Argonne NATIONAL LABORATORY

User Level Failure Mitigation Ticket #323

Fault Tolerance Working Group Plenary Session

December 2013, MPI Forum Meeting

Chicago, IL USA

U.S. DEPARTMENT OF ENERGY

Outline

- Motivation
- Foundation
- Proposal (w/ rationale)
- Library Example
- Applications work @ ORNL
- Usage cookbook (Aurelien)

Motivation

- Allow wide range of fault tolerance techniques
- Introduce minimal changes to MPI
  - Provide the basic functionality that MPI is missing
- Encourage libraries to build other semantics on top of MPI

Failure Model

- Process failures
  - Explicitly handle fail-stop failures
  - Transient failures are masked as fail-stop
- Silent (memory) errors & Byzantine errors are outside of the scope

Failure Detector

- No explicit definition of failure detector by design
- Failure detectors are very specific to the system they run on
  - Some systems may have hardware support for monitoring
  - All systems can fall back to arbitrary/configurable timeouts if necessary
- This is very much an implementation detail
- Only requirement is that failures are eventually reported if they prevent correct completion of an operation.

First and Foremost…

- Updates the non-FT chapters to not exclude FT (been in the text for a year, but we've neglected to mention them here)
  - Section 2.8 - Error Handling
    - Emphasize that MPI does not provide transparent FT, but point to the new chapter to describe the FT semantics.
    - Specifies that when MPI raises an exception described in the FT chapter, it is still in a defined state and continues to operate.
  - Section 8.3 – Error Handling
    - Updates the text "After an error is detected, the state of MPI is undefined."
    - Now says that unless specified in the FT chapter, the state is undefined.
  - Section 8.7 – Startup
    - Specific text for MPI_FINALIZE relating to process 0.
    - Now points to FT chapter to say that MPI_FINALIZE must always return successfully even in the presence of failures.
  - Section 10.5.4 – Releasing Connections
    - Not calling MPI_FINALIZE is equivalent to process failure.

When FT is not needed…

- The implementation can choose to never raise an error class related to process failure
- Even if an error is never raised, the function stubs must still be provided.
- Available as an option for specific cases:
  - Very small systems
  - Short running jobs
  - Very performance sensitive situations
    - Most failure-free overhead can be low anyway
- There are plans for a future ticket to allow the user to specify whether or not they want FT at init time

Minimum Set of Tools for FT

- Failure Notification
- Failure Propagation
- Failure Recovery
- Fault Tolerant Consensus

Failure Notification

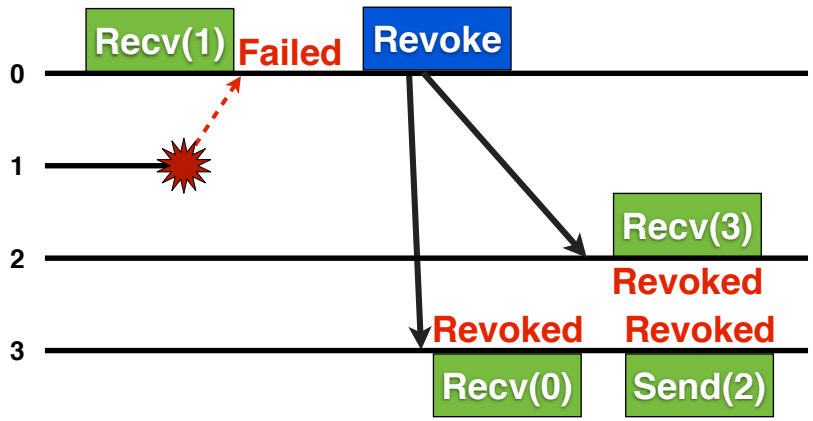
- Local failure notification only
  - Global notification can be built on top of these semantics
- Return error class to indicate process failure
  - **MPI_ERR_PROC_FAILED**
- Errors are only returned if the result of the operation would be impacted by the error
  - i.e. Point-to-point with non-failed processes should work unless routing is broken
- Some processes in an operation will receive MPI_SUCCESS while others will receive MPI_ERR_PROC_FAILED
  - i.e. Collective communication will sometimes work after a failure depending on the communication topology
    - Broadcast might succeed for the top of the tree, but fail for some children
    - Allreduce would always fail if the error occurred before the start of the operation
- Wildcard operations must return an error because the failed process might have been sending the message that would have matched the MPI_ANY_SOURCE.
  - Return MPI_ERR_PENDING for Irecv.
  - If application determines that it's ok, the request can be continued after re-enabling wildcards

Failure Notification (cont.)

- To find out which processes have failed, use the two-phase functions:
  - **MPI_COMM_FAILURE_ACK(MPI_Comm comm)**
    - Internally "marks" the group of processes which are currently locally know to have failed
      - Useful for MPI_COMM_AGREE later
    - Re-enables wildcard operations on a communicator now that the user knows about the failures
      - Could be continuing old wildcard requests or new ones
  - **MPI_COMM_FAILURE_GET_ACKED(MPI_Comm comm, MPI_Group *failed_grp)**
    - Returns an MPI_GROUP with the processes which were marked by the previous call to MPI_COMM_FAILURE_ACK
    - Will always return the same set of processes until FAILURE_ACK is called again
- Must be careful to check that wildcards should continue before starting/restarting a wildcard operation
  - Don't enter a deadlock because the failed process was supposed to send a message
- Future wildcard operations will not return errors unless a new failure occurs.

Failure Propagation

- Often unnecessary
  - Let the application discover the error as it impacts correct completion of an operation.
- When necessary, manual propagation is available.
  - **MPI_COMM_REVOKE(MPI_Comm comm)**
    - Interrupts all non-local MPI calls on all processes in comm.
    - Once revoked, all non-local MPI calls on all processes in *comm* will return **MPI_ERR_REVOKED.**
      - Exceptions are MPI_COMM_SHRINK and MPI_COMM_AGREE (later)
  - Necessary for deadlock prevention
    - Example on right

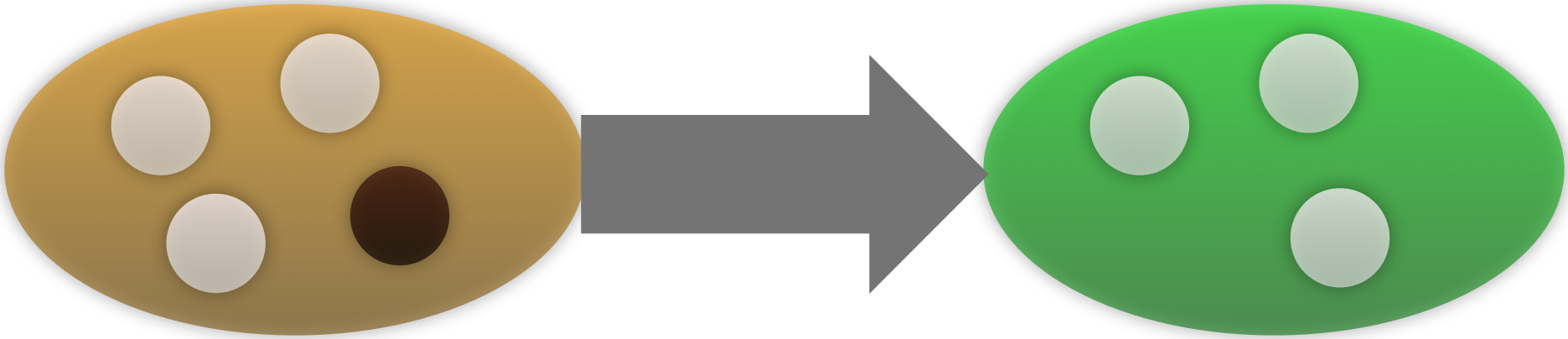


Rationale: MPI_Comm_revoke

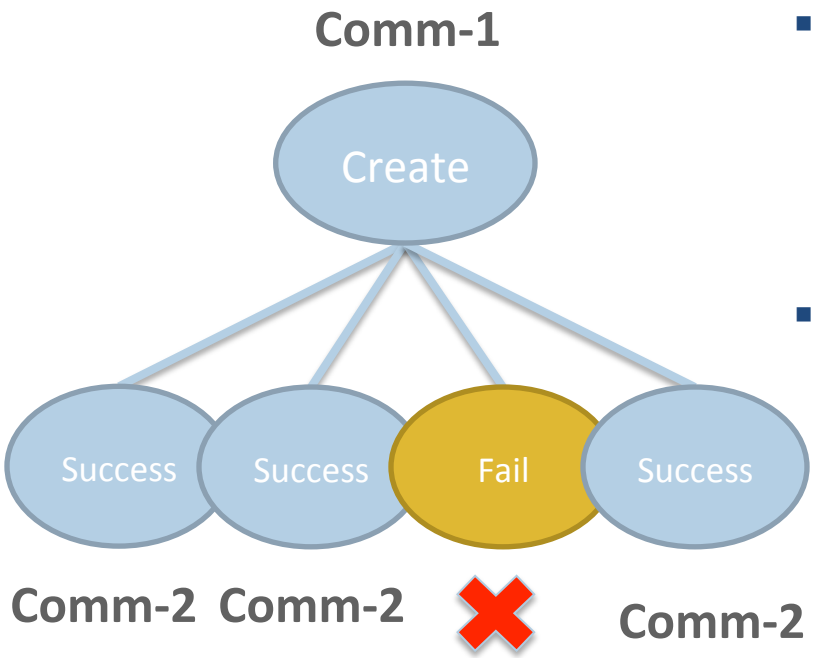
- Why do we revoke the communicator permanently instead of disabling it temporarily?
  - Internally tracking MPI_Request objects after a failure is challenging.
    - Which ones do we need to keep?
    - Which ones do we need to destroy?
  - How does the application know which requests are still good without checking them?
  - This functionality can be added on top of MPI.

Failure Recovery

- Some applications will not need recovery.
  - Point-to-point applications can keep working and ignore the failed processes.
- If collective communications are required, a new communicator must be created.
  - **MPI_Comm_shrink(MPI_Comm *comm, MPI_Comm *newcomm)**
    - Creates a new communicator from the old communicator excluding failed processes
    - If a failure occurs during the shrink, it is also excluded.
    - No requirement that *comm* has a failure. In this case, it will act identically to MPI_Comm_dup.
- Can also be used to validate knowledge of all failures in a communicator.
  - Shrink the communicator, compare the new group to the old one, free the new communicator (if not needed).
  - Same cost as querying all processes to learn about all failures



Why not use MPI_COMM_CREATE_GROUP?



- If a process fails between deciding the group of failed processes and creating the new communicator, it's possible that the new communicator would be produced in an inconsistent state.
  - Some processes have a working communicator that can be used for MPI operations while others have a broken communicator that can't be used for anything.
- An inconsistent communicator can't be correctly revoked or used for an agreement.
  - Impossible to determine if the communicator is OK or notify other processes if the communicator isn't OK.
- *How do we avoid this problem (including non-shrink communicator creation)?*
  - After creating a communicator, perform an Allreduce. If the result is OK, then the communicator is usable, otherwise, release it and create the communicator again.

Communicator Creation

```
rc = MPI_Comm_creation_fn(comm, …, &newcomm);

if (MPI_ERR_PROC_FAILED == rc)
      MPI_Comm_Revoke(&comm);

if (MPI_SUCCESS != MPI_Barrier(comm)) {
      MPI_Comm_Revoke(&newcomm);
      MPI_Comm_free(&newcomm);
}
```


Modified MPI_COMM_FREE semantics

- MPI_COMM_FREE is defined to be collective, though often implemented locally.
- If it's not possible to do collective operations, we should still be able to clean up the handle.
- Modify MPI_COMM_FREE to say that if the collective meaning of MPI_COMM_FREE cannot be established due to failure, the implementation can still clean up the local resources.
  - The handle is still set the MPI_COMM_NULL to signify that the resources are free.
  - The function should still return an error class to show that the collective meaning was not achieved.

Fault Tolerant Consensus

- Sometimes it is necessary to decide if an algorithm is done.
  - **MPI_COMM_AGREE(MPI_comm comm, int *flag);**
    - Performs fault tolerant agreement over boolean *flag*
    - Non-acknowledged, failed processes cause MPI_ERR_PROC_FAILED.
    - Will work correctly over a revoked communicator.
  - Expensive operation. Should be used sparingly.
  - Can also pair with collectives to provide global return codes if necessary.
- Can also be used as a global failure detector
  - Very expensive way of doing this, but possible.
- Also includes a non-blocking version

One-sided

- **MPI_WIN_REVOKE**
  - Provides same functionality as MPI_COMM_REVOKE
- The state of memory targeted by any process in an epoch in which operations raised an error related to process failure is undefined.
  - Local memory targeted by remote read operations is still valid.
  - It's possible that an implementation can provide stronger semantics.
    - If so, it should do so and provide a description.
  - We may revisit this in the future if a portable solution emerges.
- MPI_WIN_FREE has the same semantics as MPI_COMM_FREE

Passive Target Locks

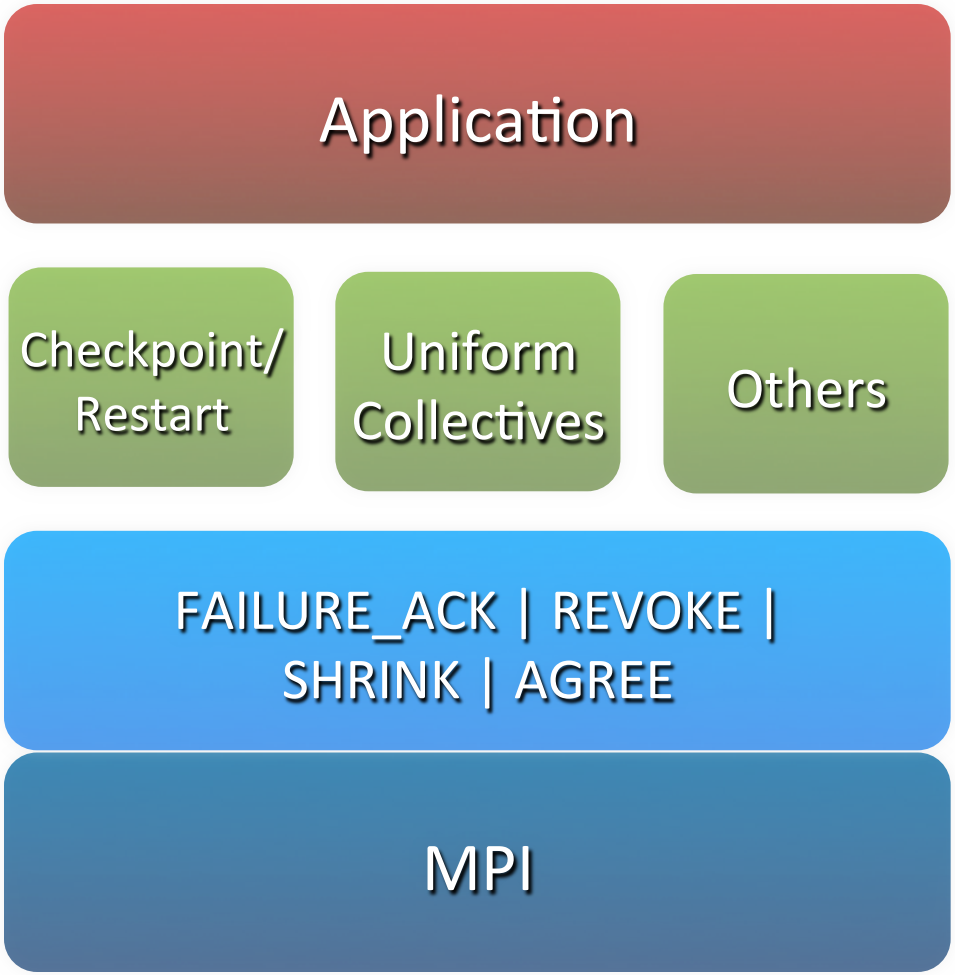
- Without restricting lock implementations, it's difficult to define the status of a lock after a failure
  - With some lock implementations, the library doesn't know who is holding the lock at any given time.
  - If the process holding the lock fails, the implementation might not be able to recover that lock portably.
    - Some other process should be notified of the failure and recovery can continue from there (probably with MPI_WIN_REVOKE).
    - If the implementation can get around the failure, it should try to do so and mask the failure.

File I/O

- When an error is returned, the file pointer associated with the call is undefined.
  - Local file pointers can be set manually
    - Application can use MPI_COMM_AGREE to determine the position of the pointer
  - Shared file pointers are broken
- **MPI_FILE_REVOKE**
  - Provides same functionality as MPI_COMM_REVOKE
- MPI_FILE_CLOSE has similar to semantics to MPI_COMM_FREE

Minimal Additions to Encourage Libraries

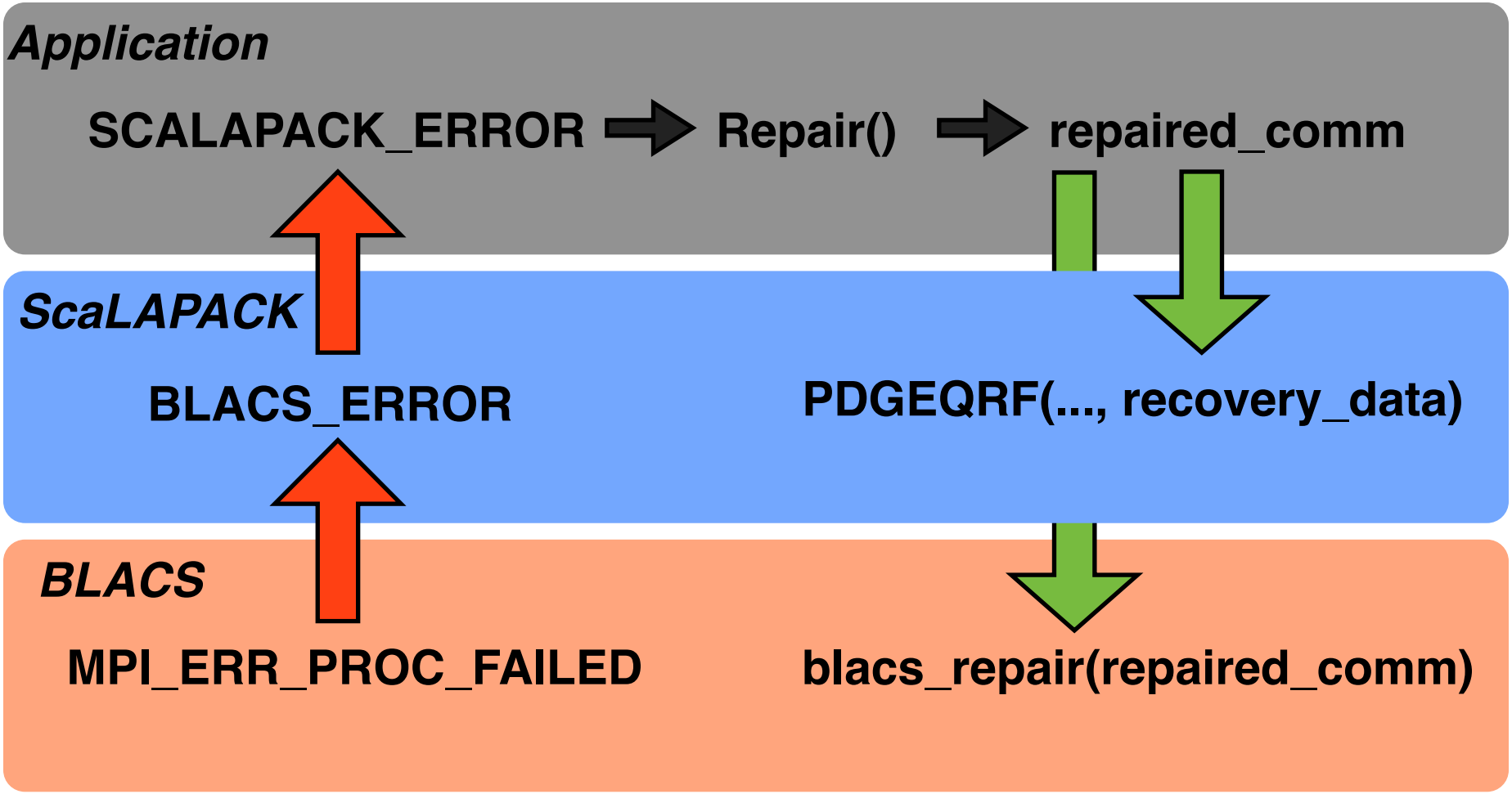
- 5 Functions & 2 Error Classes
  - Not designed to promote a specific recovery model.
  - Encourages libraries to provide FT on top of MPI.
  - In line with original MPI purpose
- Libraries can combine ULFM & PMPI to provide lots of FT models
  - Transactions
  - Transparent FT
  - Uniform Collectives
  - Checkpoint/Restart
  - ABFT
  - Etc.



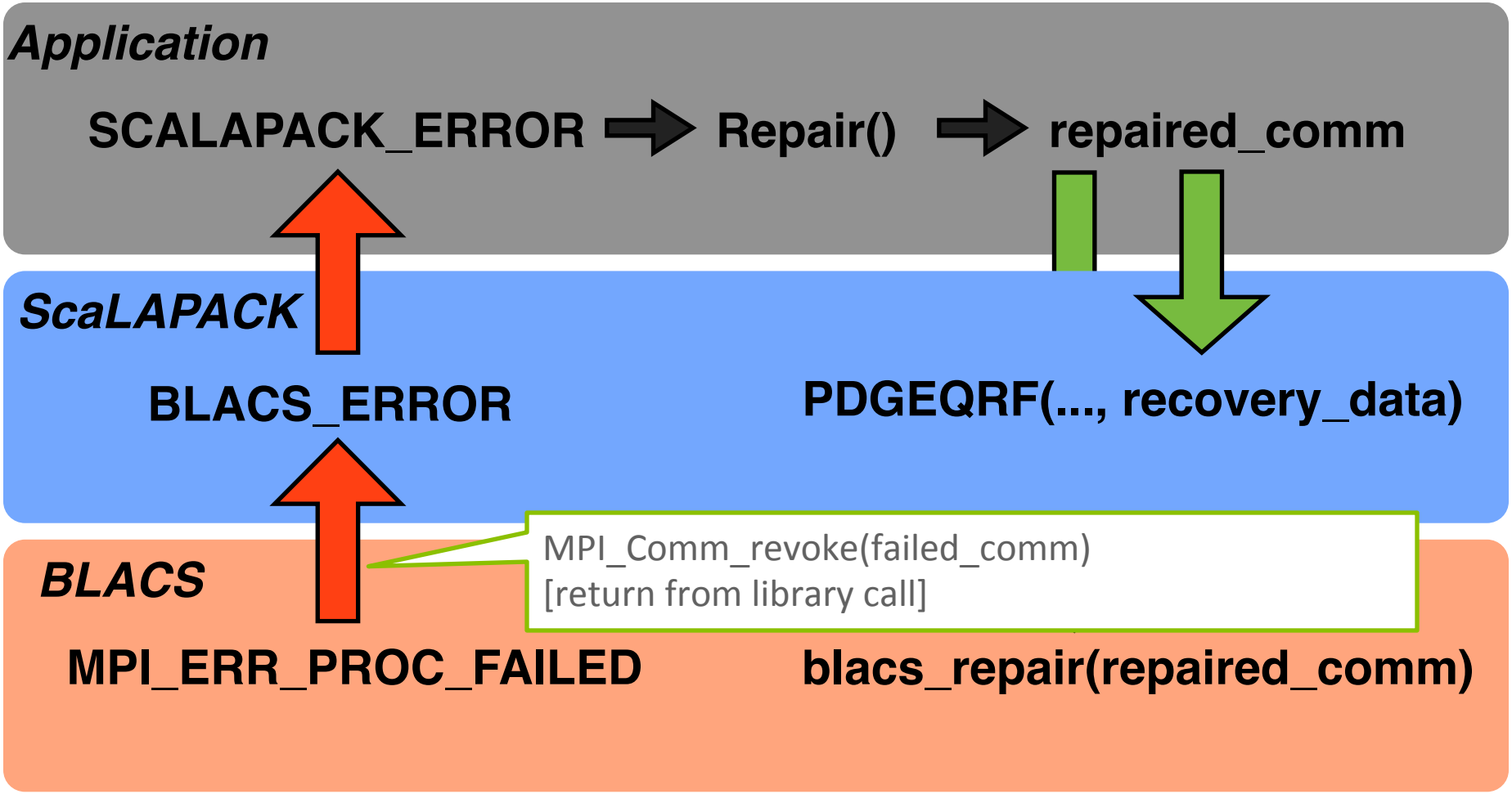
Library Composition Example

- Library Initialization
  - Provide communicator to library (not MPI_COMM_WORLD)
    - Should be doing this already
  - Library creates internal communicator and data
- Status Object
  - Provide place to maintain status between library calls (if necessary)
  - Useful for tracking recovery progress when re-entering previously failed call
    - Current location in algorithm, loop iteration number, delta value, etc.
- Overview
  - Upon failure, library revokes internal communicator and returns to higher level
  - Application repairs the communicator, creates new processes and returns control to library with recovery data
  - Library reinitializes with new communicator
  - Application continues

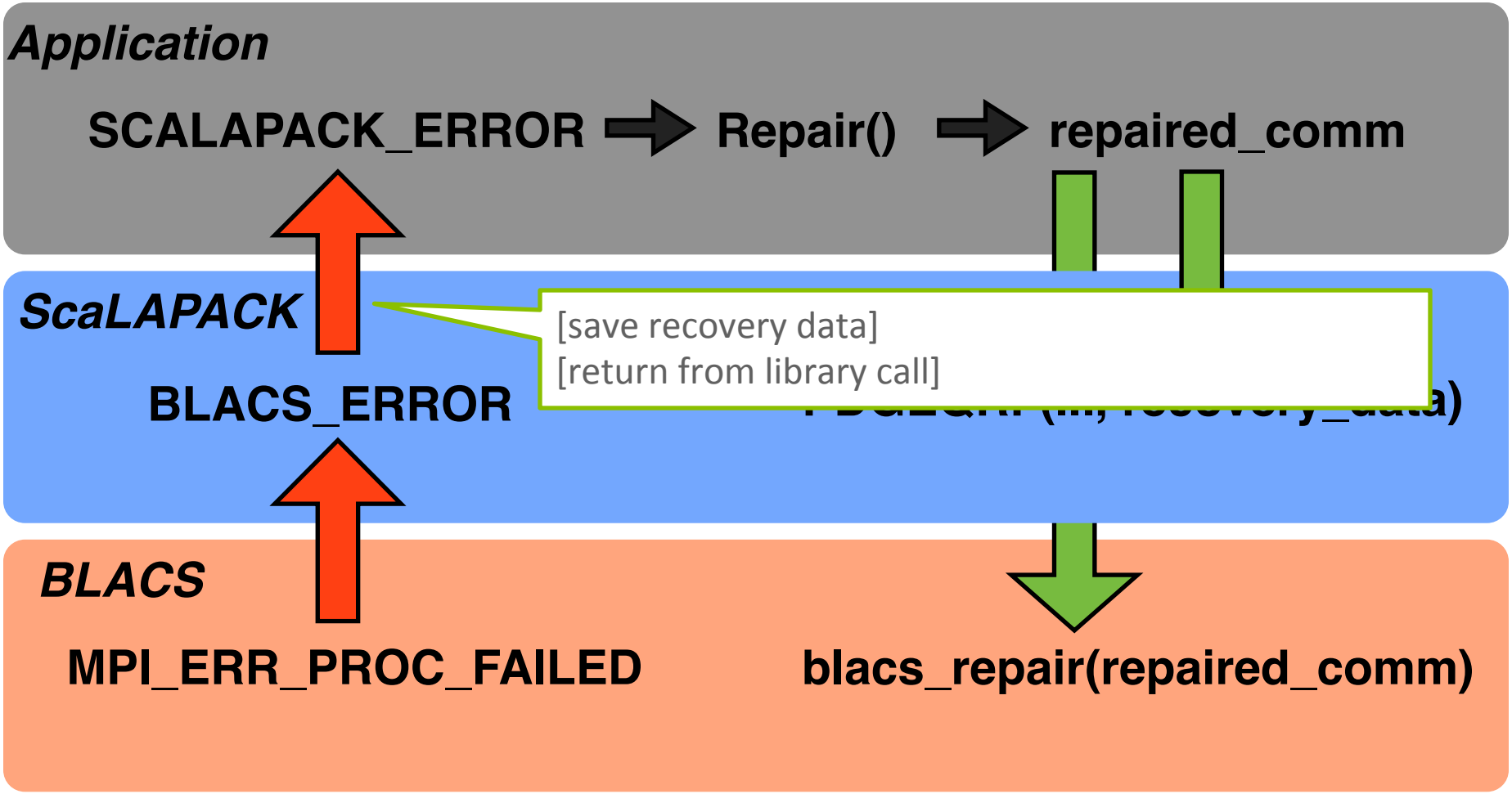
ScaLAPACK Example



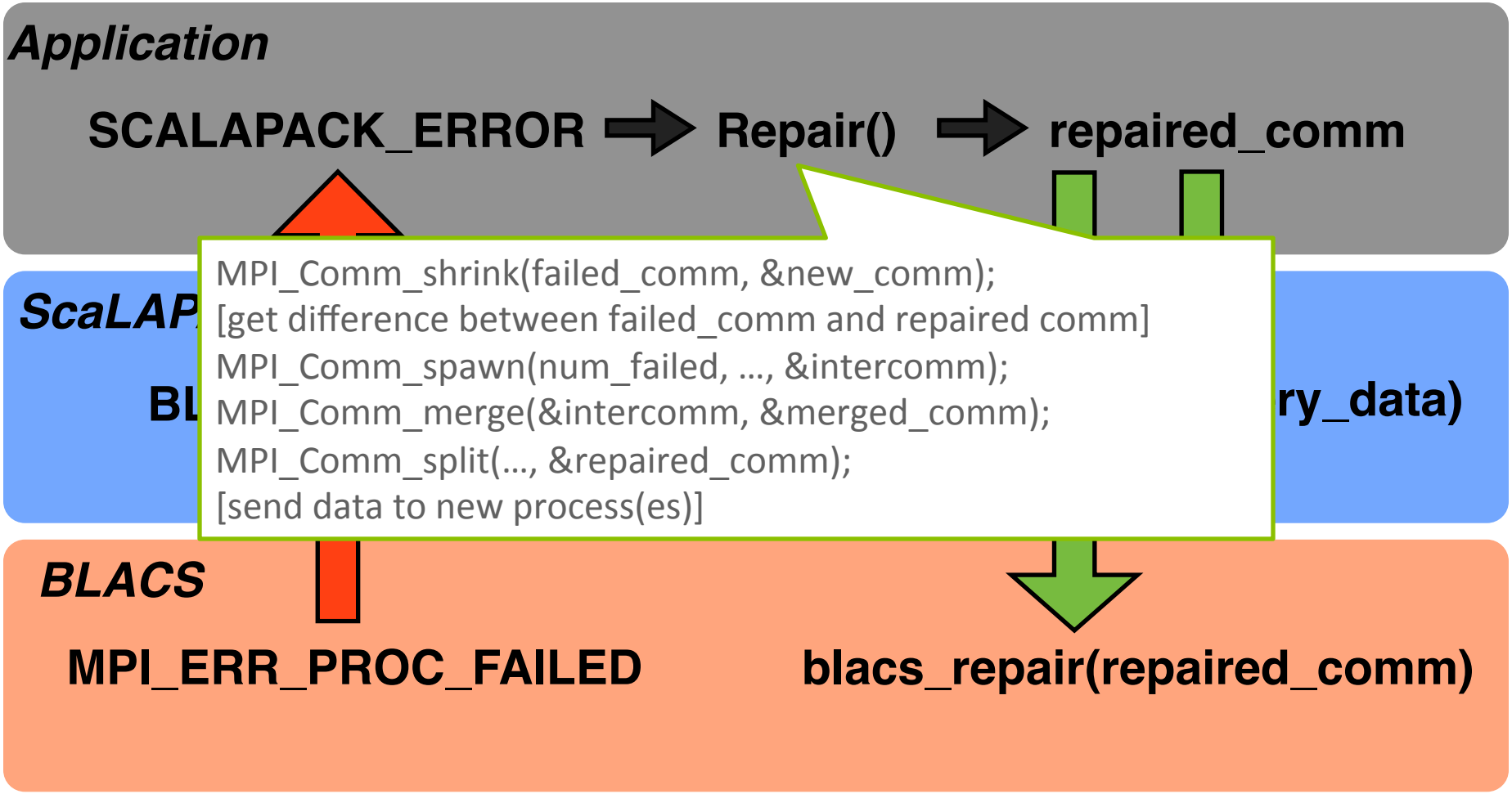
ScaLAPACK Example



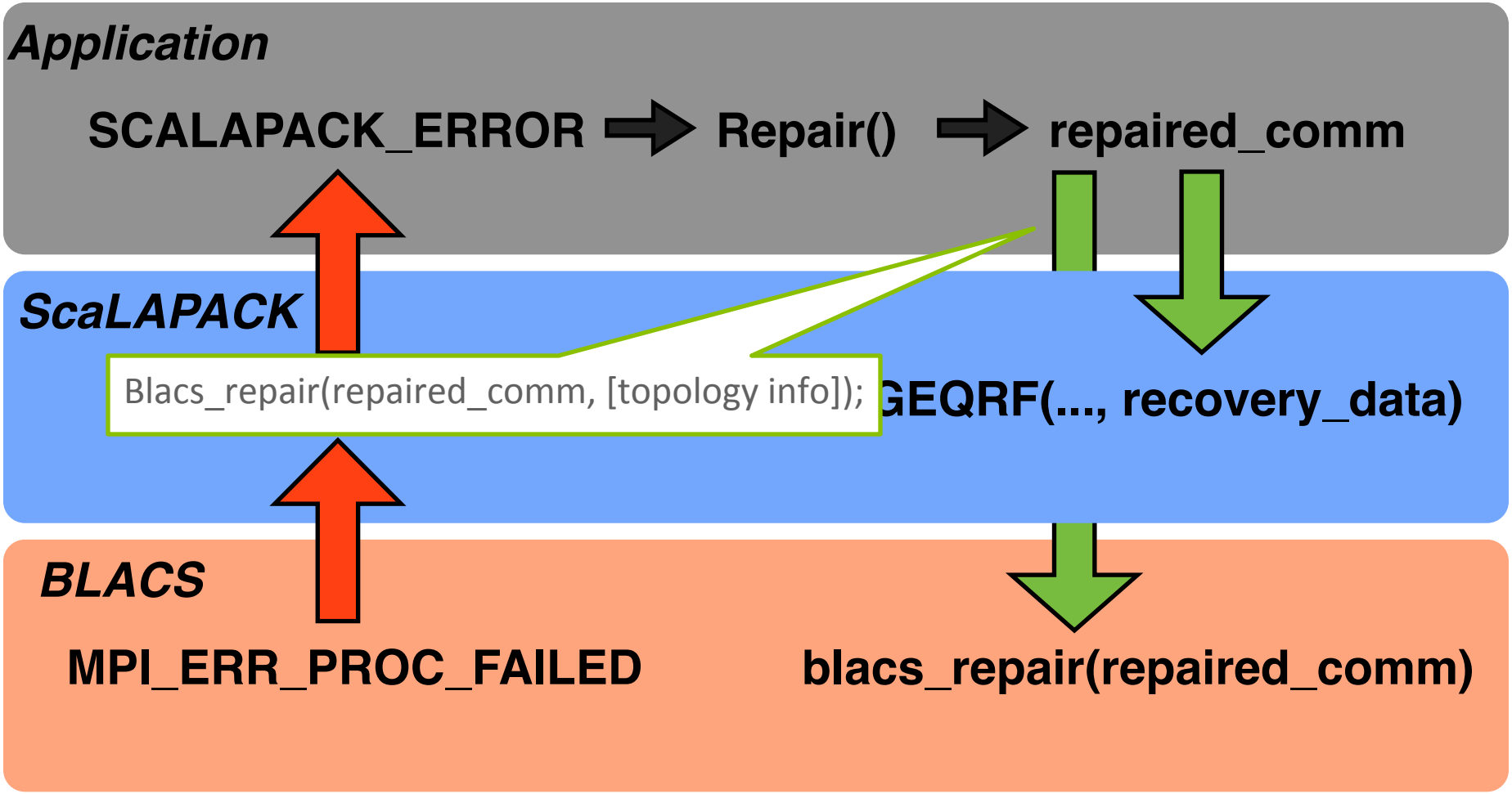
ScaLAPACK Example



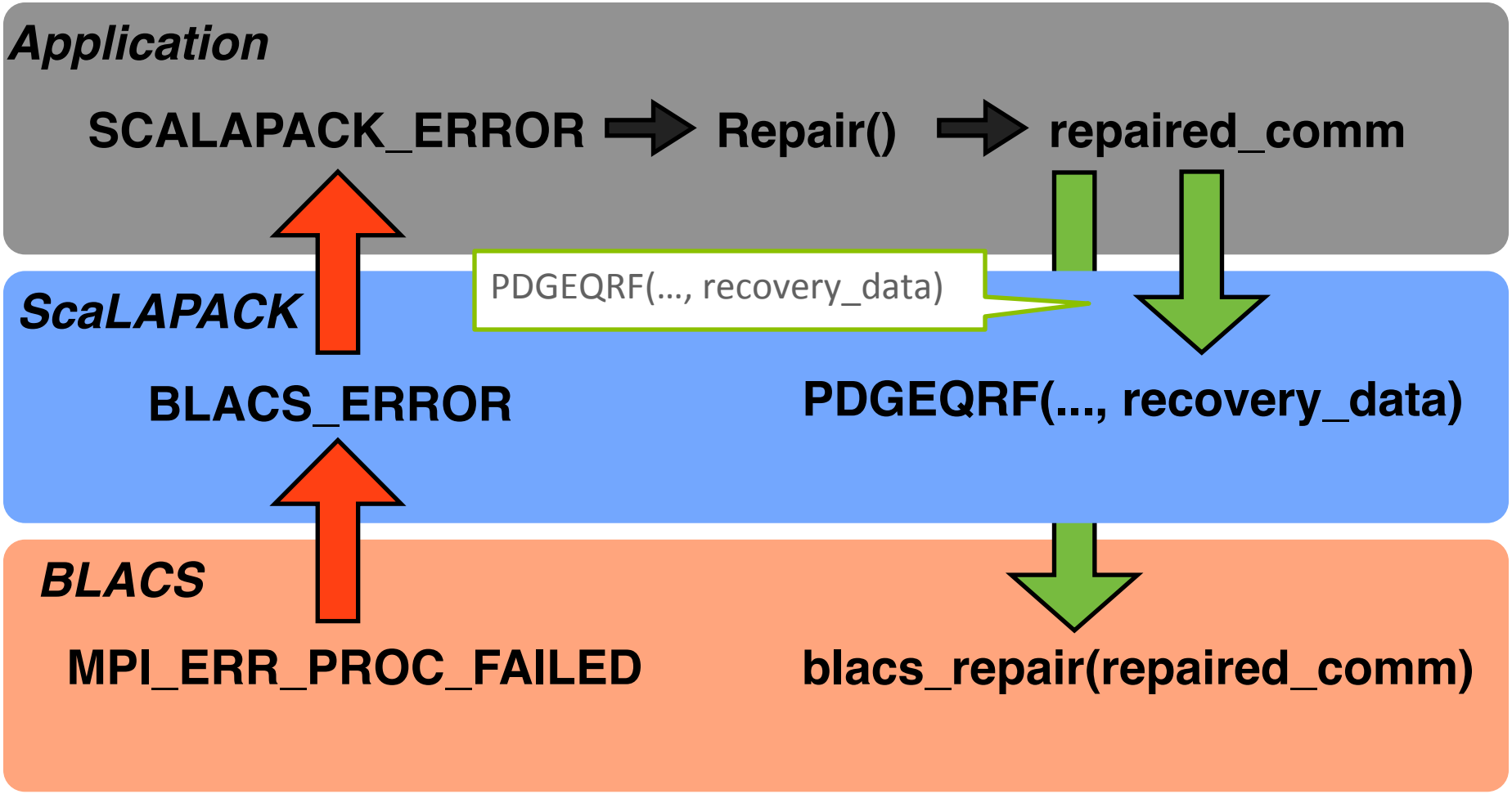
ScaLAPACK Example



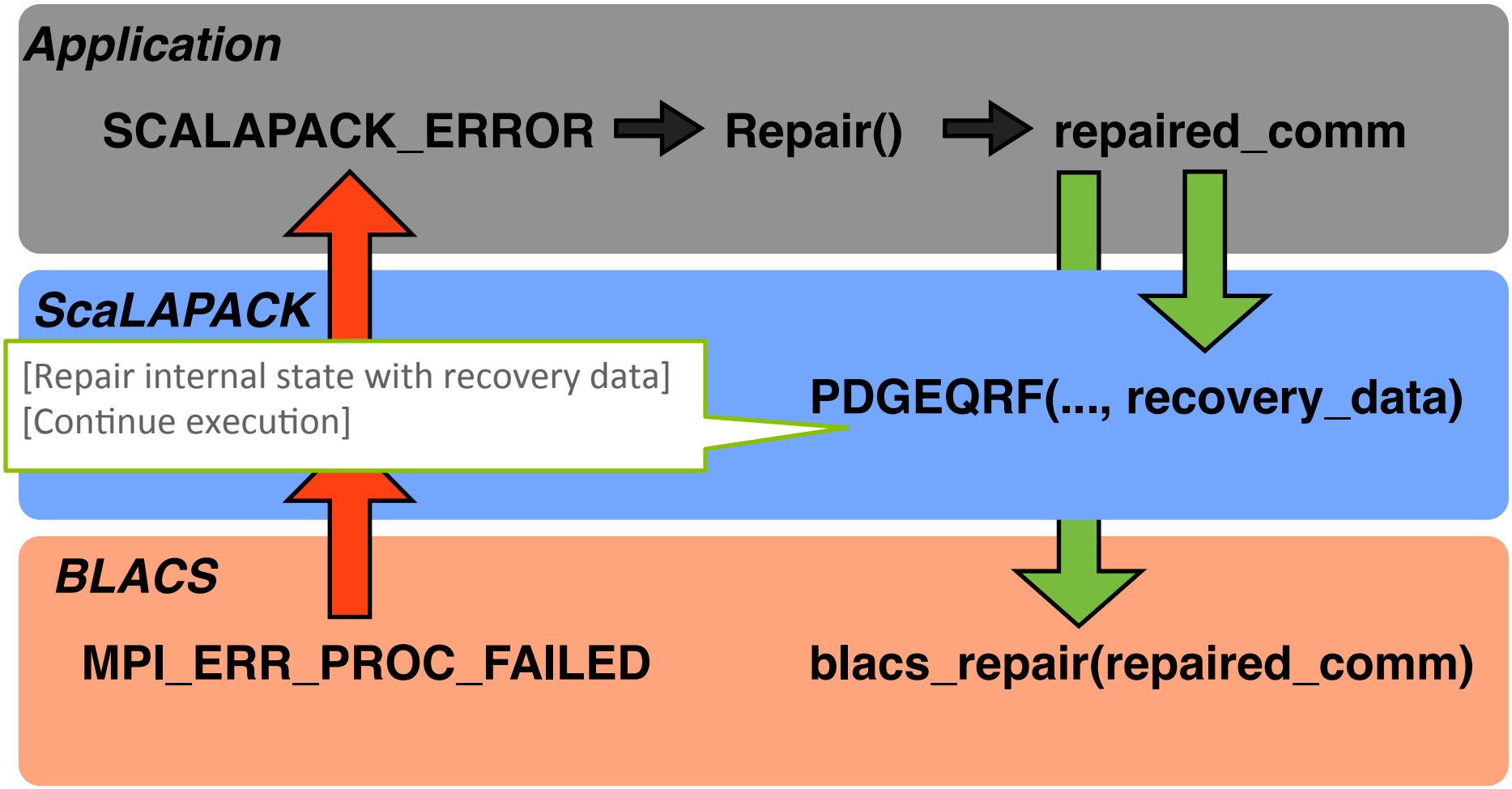
ScaLAPACK Example



ScaLAPACK Example



ScaLAPACK Example



Implementation Status

- Branch of Open MPI (branched in Jan 2012)
  - Feature completed, has external users
  - Available at http://www.fault-tolerance.org
- MPICH implementation
  - Started, not completed

Additional Forthcoming Ticket

- "MPI_INIT_THREAD" style interface for runtime ability to turn off FT
  - One info key would all requesting a certain level of FT support
  - The specifics of this aren't fixed yet, but it could look something like:

MPI_INIT_INFO(requested, provided)

IN	requested	info object describing application needs
OUT	provided	info object describing implementation support

- This would require MPI_INFO objects and the associated functions to work before MPI_INIT
- This will not come out of the FTWG. Still looking for appropriate group.

ULFM related work at ORNL

Swen Boehm, Christian Engelmann, Thomas Naughton,

Geoffroy Vallée, & Manjunath Gorentla Venkata

Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN, USA.

This work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

This work was supported by ORNL National Center for Computational Sciences (NCCS).

Overview

- Experimenting with ULFM at several different levels
  - Testing of MPI-FTWG prototype with a few applications
    - Mini-tests to exercise parts of API
    - Basic MD application extended with ULFM support
  - Extended performance/resilience tools
    - xSim simulator supports ULFM API
    - DUMPI trace library supports ULFM API
  - Extended runtime support
    - Use alternate RTE with ULFM from MPI-FTWG prototype

Application: “SimpleMD”

- Application simulates classical molecular dynamics (MD)
  - Support synchronized application-level checkpoint/restart
  - Optional support for visualization via Visual Molecular Dynamics (VMD)
  - Demonstration code used for testing
- Extended application to support ULFM
  1. Change all MPI_COMM_WORLD to “smd_comm” handle
  2. Change error handler to MPI_ERRORS_RETURN
  3. Modify main simulation loop to recognize process failures (MPI_ERR_PROC_FAILED / MPI_ERR_REVOKED)
  4. On error, all call MPI_Comm_revoke() & MPI_Comm_shrink()
  5. Replace “smd_comm” handle with *newcomm* from shrink
  6. Roll back to previous iteration & continue from previous checkpoint

Basic ULFM Tests

- Set of tests for work with ULFM
  - *ft-shrink-barrier* – Calls revoke/shrink upon collective failure & restarts collective with newcomm. Uses barrier to ensure detection.
  - *ft-shrink-agree* – Calls revoke/shrink upon collect failure & restarts collective with newcomm. Uses an agree to ensure all succeeded, i.e., detect collective error.
  - *ft-agree* – Call agreement function to see cost for non-failure usage.

xSim + ULFM

- Extreme-scale Simulator (xSim)
  - Permits running MPI applications with millions of ranks
  - Uses a lightweight parallel discrete event simulation (PDES)
  - Simulation supports using alternate network & processor models

- Extended xSim for Resilience
  - Process-level fault injection support
  - Support for ULFM API
    - Using point-to-point collectives
    - Support for tests with different network models
    - *TODO*: MPI_Comm_iagree()

MPI Trace Tool with ULFM API

- Enhanced DUMPI library to recognize ULFM API
  - Trace library & text converter tools
- *DUMPI*
  - MPI tracing library to record application execution (function calls)
  - Implemented as PMPI interposition library
  - Developed at Sandia as part of SST project
- Overview
  - Supports individual functions, performance counters (e.g. PAPI)
  - Records function input arguments and (some*) return values
  - Traces recorded as binary files w/ utilities to convert to text
  - Tools to convert to OTF* for visualization in tools like Vampir

** Note: Version we tested had partial support for retvals and OTF converter.*

Alternate Runtime with ULFM

- Extracted MPI layer from ULFM reference prototype
  - Ported *OMPI* layer of ULFM to OpenMPI trunk
- New component for OpenMPI "*rte*" framework
  - Added our runtime as *rte* component
- Combine alternate runtime with OpenMPI / ULFM
  - *OMPI+ULFM* port for OpenMPI trunk
  - RTE component for OpenMPI trunk
  - Initial support being tested at ORNL