Argonne NATIONAL LABORATORY

Proposed Fault Tolerance for MPI-4

Wesley Bland
February 10, 2014
Lawrence Livermore National Laboratory

Some slides courtesy of Aurélien Bouteiller, UTK

U.S. DEPARTMENT OF ENERGY

Outline

- Motivation
- Foundation
- Proposal
- Examples
- Wrapup/QA

Motivation

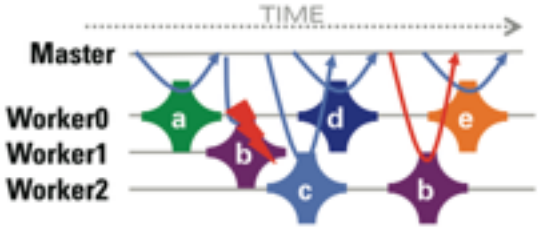
- Allow a wide range of fault tolerance techniques
  - Checkpoint/Restart, Transactions, Roll-forward recovery, Drop failed processes, etc.
- Don't pick a particular technique as better or worse than others
  - Encourage library developers to add libraries on top of this work to make it easy to use for applications
- Introduce minimal changes to MPI

Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

In-place restart (i.e., without disposing of non-failed processes) accelerates recovery, permits in-memory checkpoint

Naturally Fault Tolerant Applications, Master-Worker, Domain Decomposition, etc.

Application continues a simple communication pattern, ignoring failures



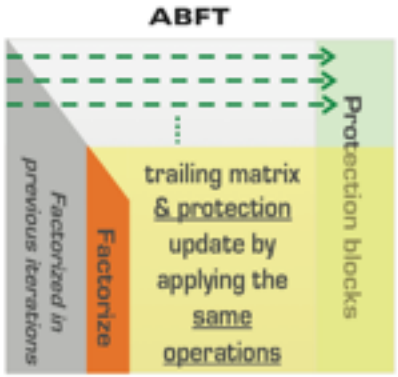
ULFM MPI Specification

Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations

Algorithm Fault Tolerance

ULFM allows for the deployment of ultra-scalable, algorithm specific FT techniques.



Wesley Bland - wbland@mcs.anl.gov

Failure Model

- Fail-stop process failure
  - Transient failures should be masked as if they were fail-stop
- Silent (memory) errors are outside of the scope

Failure Detector

- No explicit failure detector is specified in this chapter
- Failure detectors are specific to the system on which they are run
  - Some systems have hardware support for monitoring
  - All systems can fall back to arbitrary/configurable timeouts if necessary
- Only requirement is that failures are eventually reported if they prevent correct completion of an operation

TL;DR

- 5(ish) new functions (some non-blocking, RMA, and I/O equivalents)
  - MPI_COMM_FAILURE_ACK / MPI_COMM_FAILURE_GET_ACKED
    - Provide information about who has failed
  - MPI_COMM_REVOKE
    - Provides a way to propagate failure knowledge to all processes in a communicator
  - MPI_COMM_SHRINK
    - Creates a new communicator without failures from a communicator with failures
  - MPI_COMM_AGREE
    - Agreement algorithm to determine application completion, collective success, etc.
- 3 new error classes
  - MPI_ERR_PROC_FAILED
    - A process has failed somewhere in the communicator
  - MPI_ERR_REVOKED
    - The communicator has been revoked
  - MPI_ERR_PROC_FAILED_PENDING
    - A failure somewhere prevents the request from completing, but it is still valid

Minimal Set of Tools for FT

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

Failure Notification

- Failure notification is **local**.
  - Notification of a failure for one process does not mean that all other processes in a communicator have also been notified.
- If a process failure prevents an MPI function from returning correctly, it must return **MPI_ERR_PROC_FAILED.**
  - If the operation can return without an error, it should (i.e. point-to-point with non-failed processes.
  - Collectives might have inconsistent return codes across the ranks (i.e. MPI_REDUCE)
- Some operations will always have to return an error:
  - MPI_ANY_SOURCE
  - MPI_ALLREDUCE / MPI_ALLGATHER / etc.
- Special return code for MPI_ANY_SOURCE
  - **MPI_ERR_PROC_FAILED_PENDING**
  - Request is still valid and can be completed later (after acknowledgement on next slide)

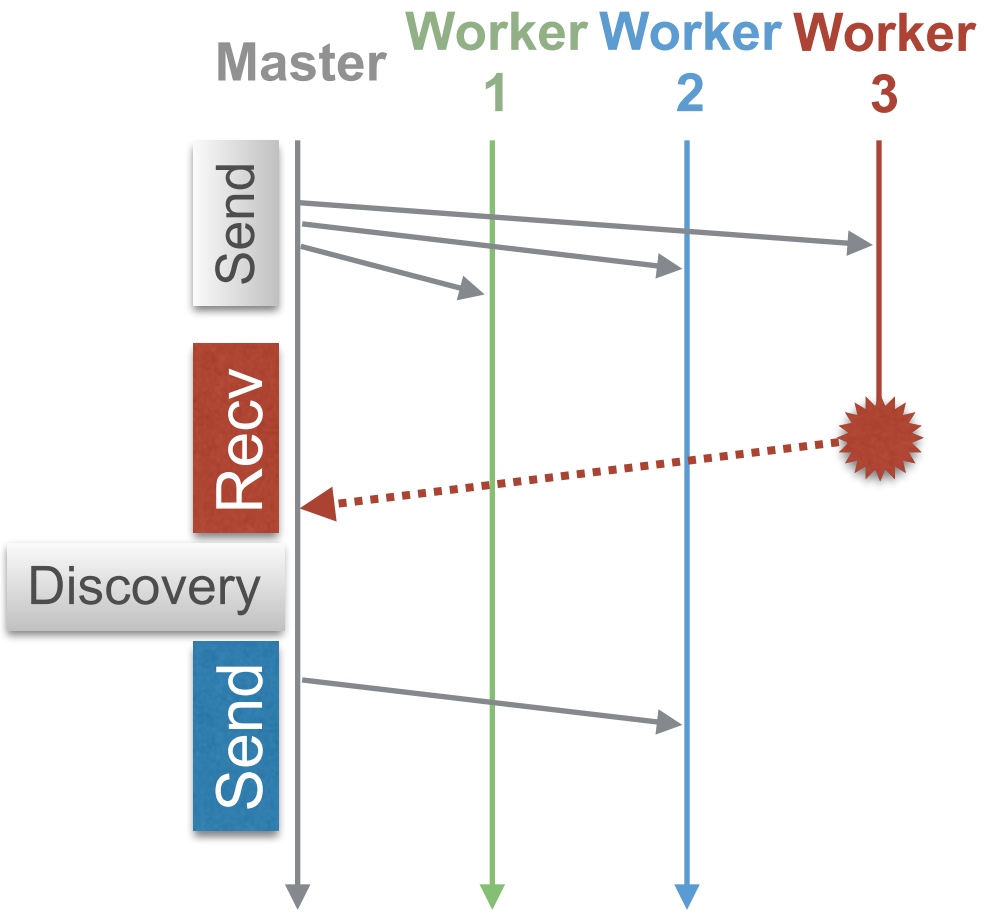
Failure Notification

- 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 MPI_ANY_SOURCE operations on a communicator now that the user knows about the failures
      - Could be continuing old MPI_ANY_SOURCE requests or starting 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 an operation
  - Don't enter a deadlock because the failed process was supposed to send a message
- Future MPI_ANY_SOURCE operations will not return errors unless a new failure occurs.

Recovery with Only Notification

Master/Worker Example

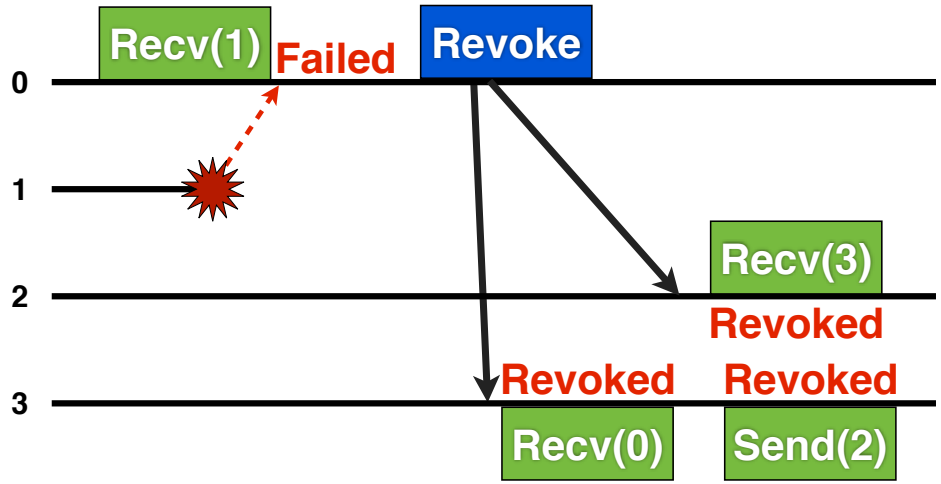
- Post work to multiple processes
- MPI_Recv returns error due to failure
  - MPI_ERR_PROC_FAILED if named
  - MPI_ERR_PROC_FAILED_PENDING if wildcard
- Master discovers which process has failed with ACK/GET_ACKED
- Master reassigns work to worker 2



Wesley Bland - wbland@mcs.anl.gov

Failure Propagation

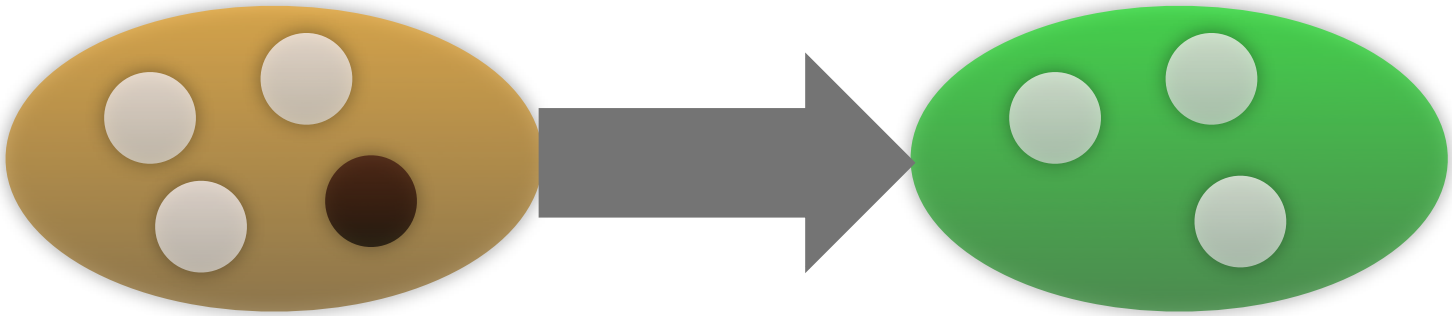
- When necessary, manual propagation is available.
  - **MPI_Comm_revoke**(MPI_Comm comm)
    - Interrupts all non-local MPI calls on all processes in comm.
    - Once revoked, all non-local MPI calls on all processes in comm will return MPI_ERR_REVOKED.
      - Exceptions are MPI_COMM_SHRINK and MPI_COMM_AGREE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



Wesley Bland - wbland@mcs.anl.gov

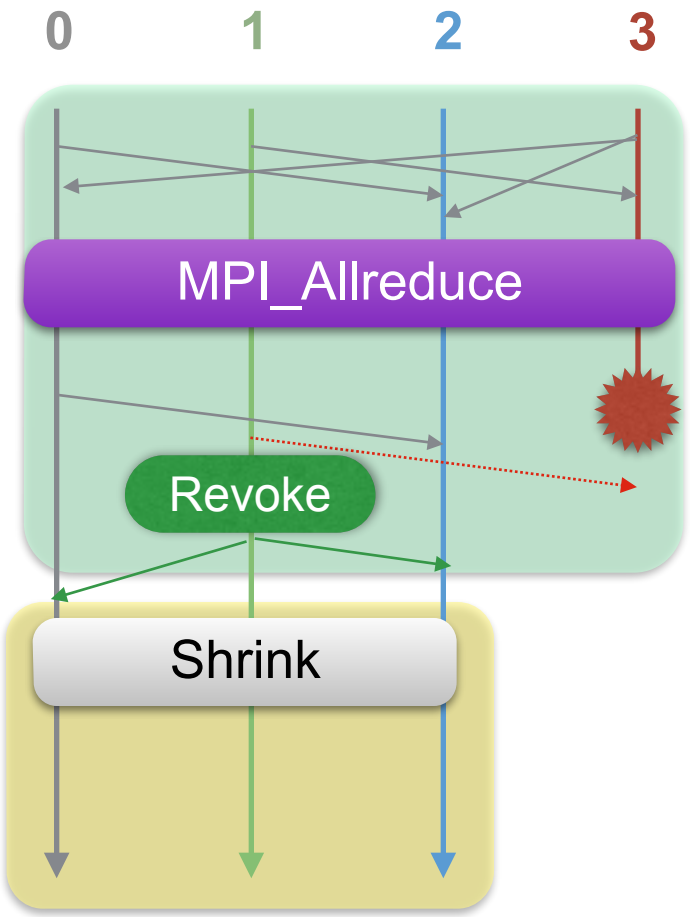
Failure Recovery

- 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



Recovery with Revoke/Shrink

ABFT Example



- ABFT Style application
- Iterations with reductions
- After failure, revoke communicator
  - Remaining processes shrink to form new communicator
- Continue with fewer processes after repairing data

Wesley Bland - wbland@mcs.anl.gov

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

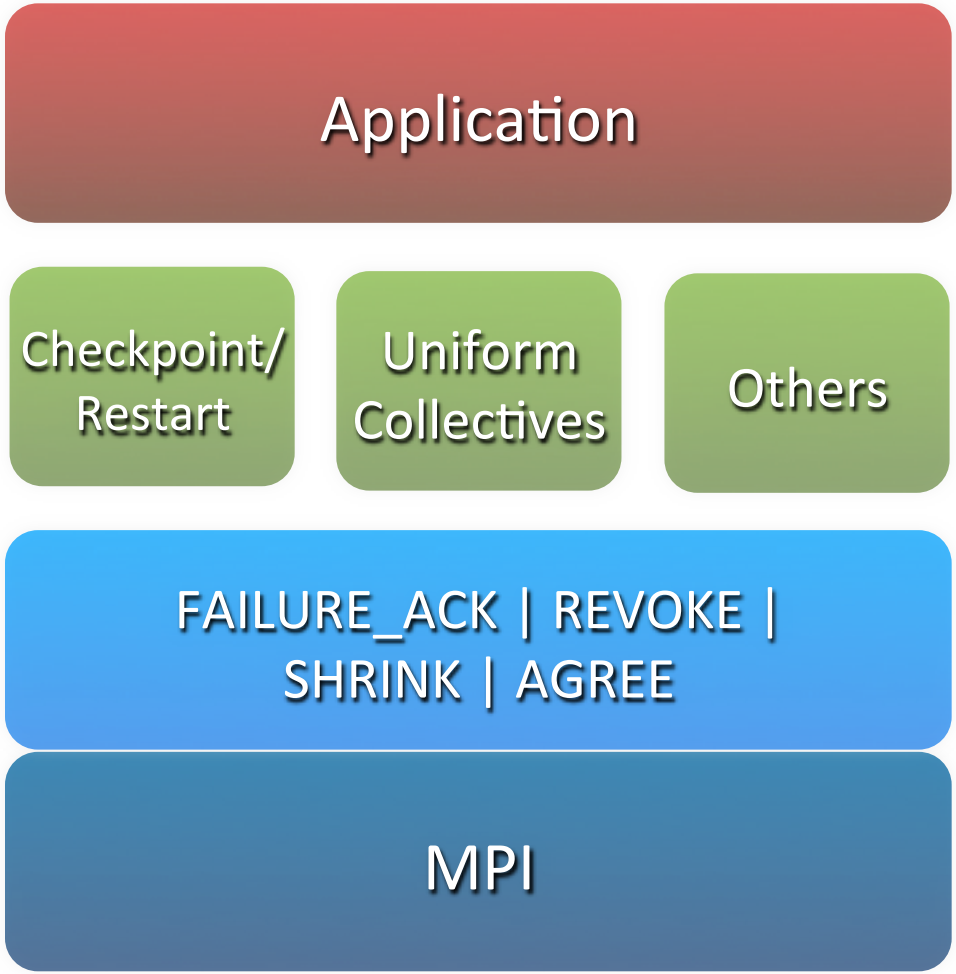
- **MPI_WIN_REVOKE**
  - Provides same functionality as MPI_COMM_REVOKE
- The state of memory targeted by any process in an epoch in which operations raised an error related to process failure is undefined.
  - Local memory targeted by remote read operations is still valid.
  - It's possible that an 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

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.



Implementation Status

- Branch of Open MPI (branched in Jan 2012)
  - Feature complete
  - Available at http://www.fault-tolerance.org
  - Not available in mainline Open MPI
- MPICH Implementation
  - Not completed
  - Targeting March '14

User activities

- ORNL: Molecular Dynamic simulation
  - Employs coordinated user-level C/R, in place restart with Shrink
- UAB: transactional FT programming model
- Tsukuba: Phalanx Master-worker framework
- Georgia University: Wang Landau Polymer Freezing and Collapse
  - Employs two-level communication scheme with group checkpoints
  - Upon failure, the tightly coupled group restarts from checkpoint, the other distant groups continue undisturbed
- Sandia: Sparse solver
  - ???
- Others…

- Cray: CREST miniapps, PDE solver Schwartz, PPStee (Mesh, automotive), HemeLB (Lattice Boltzmann)
- UTK: FTLA (dense Linear Algebra)
  - Employs ABFT
  - FTQR returns an error to the app, App calls new BLACS repair constructs (spawn new processes with MPI_COMM_SPAWN), and re-enters FTQR to resume (ABFT recovery embedded)
- ETH Zurich: Monte-Carlo
  - Upon failure, shrink the global communicator (that contains spares) to recreate the same domain decomposition, restart MC with same rank mapping as before

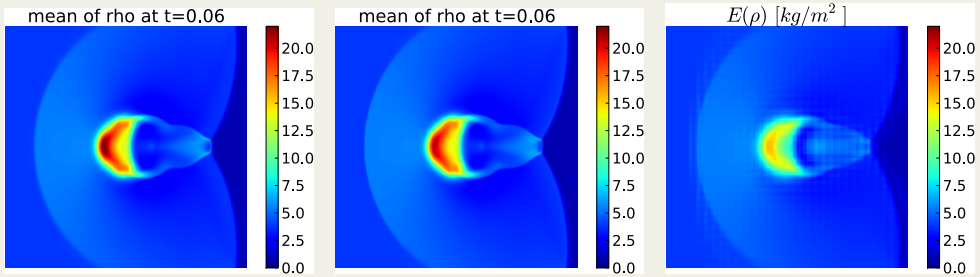


Figure 5. Results of the FT-MLMC implementation for three different failure scenarios.

Credits: ETH Zurich

Questions?

- More examples next from UTK

Creating Communicators, safely

```
int MPIX_Comm_split_safe(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
{
    int rc;
    int flag;

    rc = MPI_Comm_split(comm, color, key, newcomm);
    flag = (MPI_SUCCESS==rc);
    MPI_Comm_agree( comm, &flag);
    if( !flag ) {
        if( rc == MPI_Success ) {
            MPI_Comm_free( newcomm );
            rc = MPI_ERR_PROC_FAILED;
        }
    }
    return rc;
}
```

- Communicator creation functions are collective
- Like all other collective, they may succeed or raise ERR_PROC_FAILED differently at different ranks
- Therefore, caution is needed before using the new communicator: is the context valid at the peer?
- This code snippet solves this uncertainty and makes it simple to create comms again
- Can be embedded into wrapper routines that look like normal MPI (except for communication cost!)

Creating Communicators, safely

```
int APP_Create_grid2d_comms(grid2d_t*
grid2d, MPI_Comm comm, MPI_Comm
*rowcomm, MPI_Comm *colcomm) {
    int rc, rcr, rcc;
    int flag;
    int rank;
    MPI_Comm_rank(comm, &rank);
    int myrow = rank%grid2d->nprows;
    int mycol = rank%grid2d->npcols;

    rcr = MPI_Comm_split(comm, myrow,
rank, rowcomm);
    rcc = MPI_Comm_split(comm, mycol,
 rank, colcomm);
```

```
flag = (MPI_SUCCESS==rcr)
          && (MPI_SUCCESS==rcc);
MPI_Comm_agree( comm, &flag );
if( !flag ) {
        if( MPI_Success == rcr ) {
          MPI_Comm_free( rowcomm );
    }
    if( MPI_Success == rcc ) {
        MPI_Comm_free( colcomm );
    }
    return MPI_ERR_PROC_FAILED;
}
return MPI_SUCCESS;
}
```

- The cost of one MPI_Comm_agree is amortized when it renders consistent multiple operations at once
- Amortization cannot be achieved in “transparent” wrappers, the application has to control when agree is used to benefit from reduced cost

When one needs Revoke 1/2

```
void multiple_comm_collectives(grid2d, comm,…) {
    APP_Create_grid2d_comms(grid2d, comm, rowcomm, colcomm);
    rc = MPI_Allgather(…, rowcomm);
    if( MPI_SUCCESS != rc ) {
        MPI_Comm_revoke(colcomm);
        return;
    }
    rc = MPI_Allgather(…, colcomm);
    if( MPI_SUCCESS != rc ) return;
    compute();
}
```

- If failure is in rowcomm, chances are nobody is dead in colcomm
- A process that raises an exception on rowcomm may not want to participate to the next collective in colcomm
- yet it is not dead, so it has to match its operations, otherwise it is incorrect user code)
- Therefore, this process needs to call Revoke, so that the Allgather doesn't keep waiting on its participation

When one needs Revoke 2/2

```
void one_comm_p2p_transitive_stencil_1d(MPI_Comm comm,…) {
    int rcl=MPI_SUCCESS, rcr=MPI_SUCCESS;
    int leftrank=myrank-1;
    int rightrank=myrank+1;

    for( i=0; i<50; i++ ) {
        if(-1!=leftrank) rcl= MPI_Sendrecv(…, leftrank, …, leftrank,
… ,comm);
        if(np!=rightrank) rcr= MPI_Sendrecv(…, rightrank, …, rightrank,
… ,comm);
        if( MPI_SUCCESS!=rcl || MPI_SUCCESS!=rcr ) {
            MPI_Comm_revoke(comm);
            return;
        }
        dither(); // computation only
    }
```

- If Sendrecv(left=2) fails at rank 3, nobody but rank 1 knows (through sendrecv(right=2))
- A process that raises an exception may not want to continue the dither loop. yet it is not dead, so it has to match its operations, otherwise it is incorrect user code
- This process needs to call Revoke, so that the MPI_Sendrecv(left=3) doesn't keep waiting on its participation

P2P continues across errors

```
void one_comm_p2p_transitive_stencil_1d_norevoke(MPI_Comm comm,…) {
     int rcl=MPI_SUCCESS, rcr=MPI_SUCCESS;
     int leftrank=myrank-1; limax=-1; MPI_Status lstatus;
     int rightrank=myrank+1; rimax=-1; MPI_Status rstatus;

     for( i=0; i<50; i++ ) {
          if(-1!=leftrank && limax<i) //skip over failed iterations
               rcl= MPI_Sendrecv(…, leftrank, sendtag=i, …,
                                 leftrank, MPI_ANY_TAG,
                                 comm, MPI_STATUS_IGNORE);
          if(np!=rightrank && rimax<i)
               rcr= MPI_Sendrecv(…, rightrank, sendtag=i, …,
                                 rightrank, MPI_ANY_TAG,
                                 comm, MPI_STATUS_IGNORE);
          while( MPI_SUCCESS!=rcl ) {
               leftrank--;
               if(-1!=leftrank) {
                    rcl= MPI_Sendrecv(… , leftrank, sendtag=i, …,
                                      leftrank, MPI_ANY_TAG,
                                      comm, &lstatus);
               }
          }
          limax=lstatus.MPI_TAG;
          // (omitted: same stitching for right neighbor)
          dither();
     }
```

- If process on the left fails, stich the chain with next process on the left (some left iteration skipping may happen)
- When a new left neighbor has been found, the normal sendrecv with right will be matched, Communication pattern with right neighbor is unchanged
- Therefore, no need to revoke, dead processes are ignored, algorithm continues on the same comm w/o propagating error condition further

Detecting errors (consistently)

```
void MPIX_Comm_failures_allget(MPI_Comm comm, MPI_Group * grp) {
    MPI_Comm s; MPI_Group c_grp, s_grp;
    MPI_Comm_shrink( comm, &s);
    MPI_Comm_group( c, &c_grp ); MPI_Comm_group( s, &s_grp );
    MPI_Group_diff( c_grp, s_grp, grp );
    MPI_Group_free( &c_grp ); MPI_Group_free( &s_grp );
    MPI_Comm_free( &s );
}
```

- Rationale for not standardizing Failures_allget:
  - agreeing on all failures is as expensive as shrinking the comm (computing a new cid while making the failed group agreement is trivial and negligible)
  - Can be written in 4 lines, with the only mild annoyance of having an intermediate comm object to free.
  - Somewhat, to discourage users to use it when unnecessary (it is expensive, making it easy to spot by having a call to a “recovery” function is good)

Spawning replacement ranks 1/2

```
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm) {
    MPI_Comm shrinked, spawned, merged;
    int rc, flag, flagr, nc, ns;

    redo:
        MPI_Comm_shrink(comm, &shrinked);
        MPI_Comm_size(comm, &nc); MPI_Comm_size(shrinked, &ns);
        rc = MPI_Comm_spawn(…, nc-ns, …, 0, shrinked, &spawned, …);
        flag = MPI_SUCCESS==rc;
        MPI_Comm_agree(shrinked, &flag);
        if( !flag ) {
            if(MPI_SUCCESS == rc) MPI_Comm_free(&spawned);
            MPI_Comm_free(&shrinked);
            goto redo;
        }
        rc = MPI_Intercomm_merge(spawned, 0, &merged);
        flagr = flag = MPI_SUCCESS==rc;
        MPI_Comm_agree(shrinked, &flag);
        MPI_Comm_agree(spawned, &flagr);
        if( !flag || !flagr ) {
            if(MPI_SUCCESS == rc) MPI_Comm_free(&merged);
            MPI_Comm_free(&spawned);
            MPI_Comm_free(&shrinked);
            goto redo;
        }
```

Spawning replacement ranks 2/2

```
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm) {
    …
/* merged contains a replacement for comm, ranks are not ordered properly */
    int c_rank, s_rank;
    MPI_Comm_rank(comm, &c_rank);
    MPI_Comm_rank(shrinked, &s_rank);
    if( 0 == s_rank ) {
        MPI_Comm_grp c_grp, s_grp, f_grp; int nf;
        MPI_Comm_group(comm, &c_grp); MPI_Comm_group(shrinked, s_grp);
        MPI_Group_difference(c_grp, s_grp, &f_grp);
        MPI_Group_size(f_grp, &nf);
        for(int r_rank=0; r_rank<nf; r_rank++) {
            int f_rank;
            MPI_Group_translate_ranks(f_grp, 1, &r_rank, c_grp, f_rank);
            MPI_Send(&f_rank, 1, MPI_INT, r_rank, 0, spawned);
        }
    }
    rc = MPI_Comm_split(merged, 0, c_rank, newcomm);
    flag = (MPI_SUCCESS==rc);
    MPI_Comm_agree(merged, &flag);
    if( !flag ) { goto redo; } // (removed the Free clutter here)
```

Example: in-memory C/R

```
int checkpoint_restart(MPI_Comm *comm) {
    int rc, flag;
    checkpoint_in_memory(); // store a local copy of my checkpoint
    rc = checkpoint_to(*comm, (myrank+1)%np); //store a copy on myrank+1
    flag = (MPI_SUCCESS==rc); MPI_Comm_agree(*comm, &flag);
    if( !flag ) { // if checkpoint fails, we need restart!
        MPI_Comm newcomm; int f_rank; int nf;
        MPI_Group c_grp, n_grp, f_grp;
redo:
        MPIX_Comm_replace(*comm, &newcomm);
        MPI_Comm_group(*comm, &c_grp); MPI_Comm_group(newgroup, &n_grp);
        MPI_Comm_difference(c_grp, n_grp, &f_grp);
        MPI_Group_size(f_grp, &nf);
        for(int i=0; i<nf; i++) {
            MPI_Group_translate_ranks(f_grp, 1, &i, c_grp, &f_rank);
            if( (myrank+np-1)%np == f_rank ) {
                serve_checkpoint_to(newcomm, f_rank);
            }
        }
        MPI_Group_free(&n_grp); MPI_Group_free(&c_grp); MPI_Group_free(&f_grp);
        rc = MPI_Barrier(newcomm);
        flag=(MPI_SUCCESS==rc); MPI_Comm_agree(*comm, &flag);
        if( !flag ) goto redo; // again, all free clutter not shown
        restart_from_memory(); // rollback from local memory
        MPI_Comm_free(comm);
        *comm = newcomm;
    }
```

Creating Communicators, variant 2

- Communicator creation functions are collective
- Like all other collective, they may succeed or raise ERR_PROC_FAILED differently at different ranks
- Therefore, users need extra caution when using the resultant communicator: is context valid at the target peer?
- This code snippet solves this uncertainty and makes it simple to create comms again.

```
int MPIX_Comm_split_safe(MPI_Comm comm, int
color, int key, MPI_Comm *newcomm) {
    int rc1, rc2;
    int flag;

    rc1 = MPI_Comm_split(comm, color, key,
newcomm);
    rc2 = MPI_Barrier(comm);
    if( MPI_SUCCESS != rc2 ) {
        if(MPI_SUCCESS == rc1 {
            MPI_Comm_revoke(newcomm);
        }
    }
    /* MPI_SUCCESS == rc2 => MPI_SUCCESS == rc1*/
    /* note: when MPI_SUCCES!=rc2, we may revoke
     * a fully working newcomm, but it is safer */
    return rc2;
}
```