &e) { // catch a remote/propagated exception
        // initiate a recovery process: e.g. if a skewed Krylov basis is detected locally
        // a global restart with the current approximation can be initiated
        // further use cases are mentioned in the introduction
    }
} catch(MPICXX::Comm_corrupted_exception &e) { // catch a corrupted communicator exception
    // repair communicator, initiate rollback, ...
    // see Section II.B ULFM for further information
}
```

error state. When all ranks reach the barrier, the propagating ranks cancel the pending send requests, which are the send requests to the ranks that got signaled by another rank. Then all ranks perform an `MPI_Allreduce` operation with an `MPI_BAND` operator to determine if the communicator is corrupted, i.e. `signal_error` was called by the destructor of `Comm` during stack unwinding. If the call results positive, all ranks throw a `Comm_corrupted_exception`, otherwise the following algorithm determines the failed ranks and respective error codes:

*Determine failed ranks and codes:* Once one or more errors have been signaled, all ranks throw a `Propagated_exception` and all information is propagated to all ranks. For that, we do an `MPI_Scan` with the operation `MPI_SUM`, where failed ranks participate with a 1 and non-failed ranks with a 0. This assigns every failed node an `index`. The number of failed nodes is then propagated by an `MPI_Bcast` of the last rank (i.e. rank `size−1`). Now all ranks allocate memory for the rank numbers and error codes of the failed ranks and initialise it with zeros. The failed ranks write their rank number and error code to this array with respect to their `index`. Finally an `MPI_Allreduce` with `MPI_MAX` is performed to propagate all the information.

*Future:* During computations the user initiates non-blocking requests by calling the `send` or `recv` method of the `Comm` object. The respective requests are stored in the `request` field of a `Future` object. Instead of just waiting for the `send`/`recv` request to finish in the `wait` method, `MPI_Waitany` is used that waits for either the `request` or the error requests `err_req` to complete. It is possible that `MPI_Waitany` completes `request` while an error was signaled as well. Therefore, if `MPI_Waitany` completes `request`, the method uses `MPI_Test` to check whether an error was signaled. If no error code is received, the program continues its computations. Otherwise an error code is received, the above described algorithm is executed to handle the error and throw the appropriate exception.

*Corrupted communicator:* The destructor of the `Comm` class checks whether the object is deconstructed due to a thrown exception. If this is the case, `signal_error` is called. At the following `MPI_Allreduce` this rank participates with a 0, indicating that the communicator is corrupted. All other ranks will throw a `Comm_corrupted` exception.

*Preclusion of deadlocks:* This approach precludes deadlocks that are caused by thrown exceptions, since either the program executes successfully, i.e. all `requests` are completed by the `MPI_Waitany` method or, if an exception is thrown, an error is signaled and the `MPI_Waitany` will return with an error and throw a `Comm_corrupted_exception`. Furthermore, in erroneous cases, the execution path of the ranks can be synchronised with the `signal_error` method.

## C. ULFM adoption

Until ULFM is available on HPC clusters, the presented Black-Channel approach can be used to develop fault-tolerant programs using the interface described in Section III-A. However, as soon as ULFM is available, it constitutes the proper tool to handle failures. As mentioned earlier, ULFM enables the detection of hard failures and even communicates this to other ranks, which our proof-of-concept Black-Channel cannot. Therefore we adapt our implementation to use ULFM features if available. This makes it possible to increase the functionality of user-level code written against our interface in the future, without changing the general strategy to react to erroneous behaviour.

If ULFM is available, the `wait` method of the `Future` invokes an `MPI_Wait`, instead of the `MPI_Waitany`, and checks the return code. This `MPI_Wait` call returns with the error code `MPI_ERR_REVOKED`, if any rank has called `MPI_Comm_revoke`. Also the additional communicator `err_comm` and the pending `MPI_Irecv` requests are not necessary any more.

After the communicator is revoked the function `MPI_Comm_agree` is used to determine whether the communicator is corrupted or an error code is signaled. If the communicator is corrupted a `Comm_corrupted_exception` is thrown, otherwise `MPI_Comm_shrink` is called to obtain a valid communicator. Then we proceed with the same algorithm like in the Black-Channel case to propagate the rank numbers and error codes of the failed ranks.

There are three cases in which the communicator is revoked. The first case is the call of the method `signal_error`. The following `MPI_Comm_agree` proceeds with 1 on all ranks, indicating that the communicator is not corrupted. The other cases are when the communicator object is deconstructed during stack unwinding caused by a thrown exception, or that an MPI call returns `MPI_ERR_PROC_FAILED` or `MPI_ERR_PROC_FAILED_PENDING`. The latter implies a hard failure of a node or rank. In these cases the respective ranks participate with 0 at the following `MPI_Comm_agree`, indicating that the communicator is corrupted and a `Comm_corrupted_exception` is thrown on all ranks.

## IV. VALIDATION AND DISCUSSION

### A. Validation

We tested both implementations on PALMA, the HPC cluster of the University of Münster. We use 12 nodes and 48 nodes, with 12 processes each, i.e. 144 and 576 ranks, respectively. Every node consists of two hexacore Intel Westmere processors, and the nodes are connected by QDR InfiniBand. IntelMPI (version 5.1.3) and OpenMPI (version 1.8.4) use the RDMA protocol on the interconnect. The ULFM variant of OpenMPI (based on OpenMPI

1.7.1) does not support this, thus TCP/IP over InfiniBand is used. This drawback of the OpenMPI-ULFM implementation unfortunately affects the latency of the system. For reference, we also show timings for the newer OpenMPI version and TCP/IP over InfiniBand. In the OSU Benchmark (osu_barrier) [21], the ULFM instance is 35 times slower than IntelMPI, and 6 times slower than the standard OpenMPI installation, see Table I for details.

| IntelMPI | OpenMPI | OpenMPI (tcp) | OpenMPI-ULFM |
|----------|---------|---------------|--------------|
| $16.7\mu s$ | $97.3\mu s$ | $502.6\mu s$ | $585.5\mu s$ |

Table I: Average latency of the OSU Benchmark (osu_barrier, 1000 iterations) on PALMA.

In addition to testing the functionality, we measure the time that is needed to propagate an exception. We measure the time for duplicating `comm_world`, propagating an exception from rank 0 and cleaning up the duplicated communicator, i.e., we simultaneously measure the overhead and propagation time. For the Black-Channel approach this results in two calls of `MPI_Comm_dup` while with ULFM only one call is necessary. We repeat this test 1000 times for the MPI implementations available to us. Figure 2 shows boxplots over the duration measured on the root rank in milliseconds. While the Black-Channel approach is competitive at 12 nodes, we observe that on 48 nodes the Black-Channel approach (for both libraries) is slower than the ULFM implementation, as it is not as optimised as the algorithms used in ULFM. However, the OpenMPI implementation of ULFM is based on an older version of OpenMPI and is not optimised for performance yet, thus further speed-up of our propagation strategy can be expected, once ULFM is integrated into the standard.
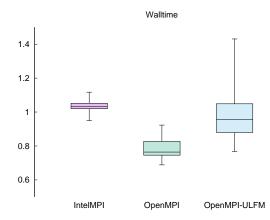
### B. Issues

The main issue with the Black-Channel approach is that it only works properly for point-to-point communication. If we invoke a non-blocking collective communication, we cannot call `MPI_Cancel` for these requests. The standard states explicitly:
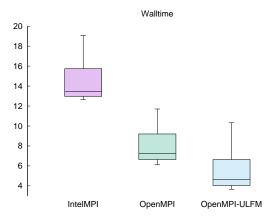
> MPI Standard 3.1 [4], page 197 It is erroneous to call `MPI_REQUEST_FREE` or `MPI_CANCEL` for a request associated with a nonblocking collective operation. [...]
> *Rationale.* Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. *(End of rationale.)*

This implies that all buffers involved in the non-blocking collective communication should be valid until the request finishes (which will, in many cases, never happen). This extra memory and the state information of the actual request



(a) Propagating an error on 12 nodes with 12 processes each.



(b) Propagating an error on 48 nodes with 12 processes each.

Figure 2: Duration of the propagation

in some internal MPI data structures might be negligible in many cases, e.g. `MPI_Ireduce`, but is becoming a problem when gather/scatter operations on a large communicator are called. On the other hand for really large scale computations, collective communication should be avoided anyway and one should work on small subcommunicators, as the cost rises with the number of ranks. Then also the (unavoidable) memory leak is small. The real issue arises when long time computations are considered, as the communicator (according to the current MPI standard) cannot be freed (internally) as long as there are open requests on this communicator.

The problem is solved once the proposed ULFM extension is included in the MPI standard: This deprecates our Black-Channel workaround, and supports to revoke a communicator and cancel all open requests.

An issue with the Black-Channel approach is that the propagation of the error needs at least $n-1$ point-to-point communications that are initiated by a single rank. This might imply problems if many ranks are used and errors occur often. However, optimising the performance of the error propagation strategy needs knowledge of the

network topology and is therefore not discussed further in this paper. ULFM might also be the solution, since the propagation strategy in `MPI_Comm_revoke` is implementation dependent, and MPI implementors are known for providing extremely efficient realisations of the MPI standard.

<div align="center">REFERENCES</div>

[1] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander, "The distributed and unified numerics environment, version 2.4," *Archive of Numerical Software*, vol. 4, no. 100, pp. 13–29, 2016. [Online]. Available: https://journals.ub.uni-heidelberg.de/index.php/ans/article/view/26526

[2] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, "The deal.II library, version 8.5," *Journal of Numerical Mathematics*, vol. 24, no. 3, pp. 135–141, Oct. 2016.

[3] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[4] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 3.1," June 2015, available at http://mpi-forum.org.

[5] J. Dongarra, J. Hittinger, J. Bell, L. Chacón, R. Falgout, M. Heroux, P. Howland, E. Ng, C. Webster, S. Wild, and K. Pau, "Applied mathematics research for exascale computing," U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, Tech. Rep., Mar. 2014.

[6] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. Van Hensbergen, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, May 2014.

[7] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra, "Failure detection and propagation in HPC systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016, pp. 27:1–27:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014941

[8] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012 Proceedings*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds., 2012, pp. 193–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_24

[9] D. Gregor and M. Troyer, "Boost.MPI, version 1.64," April 2017, available at http://www.boost.org/ (Chapter 26).

[10] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using MPI-ULFM," in *EuroMPI/ASIA '14*, 2014, pp. 51:51–51:56.

[11] M. Huber, B. Gmeiner, U. Rüde, and B. Wohlmuth, "Resilience for massively parallel multigrid solvers," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S217–S239, 2016. [Online]. Available: http://dx.doi.org/10.1137/15M1026122

[12] D. Göddeke, M. Altenbernd, and D. Ribbrock, "Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing," *Parallel Computing*, vol. 49, pp. 117–135, Oct. 2015.

[13] J. R. Stearley, R. Riesen, J. H. Laros III, K. B. Ferreira, K. Pedretti, R. A. Oldfield, and R. Brightwell, "Redundant computing for exascale systems," Sandia National Laboratories, Tech. Rep. SAND2010-8709, Dec. 2010.

[14] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/i=1/a=012022

[15] D. Lammers, "The era of error-tolerant computing," *IEEE Spectrum*, vol. 47, no. 11, pp. 15–15, Nov. 2010.

[16] J. Elliott, M. Hoemmen, and F. Mueller, "Evaluating the impact of SDC on the GMRES iterative solver," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*, May 2014, pp. 1193–1202.

[17] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Proceedings of the 2012 IEEE International Conference on Distributed Computing Systems (ICDCS'12)*, 2012, pp. 615–626.

[18] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp. 44:1–44:12.

[19] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, Nov. 2012, pp. 78:1–78:12.

[20] W. Bland, H. Lu, S. Seo, and P. Balaji, "Lessons learned implementing user-level failure mitigation in mpich," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1123–1126.

[21] D. K. Panda, "OSU micro-benchmark," 2013, available at http://mvapich.cse.ohio-state.edu/benchmarks/.