Resilient applications using MPI-level constructs

SC'19 Fault Tolerant MPI Tutorial

ICL UT
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

Getting the hands-on material

- Slides, examples, instructions:

http://fault-tolerance.org/sc19

- Examples direct link:

http://fault-tolerance.org/downloads/sc19-examples.tgz

- Tag along and run with The ULFM Docker image

http://fault-tolerance.org/ulfm2-docker/

1. Install Docker
2. Docker pull abouteiller/mpi-ft-ulfm
3. source dockervars.sh
4. mpirun -np 10 example

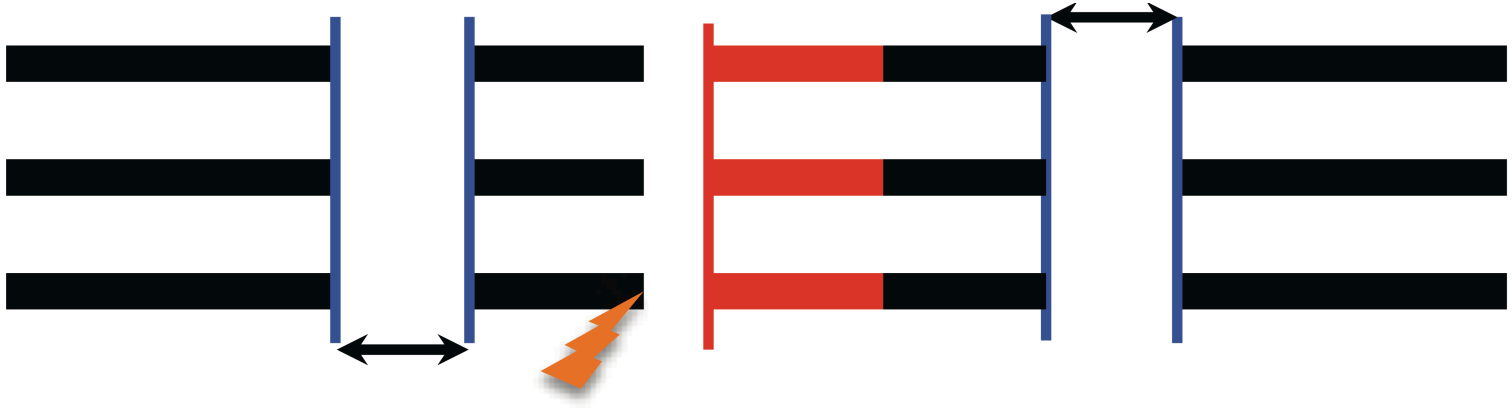
Hands On: Fault Tolerant MPI with ULFM

Aurelien Bouteiller @SC19

A failure, you say?

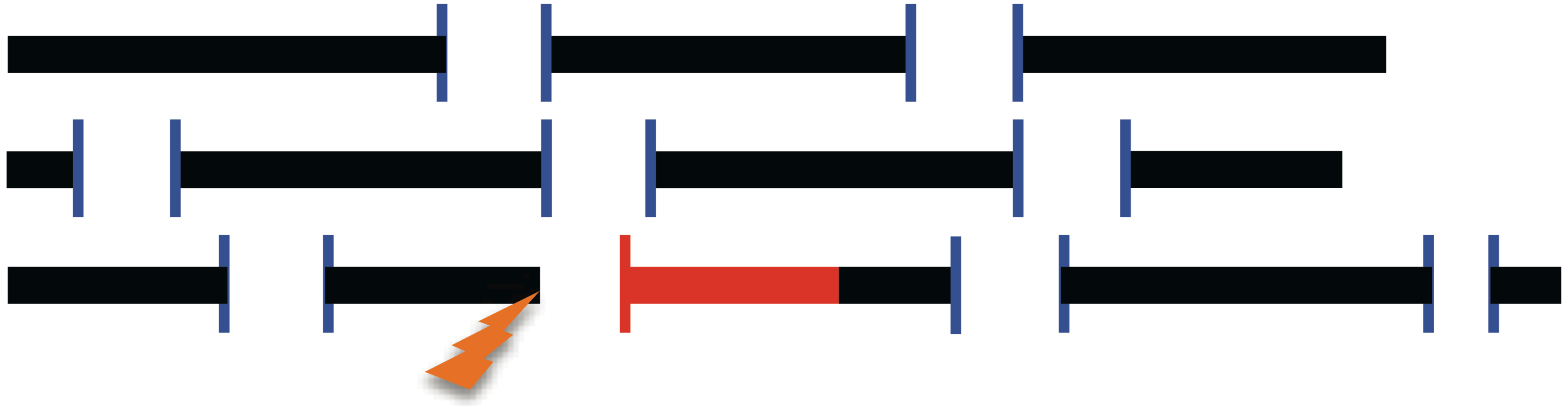
Backward recovery: C/R

Coordinated checkpoint (possibly with incremental checkpoints)



- Coordinated checkpoint is the workhorse of FT today
  - I/O intensive, significant failure free overhead ☹
  - Full rollback (1 fails, all rollback) ☹
  - Can be deployed w/o MPI support ☺
- ULFM enables user-level deployment of in-memory, Buddy-checkpoints, Diskless checkpoint
  - Checkpoints stored on other compute nodes
  - No I/O activity (or greatly reduced), full network bandwidth
  - Potential for a large reduction in failure free overhead, better restart speed

Uncoordinated C/R

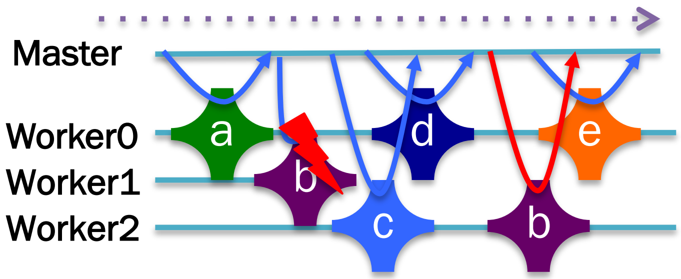


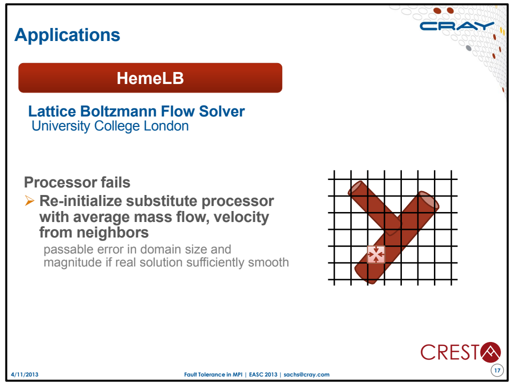
- Checkpoints taken independently
- Based on variants of Message Logging
- 1 fails, 1 rollback
- Can be implemented w/o a standardized user API
- Benefit from ULFM: implementation becomes portable across multiple MPI libraries

- 8:45am - 9:10am Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS) Location 301-302-303; **Asynchronous Receiver-Driven Replay for Local Rollback of MPI Applications**; *Nuria Losada, Aurelien Bouteiller, George Bosilca*

Forward Recovery

- Forward Recovery: *Any technique that permit the application to continue without rollback*
  - Master-Worker with simple resubmission
  - Iterative methods, Naturally fault tolerant algorithms
  - Algorithm Based Fault Tolerance
  - Replication *(the only system level Forward Recovery)*
- No checkpoint I/O overhead
- No rollback, no loss of completed work
- May require (sometime expensive, like replicates) protection/recovery operations, *but still generally more scalable than checkpoint ☺*
- Often requires in-depths algorithm rewrite (in contrast to automatic system based C/R) ☹





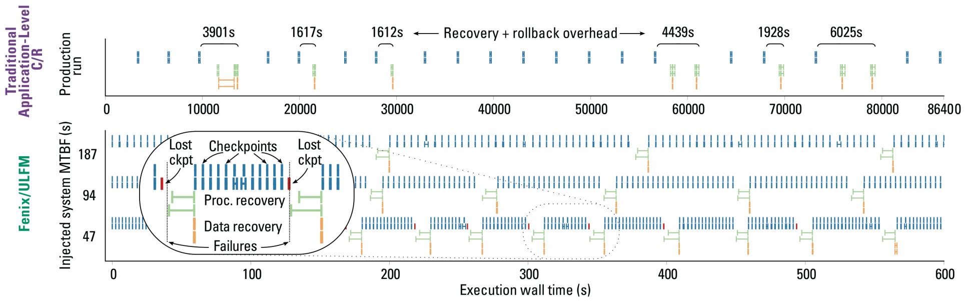
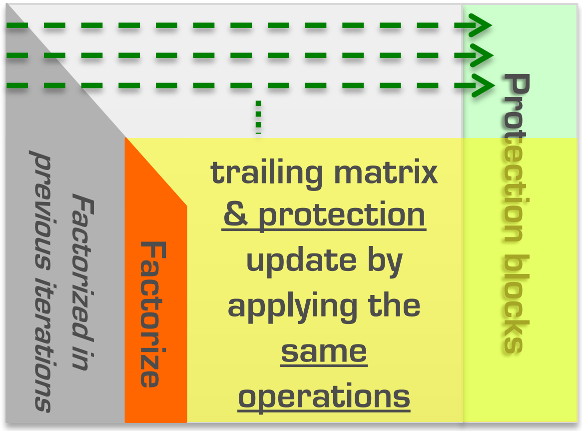


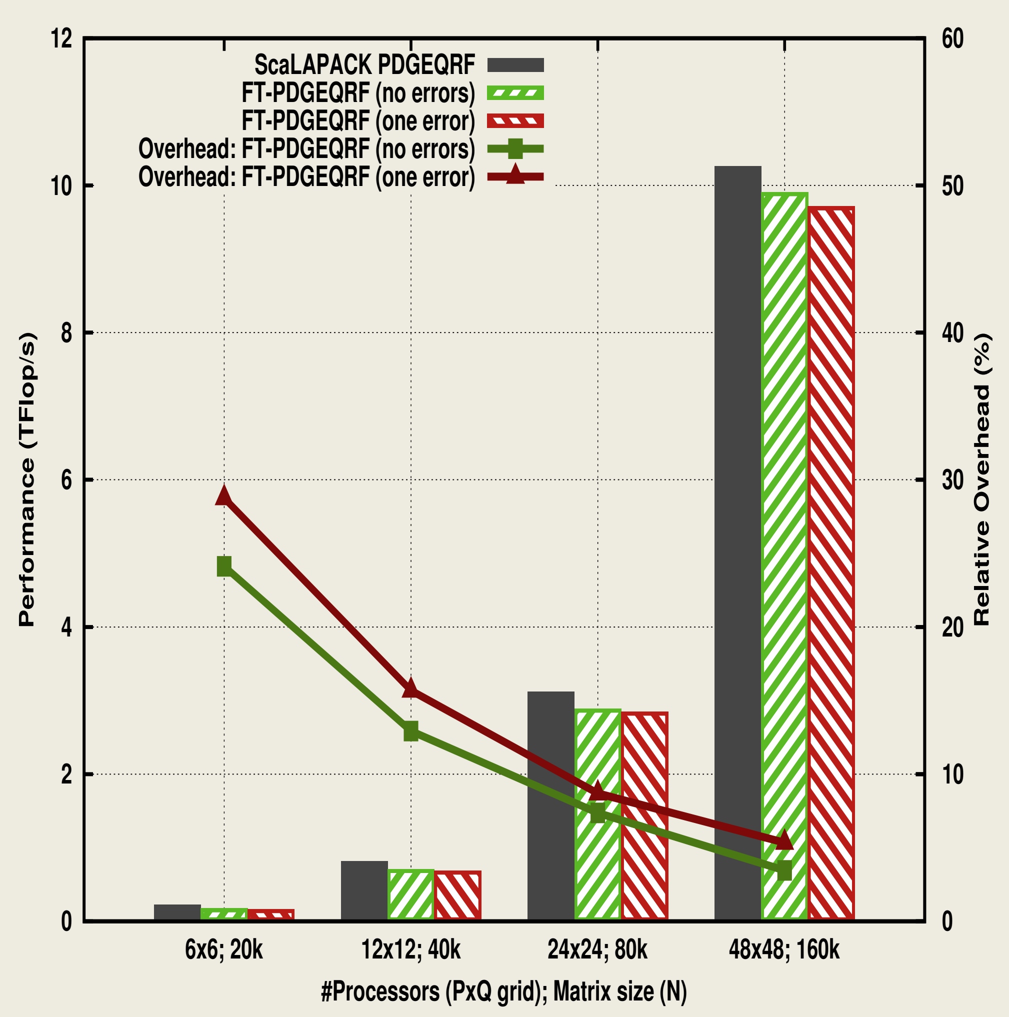
Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

Application specific forward recovery

- Algorithm specific FT methods
  - Not General, but…
  - Very scalable, low overhead ☺
  - *Can't be deployed w/o a fault tolerant MPI*

ABFT

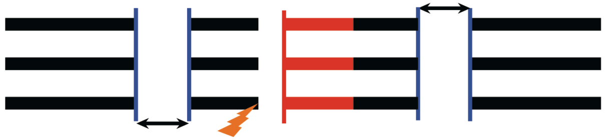




An API for diverse FT approaches

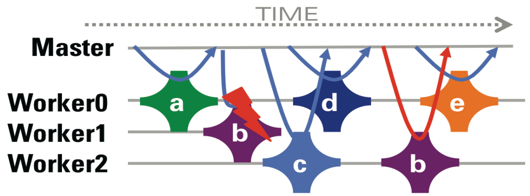
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

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.

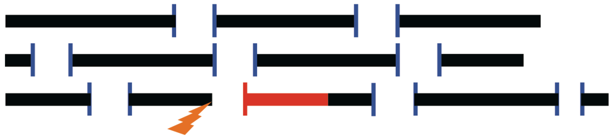
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

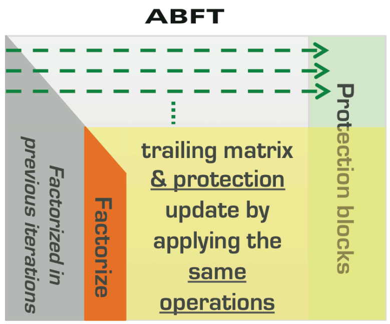
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

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



User Level Failure Mitigation: a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures

What is the status of FT in MPI 3.0?

- Total denial
  - "After an error is detected, the state of MPI is undefined. An MPI implementation is free to allow MPI to continue after an error but is not required to do so."
- Two forms of management
  - Return codes: all MPI functions return either MPI_SUCCESS or a specific error code related to the error class encountered (eg MPI_ERR_ARG)
  - Error handlers: a callback automatically triggered by the MPI implementation before returning from an MPI function.

Error Handlers

- Can be attached to all objects allowing data transfers: communicators, windows and files
- Allow for minimalistic error recovery: the standard suggests only non-MPI related actions
- All newly created MPI objects inherit the error handler from their parent
  - A global error handler can be specified by associating an error handler to MPI_COMM_WORLD right after MPI_Init

```
typedef void MPI_Comm_errhandler_function
(MPI_Comm *, int *, ...);
```

Summary of existing functions

- MPI_Comm_create_errhandler(errh, errhandler_fct)
  - Declare an error handler with the MPI library
- MPI_Comm_set_errhandler(comm, errh)
  - Attach a declared error handler to a communicator
  - Newly created communicators inherits the error handler that is associated with their parent
  - Predefined error handlers:
    - MPI_ERRORS_ARE_FATAL (default)
    - MPI_ERRORS_RETURN

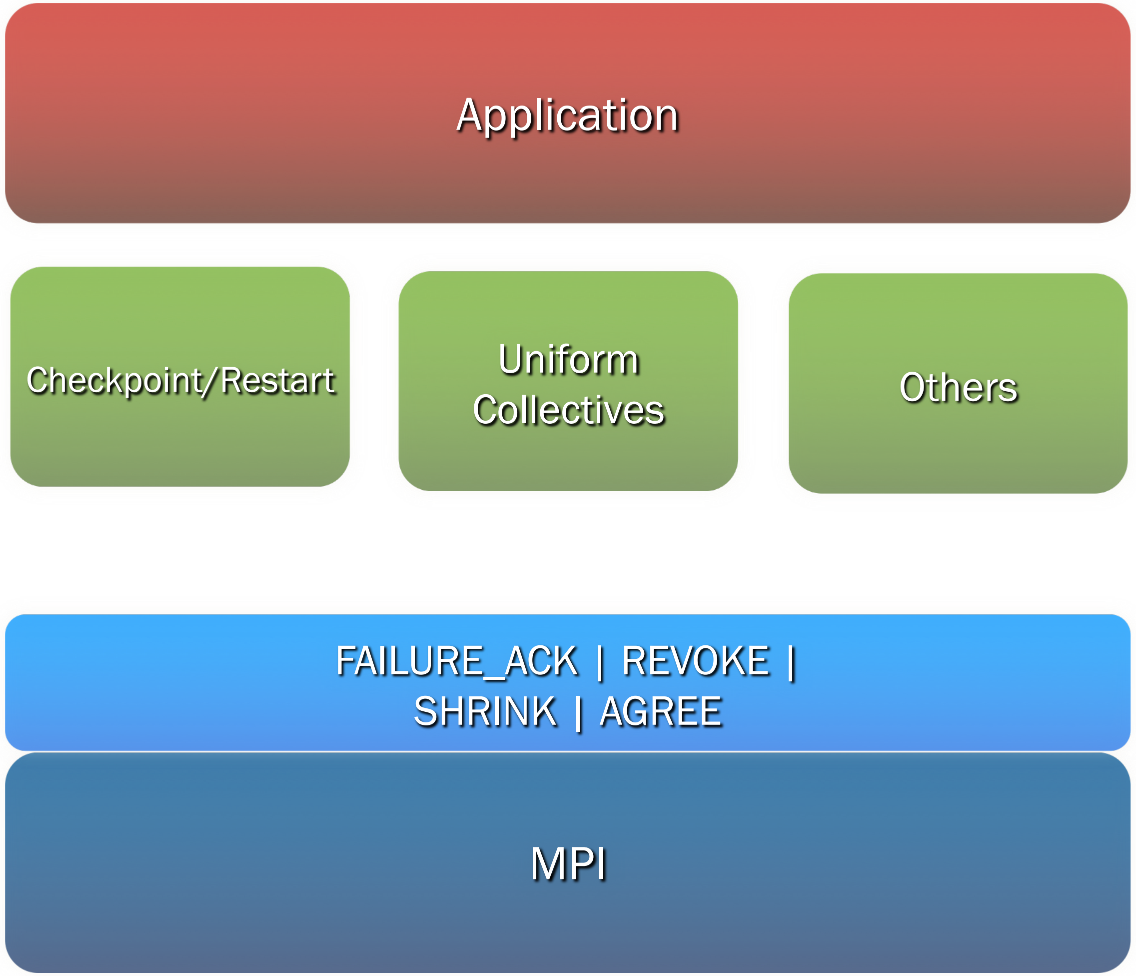
What about MPI 4.0?

- Voted-in tickets in MPI-4 allow some crude error management
- Localize error impact of MPI operations.
  - *Non-communicator operations* (i.e., MPI_ALLOC_MEM) will now raise an error on **MPI_COMM_SELF**, not MPI_COMM_WORLD
- New MPI Error Handler - MPI_ERRORS_ABORT
- Add MPI_ERR_PROC_ABORTED.
- Specify that MPI should avoid fatal errors when the user doesn't use MPI_ERRORS_ARE_FATAL. *By default, an error does not mean that MPI is 'broken'*, and users may interpret the error code and try again (or something else).
- Allow the user to specify the default error handler at `mpiexec` time (i.e., select ERROR_RETURNS or ERRORS_ARE_FATAL before/during MPI_INIT)
- Use some of MPI facilities (with error handling) before MPI_INIT (i.e., MPI_INFO)

Good foundation for full fault management, but not quite complete yet

Requirements for MPI standardization of FT

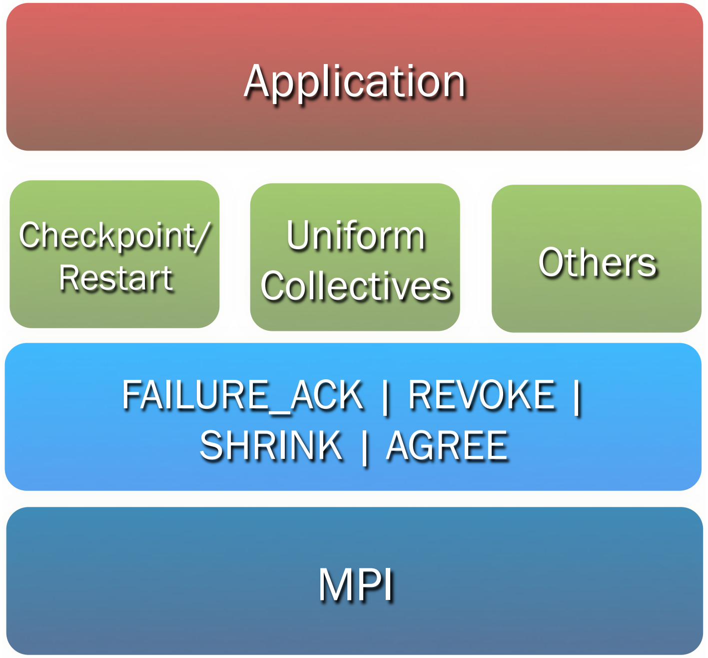
- Expressive, simple to use
  - Support legacy code, backward compatible
  - Enable users to port their code simply
  - Support a variety of FT models and approaches
- Minimal (ideally **zero**) **impact** on failure free performance
  - No global knowledge of failures
  - No supplementary communications to maintain global state
  - Realistic memory requirements
- Simple to implement
  - Minimal (or **zero**) **changes to existing functions**
  - Limited number of new functions
  - Consider thread safety when designing the API



Minimal Feature Set for a Resilient MPI

- Failure Notification
- Error Propagation
- Error Recovery

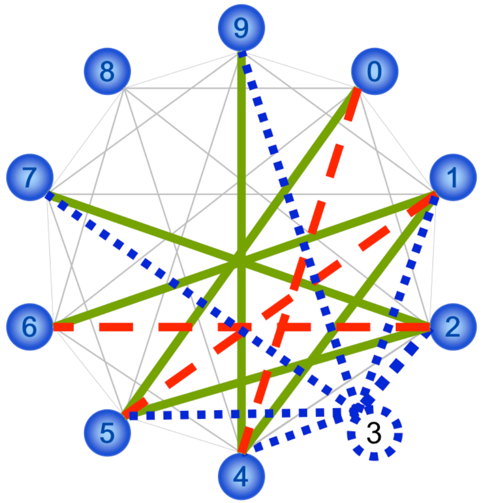
Not all recovery strategies require all of these features, that's why the interface splits notification, propagation and recovery.



ULFM is not a recovery strategy, but a minimalistic set of building blocks for implementing complex recovery strategies.

Failure Notification

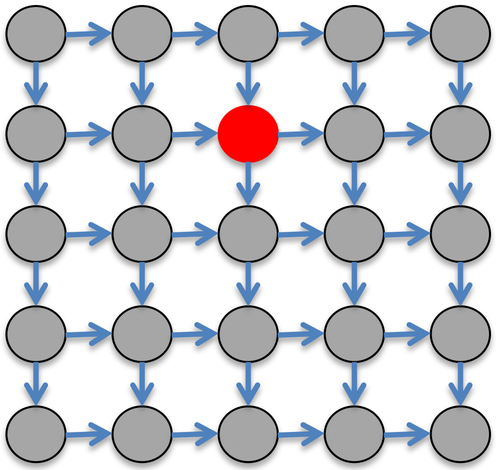
- MPI stands for scalable parallel applications it would be unreasonable to expect full connectivity between all peers
- The failure detection and notification should have a neighboring scope: only processes involved in a communication with the failed process might detect the failure



- But at least one neighbor should be informed about a failure
- MPI_Comm_free must free "broken" communicators and MPI_Finalize must complete despite failures.

Error Propagation

- What is the scope of a failure? Who should be notified about?
- ULFM approach: offer flexibility to allow the library/application to design the scope of a failure, and to limit the scope of a failure to only the needed participants
  - eg. What is the difference between a Master/Worker and a tightly coupled application ?
  - In a 2d mesh application how many nodes should be informed about a failure?

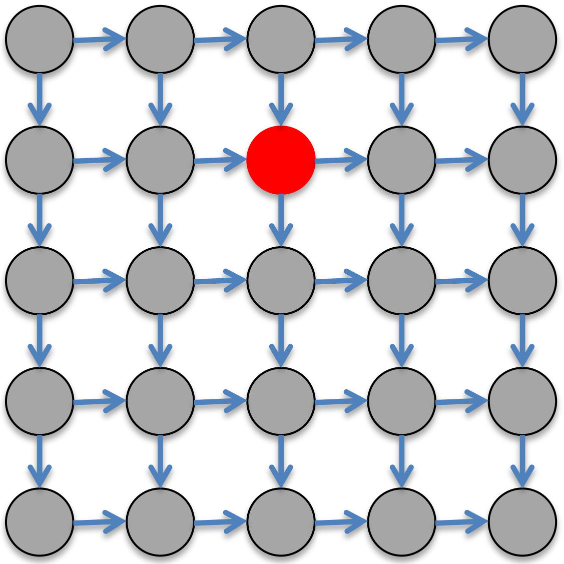


Error Recovery

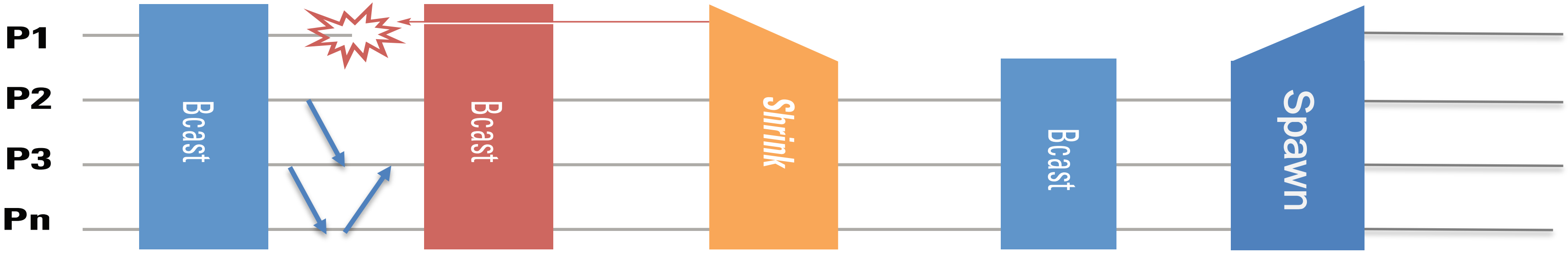
- What is the right recovery strategy?
- Keep going with the remaining processes?
- Shrink the living processes to form a new consistent communicator?
- Spawn new processes to take the place of the failed ones?
- Who should be in charge of defining this survival strategy? What would be the application feedback?

Problem statement

What is the scope of a failure?
Who should be notified about?
What actions should be taken?



- Some applications can continue w/o recovery
- Some applications are maleable
  - Shrink creates a new, smaller communicator on which collectives work
- Some applications are *not* maleable
  - Spawn can recreate a "same size" communicator
  - It is easy to reorder the ranks according to the original ordering
  - Pre-made code snippets available



- Failure Notification
- Error Propagation
- Error Recovery
- Respawn of nodes
- Dataset restoration

Not all recovery strategies require all of these features, that's why the interface should split notification, propagation and recovery.

Summary of new functions

- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

Notification

- MPI_Comm_revoke(comm)
  - Non-collective collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

Propagation

- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return core

Recovery

Integration with existing mechanisms

- New error codes to deal with failures
  - MPI_ERROR_PROC_FAILED: report that the operation discovered a newly dead process. Returned from all blocking function, and all completion functions.
  - MPI_ERROR_PROC_FAILED_PENDING: report that a non-blocking MPI_ANY_SOURCE potential sender has been discovered dead.
  - MPI_ERROR_REVOKED: a communicator has been declared improper for further communications. All future communications on this communicator will raise the same error code, with the exception of a handful of recovery functions
- Is that all?
  - Matching order (MPI_ANY_SOURCE), collective communications

Part rationale, part examples

ULFM MPI API, CONTINUING THROUGH ERRORS

Summary of new functions

- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

- MPI_Comm_revoke(comm)
  - Non-collective collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return core

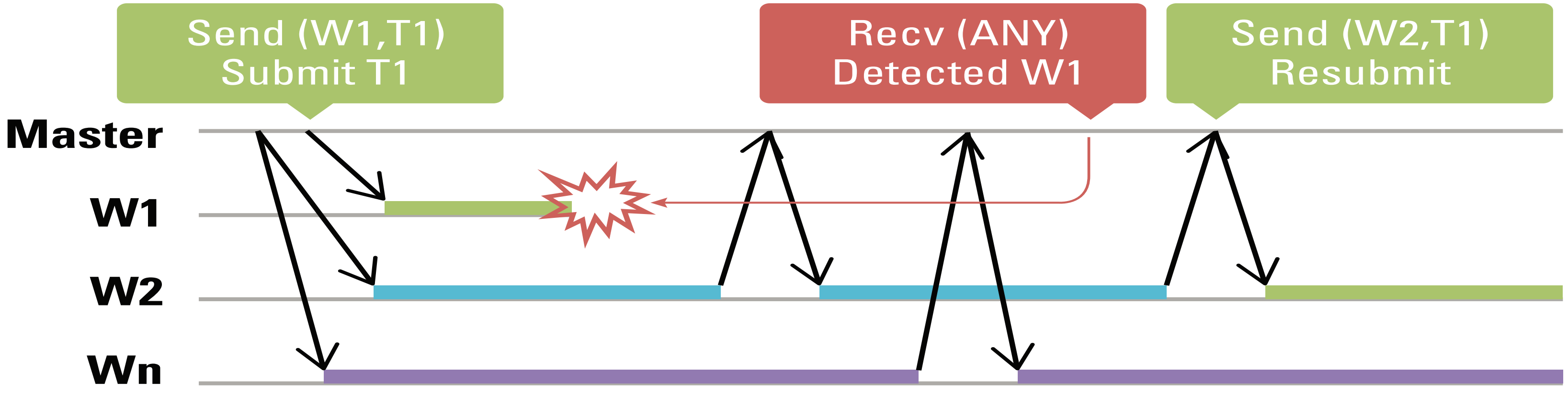
Notification

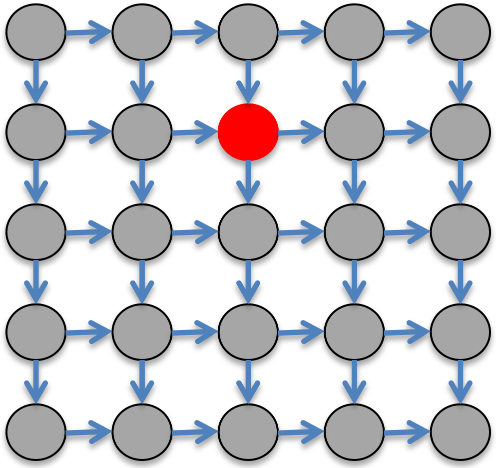
Propagation

Recovery

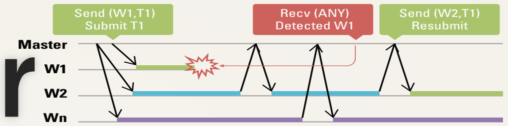
Continuing through errors

- Error notifications do not break MPI
  - App can continue to communicate on the communicator
  - More errors may be raised if the op cannot complete (typically, most collective ops are expected to fail), but p2p between non-failed processes works
- In this Master-Worker example, we can continue w/o recovery!
  - Master sees a worker failed
  - Resubmit the lost work unit onto another worker
  - Quietly continues
- Same story with *mendable* Stencil pattern!
  - Exchange with next neighbor in the same direction instead





Full application: Master Worker



- "bagoftasks-noft" application computes the value of Pi
- main.f: basic program structure
- master_gen.f: master code
  - Master will submit "slicestodo" pieces of work to the workers total
  - First send 1 piece of work to each workers (line 57)
  - Receive results from workers (line 74)
  - If work remains to do, submit another round of work to the workers (line 94)
- slave_gen.f: worker code
  - Receive work from the master (line 44)
  - Compute part of the Pi formula (line 53)
  - Inject a failure (line 70)
  - Return the result to the master (line 83)
  - Verify if the master sent the "finish" token (line 120)
- errh_blank.f: error management
  - Only a skeleton, not compiled or invoked in the provided version

We are going to make this code fault tolerant!

Bye bye, world

See 00.noft.c

```c
int main(int argc, char *argv[])
{
    int rank, size;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d\n", rank, size);

    MPI_Finalize();
}
```

Injecting a failure at the highest rank processor

- I run this program with a FT MPI, and ......
- What do we need to do to make it fault tolerant?

See q01.err_returns.c

Bye bye, world, but orderly

See 01.err_returns.c

```c
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                            MPI_ERRORS_RETURN);

    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n",
           rank, size, errstr);

    MPI_Finalize();
}
```

We can get a nice error string

Errors are not fatal anymore: return an error code instead

collect the error code in rc

All non-faulty processes survive and print the success or error, as reported from MPI_Barrier

- Using only MPI-2 at the moment ☺

Handling errors separately

See q02.err_handler.c

```
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
…
    char errstr[MPI_MAX_ERROR_STRING];
…
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
           rank, size, errstr);
}

int main(int argc, char *argv[]) {
…
    MPI_Errhandler errh;
…
    MPI_Comm_create_errhandler(verbose_errhandler,
                               &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                            errh);
…
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
```

We can pack all error management in an “error handler”

Create an “errhandler” object from the C function, and attach it to the communicator

- Still using only MPI-2 ☺

Handling errors separately

See q02.err_handler.c

```
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
…
    char errstr[MPI_MAX_ERROR_STRING];
…
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
           rank, size, errstr);
}

int main(int argc, char *argv[]) {
…
    MPI_Errhandler errh;
…
    MPI_Comm_create_errhandler(verbose_errhandler,
                               &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                            errh);
…
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
```

No need to collect rc anymore ☺

- Still using only MPI-2 ☺

What caused the error?

See 02.err_hander.c

```
#include <mpi.h>
#include <mpi-ext.h>
…
static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
    MPI_Comm comm = *pcomm;
    int err = *perr;
…
    int …, eclass;
…
    MPI_Error_class(err, &eclass);
    if( MPIX_ERR_PROC_FAILED != eclass ) {
        MPI_Abort(comm, err);
    }
…
```

ULFM is an extension to the MPI standard

This is an “MPI error code”

Convert the “error code” to an “MPI error class”

MPIX_ERR_PROC_FAILED: a process failed, we can deal with it.
Something else: ULFM MPI return the error but it still may be impossible to recover; in this app, we abort when that happens

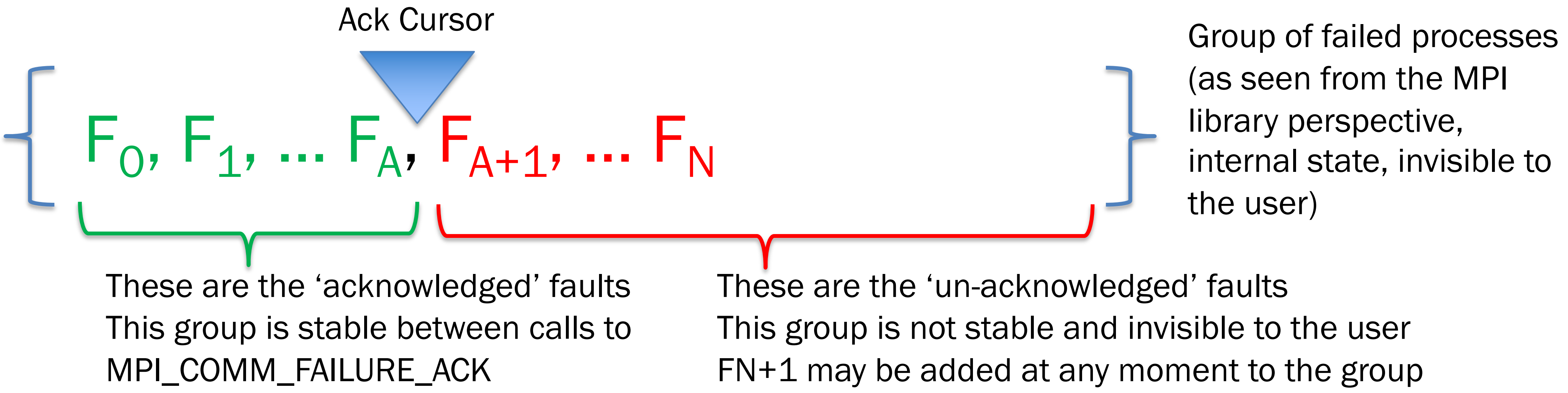
- ULFM defines 3 new error classes:
  - MPI_ERR_PROC_FAILED
  - MPI_ERR_PROC_FAILED_PENDING
  - MPI_ERR_REVOKED
  - After these errors, MPI can be repaired
- All other errors still have MPI-2 semantic
  - May or may not be able to continue after it has been reported

Who caused the error?

- Discovery of failures is *local* (different processes may know of different failures)
- MPI_COMM_FAILURE_ACK(comm)
  - This local operation gives the users a way to acknowledge all locally notified failures on comm. After the call, unmatched MPI_ANY_SOURCE receive operations proceed without further raising MPI_ERR_PROC_FAILED_PENDING due to those acknowledged failures.
- MPI_COMM_FAILURE_GET_ACKED(comm, &grp)
  - This local operation returns the group *grp* of processes, from the communicator comm, that have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK.
- Employing the combination ack/get_acked, a process can obtain the list of all failed ranks (as seen from its local perspective)

MPI_Comm_failure_get_acked

- Local operation returning the group of failed processes in the associated communicator that have been locally acknowledged
- Hint: All calls to MPI_Comm_failure_get_acked between a set of MPI_Comm_failure_ack return the same set of failed processes



MPI_Comm_failure_get_acked

- Local operation returning the group of failed processes in the associated communicator that have been locally acknowledged
- Hint: All calls to MPI_Comm_failure_get_acked between a set of MPI_Comm_failure_ack return the same set of failed processes

A call to MPI_COMM_FAILURE_ACK moves the cursor

$\{ F_0, F_1, \ldots F_A, F_{A+1}, \ldots F_N, F_{N+1}, F_{N+2} \}$

Group of failed processes (as seen from the MPI library perspective)

These are the 'acknowledged' faults
This group is stable between calls to MPI_COMM_FAILURE_ACK

These are the 'un-acknowledged' faults
This group is not stable and invisible to the user
FN+1 may be added at any moment to the group

Who caused the error

Still in 02.err_handler.c

```
…
static void verbose_errhandler(MPI_Comm* pcomm, int* perr,
...) {
    MPI_Comm comm = *pcomm;
…
    MPIX_Comm_failure_ack(comm);
    MPIX_Comm_failure_get_acked(comm, &group_f);
    MPI_Group_size(group_f, &nf);
    MPI_Error_string(err, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. %d found
dead: { ",
           rank, size, errstr, nf);


}
```

Move the “mark” in the known failures list

Get the group of marked failed processes

Who caused the error

Still in 02.err_handler.c

```
…
static void verbose_errhandler(MPI_Comm* pcomm, int* perr,
...) {
    MPI_Comm comm = *pcomm;
…
    MPIX_Comm_failure_ack(comm);
    MPIX_Comm_failure_get_acked(comm, &group_f);
    MPI_Group_size(group_f, &nf);
    MPI_Error_string(err, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. %d found
dead: { ",
           rank, size, errstr, nf);

    ranks_gf = (int*)malloc(nf * sizeof(int));
    ranks_gc = (int*)malloc(nf * sizeof(int));
    MPI_Comm_group(comm, &group_c);
    for(i = 0; i < nf; i++)
        ranks_gf[i] = i;
    MPI_Group_translate_ranks(group_f, nf, ranks_gf,
                              group_c, ranks_gc);
    for(i = 0; i < nf; i++)
        printf("%d ", ranks_gc[i]);
    printf("}\n");
}
```

Move the “mark” in the known failures list

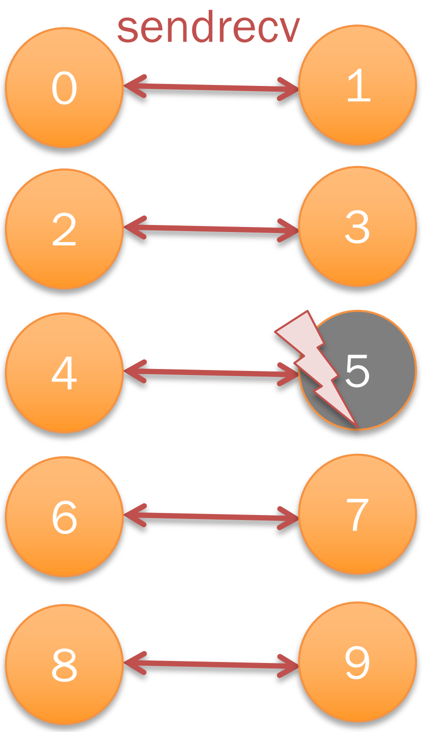
Get the group of marked failed processes

Translate the failed group member’s ranks, in comm

Insulation from irrelevant failures

See 03.undisturbed.c

```
    double myvalue, hisvalue=NAN;
…
    myvalue = rank/(double)size;
    if( 0 == rank%2 )
        peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;
    else
        peer = rank-1;

    if( rank == (size/2) ) raise(SIGKILL);
    /* exchange a value between a pair of two consecutive
     * odd and even ranks; not communicating with anybody
     * else. */
    MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
                 &hisvalue, 1, MPI_DOUBLE, peer, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if( peer != MPI_PROC_NULL)
        printf("Rank %d / %d: value from %d is %g\n",
               rank, size, peer, hisvalue);
```

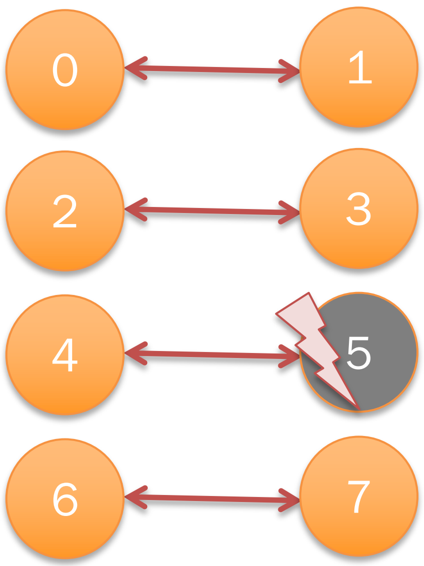


What happens?

Insulation from irrelevant failures

See 03.undisturbed.c

```
    double myvalue, hisvalue=NAN;
…
    myvalue = rank/(double)size;
    if( 0 == rank%2 )
        peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;
    else
        peer = rank-1;

    if( rank == (size/2) ) raise(SIGKILL);
    /* exchange a value between a pair of two consecutive
     * odd and even ranks; not communicating with anybody
     * else. */
    MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
                 &hisvalue, 1, MPI_DOUBLE, peer, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

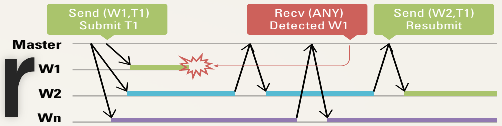



```
bash$ $ULFM_PREFIX/bin/mpirun  -np 10 03.undisturbed
Rank 0 / 10: value from 1 is 0.1
Rank 1 / 10: value from 0 is 0
Rank 3 / 10: value from 2 is 0.2
Rank 2 / 10: value from 3 is 0.3
Rank 6 / 10: value from 7 is 0.7
Rank 7 / 10: value from 6 is 0.6
Rank 9 / 10: value from 8 is 0.8
Rank 8 / 10: value from 9 is 0.9
Rank 4 / 10: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 5 }
Rank 4 / 10: value from 5 is nan
```

Sendrecv between pairs of live processes complete w/o error. Can post more, it will work too!

Sendrecv failed at rank 4 (5 is dead)
Value not updated!

Full application: Master Worker

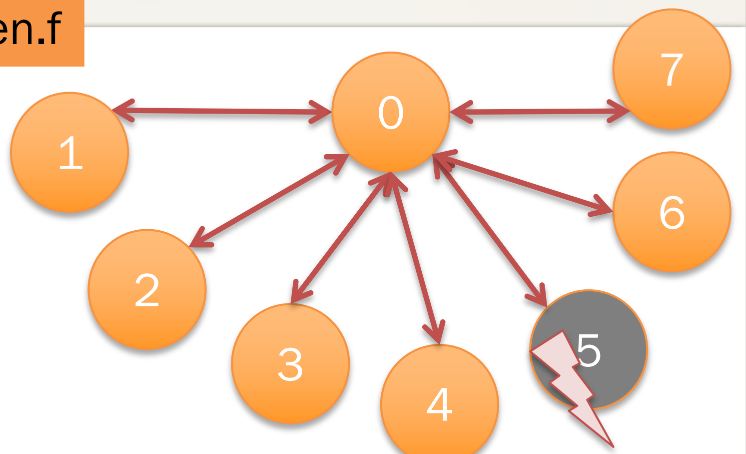


- “bagoftasks-noft” application computes the value of Pi
- main.f: basic program structure
- master_gen.f: master code
  - Set the errhandler
- slave_gen.f: worker code
  - Receive work from the master (line 44)
  - Compute part of the Pi formula (line 53)
  - Inject a failure (line 70)
  - Return the result to the master (line 83)
  - Verify if the master sent the “finish” token (line 120)
- errh_blank.f: error management
  - Only a skeleton, not compiled or invoked in the provided version
  - **Enable the errhandler in the Makefile**

Resilience from the workers perspective

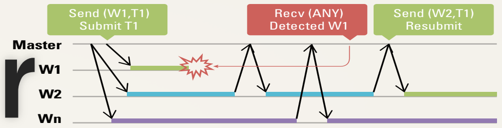
See slave_gen.f

```
C     get work
      ¦ if ( mystate .eq. AVAILABLE ) then
      ¦   ¦call MPI_Recv ( todo, 1, MPI_INTEGER, masterrank, WORK_TAG,
      ¦&        lib_comm, status, rc )
      ¦   ¦if (rc.ne.0) write(*,*) "[", myrank, "] ERRORCODE",rc
      ¦   ¦write (*,*) "RECV WORK: [", myrank, "] work [", todo,"]"
      ¦   ¦call slave_advance_state (mystate, todo)
      ¦ end if
```



What does that mean for the required error handling at our workers?

Full application: Master Worker



- "bagoftasks-noft" application computes the value of Pi
- main.f: basic program structure
- master_gen.f: master code
  - Set the errhandler
- slave_gen.f: worker code
  - Receive work from the master (line 44)
  - Compute part of the Pi formula (line 53)
  - Inject a failure (line 70)
  - Return the result to the master (line 83)
  - Verify if the master sent the "finish" token (line 120)
- errh_blank.f: error management
  - ~~Enable the errhandler in the Makefile~~
  - **Identify failed processes and mark them as failed in application structures**

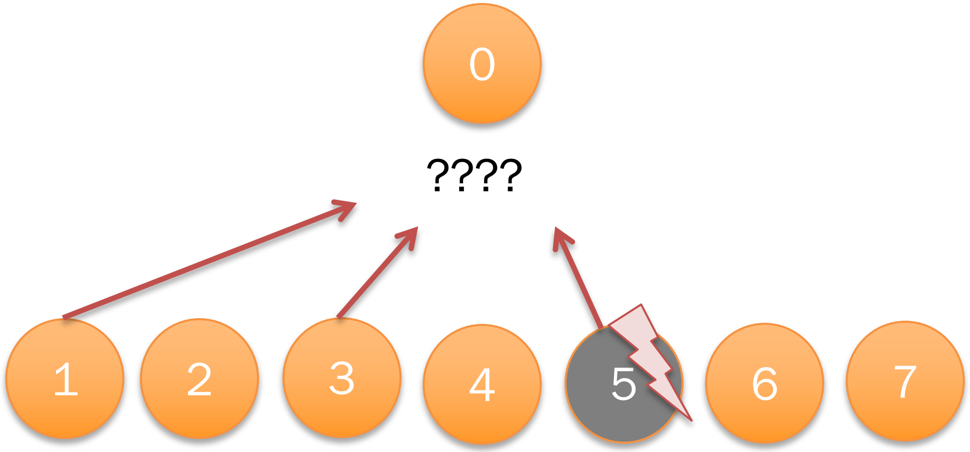
Dealing with MPI_ANY_SOURCE

See 08.err_any_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            rc = MPI_Recv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
            if( MPI_SUCCESS == rc ) {
                printf("Received from %d during recv %d\n", unused, i);
                i++;
            }
            else {
```

Assume a process dies before sending the message

No specified source



Dealing with MPI_ANY_SOURCE

See 08.err_any_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            rc = MPI_Recv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
            if( MPI_SUCCESS == rc ) {
                printf("Received from %d during recv %d\n", unused, i);
                i++;
            }
            else {
```

Assume a process dies before sending the message

No specified source, the failure detection is homogeneous

MPIX_ERR_PROC_FAILED on every node posting an ANY_SOURCE.

- If the recv uses ANY_SOURCE:
  - Any failure in the comm is potentially a failure of the matching sender!
  - The recv MUST be interrupted
  - If the application knows the receive is safe, it can repost the MPI_RECV operation
- But… it will return an error again, wouldn't it?

MPI_Comm_failure_ack

- Local operations that acknowledge all locally notified failures
  - Updates the group returned by MPI_COMM_FAILURE_GET_ACKED
- Unmatched **MPI_ANY_SOURCE** that would have raised MPI_ERR_PROC_FAILED or MPI_ERR_PROC_FAILED_PENDING **proceed without further exceptions** regarding the acknowledged failures.
- MPI_COMM_AGREE do not raise MPI_ERR_PROC_FAILED due to acknowledged failures
  - No impact on other MPI calls especially not on collective communications

Dealing with MPI_ANY_SOURCE

See 08.err_any_source.c

```
        for(i = 1; i < size-nf; ) {
            rc = MPI_Recv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
            if( MPI_SUCCESS == rc ) {
                printf("Received from %d during recv %d\n", unused, i);
                i++;
            }
            else {
                int eclass;
                MPI_Group group_f;
                MPI_Error_class(rc, &eclass);
                if( MPIX_ERR_PROC_FAILED != eclass ) {
                    MPI_Abort(MPI_COMM_WORLD, rc);
                }
                MPIX_Comm_failure_ack(MPI_COMM_WORLD);
                MPIX_Comm_failure_get_acked(MPI_COMM_WORLD, &group_f);
                MPI_Group_size(group_f, &nf);
                MPI_Group_free(&group_f);
                printf("Failures detected! %d found so far\n", nf);
            }
        }
```

MPIX_ERR_PROC_FAILED on every node posting an ANY_SOURCE.

Resumes normal ANY_SOURCE operations

Non-Blocking MPI_ANY_SOURCE

See 08.2.err_iany_source.c

```
if( 0 != rank ) {
    MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
else {
    printf("Recv(ANY) test\n");
    for(i = 1; i < size-nf; ) {
        MPI_Request req;
        rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD
&req);
…
        rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
  - *The recv MUST be interrupted*
  - Interrupting non-blocking ANY_SOURCE could change matching order…
- New error code MPIX_ERR_PROC_FAILED_PENDING: the operation is interrupted by a process failure, **but is still *pending***
- If the application knows the receive is safe, and the matching order respected, the pending operation can be waited upon (otherwise MPI_Cancel)

Why ERR_PROC_FAILED_PENDING

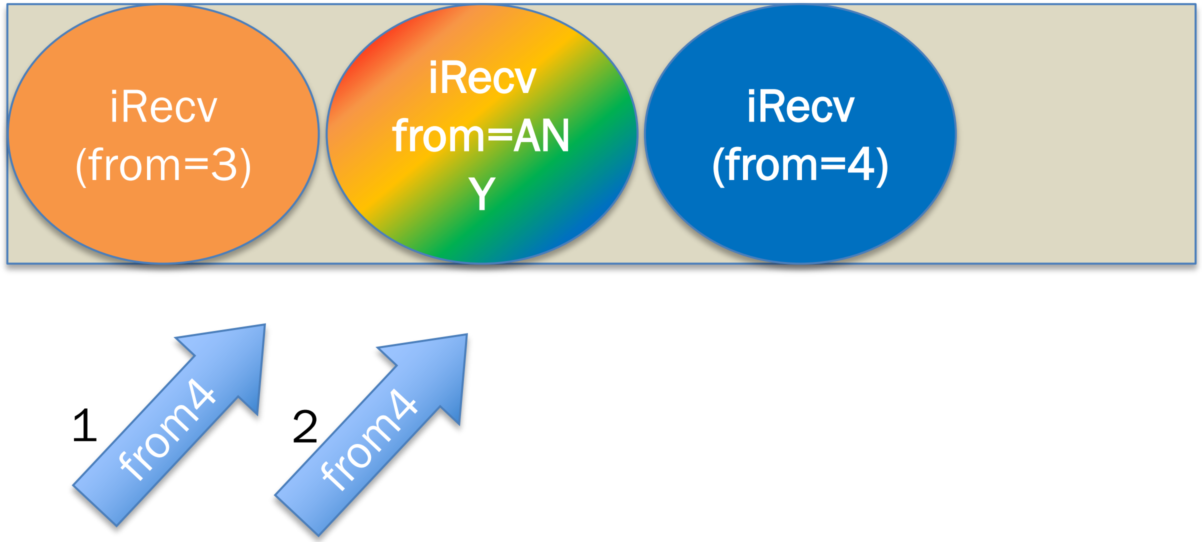
See 08.2.err_iany_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            MPI_Request req;
            rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD
&req);
…
            rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
- *The recv MUST be interrupted*
- Interrupting non-blocking ANY_SOURCE could change matching order…



Why ERR_PROC_FAILED_PENDING

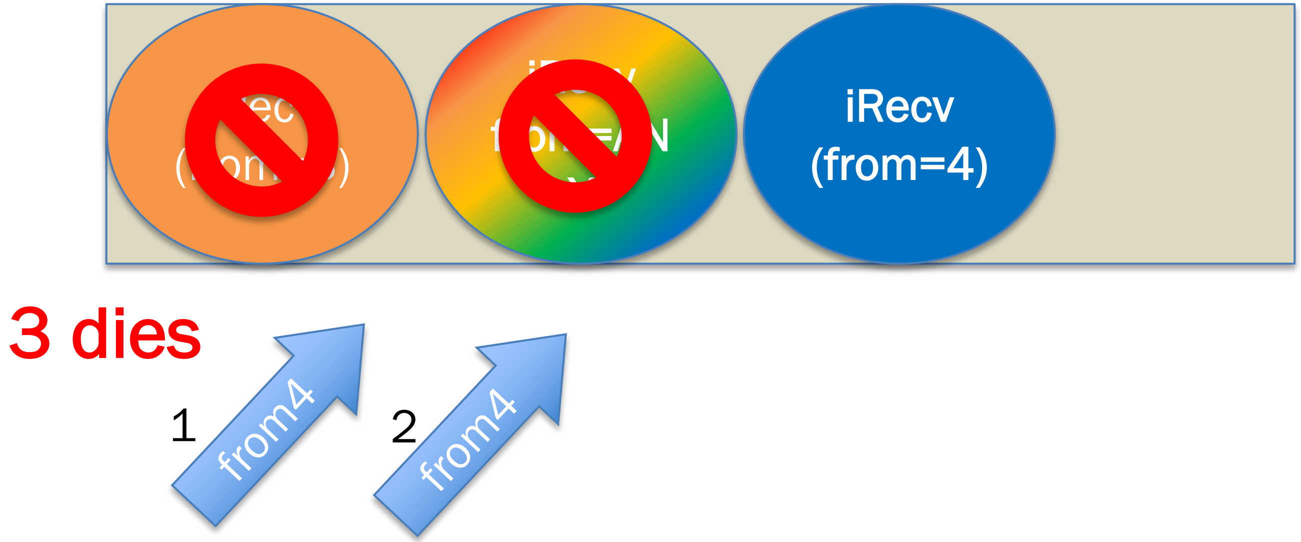
See 08.2.err_iany_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            MPI_Request req;
            rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
&req);
…
            rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
- *The recv MUST be interrupted*
- Interrupting non-blocking ANY_SOURCE could change matching order…



Why ERR_PROC_FAILED_PENDING

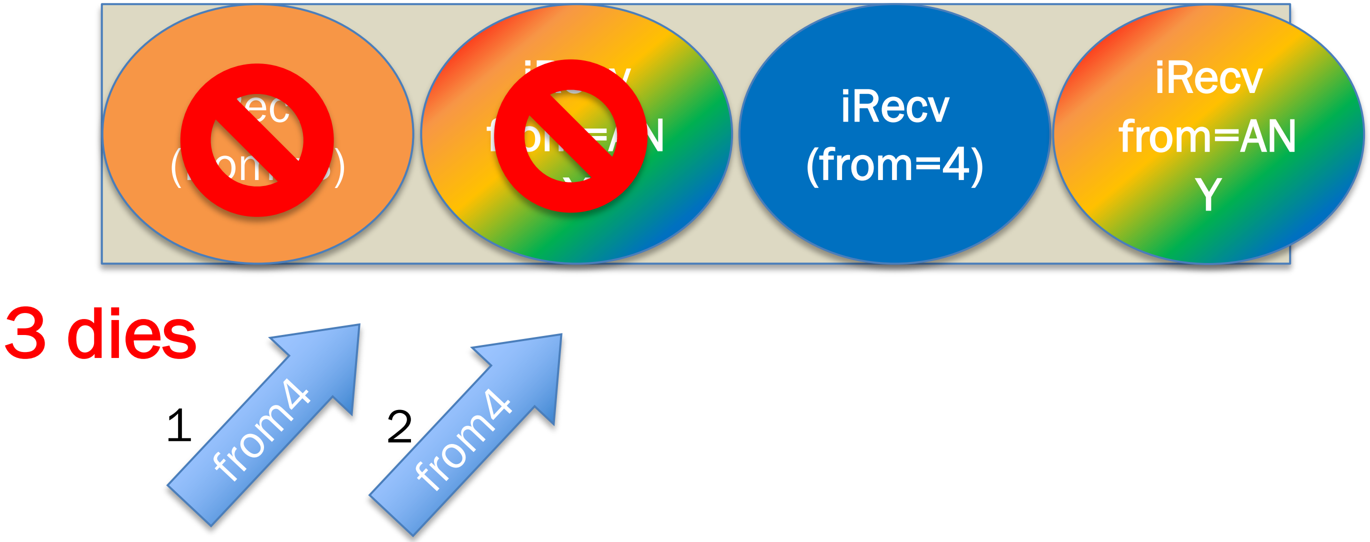
See 08.2.err_iany_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            MPI_Request req;
            rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
&req);
…
            rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
- *The recv MUST be interrupted*
- Interrupting non-blocking ANY_SOURCE could change matching order…



Why ERR_PROC_FAILED_PENDING

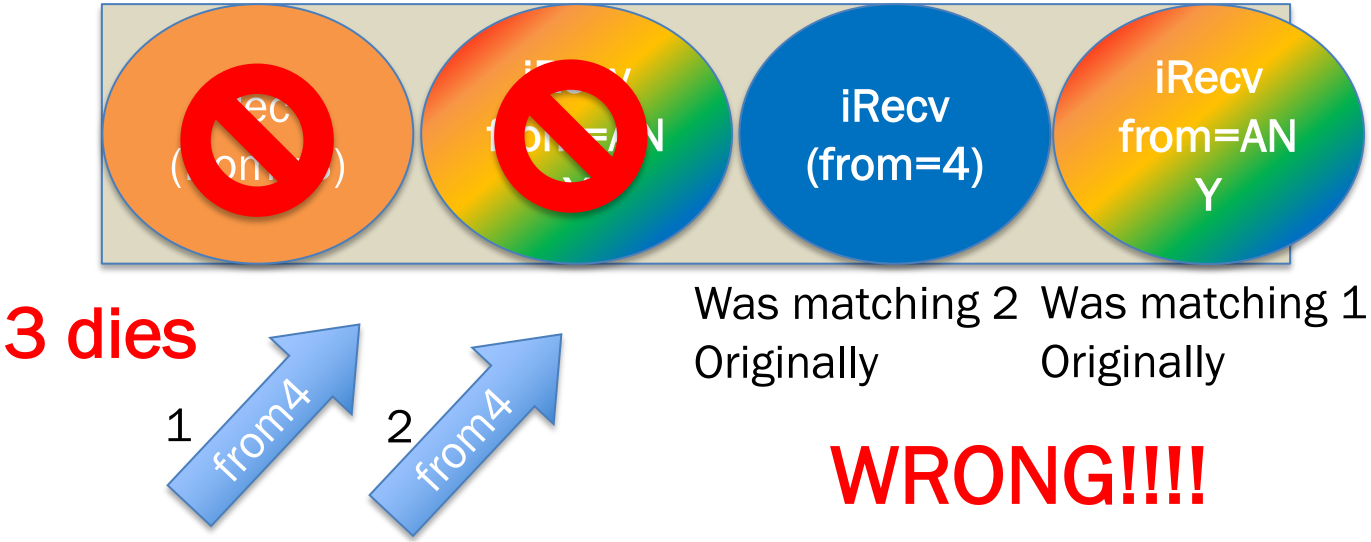
See 08.2.err_iany_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            MPI_Request req;
            rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
&req);
…
            rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
- *The recv MUST be interrupted*
- Interrupting non-blocking ANY_SOURCE could change matching order…



Why ERR_PROC_FAILED_PENDING

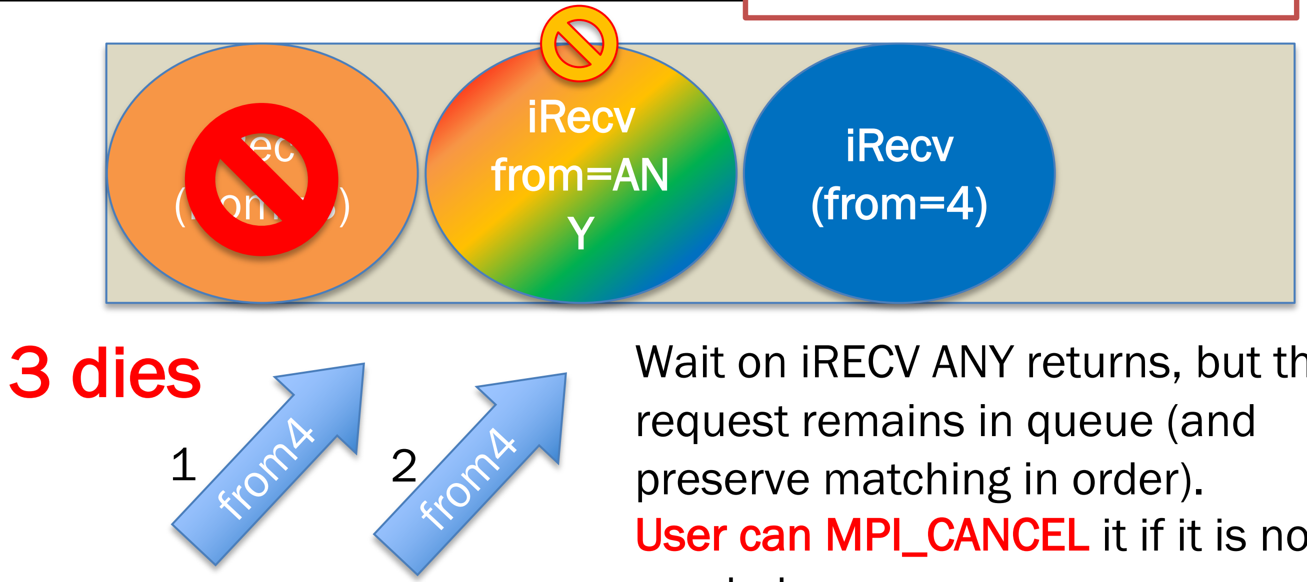
See 08.2.err_iany_source.c

```
    if( 0 != rank ) {
        MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else {
        printf("Recv(ANY) test\n");
        for(i = 1; i < size-nf; ) {
            MPI_Request req;
            rc = MPI_Irecv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
&req);
…
            rc = MPI_Wait(&req, &status);
```

Assume a process dies before sending the message

No specified source, non-blocking reception

- If the non-blocking recv uses ANY_SOURCE:
  - *Any failure in the comm is potentially a failure of the matching sender!*
- *The recv MUST be interrupted*
- Interrupting non-blocking ANY_SOURCE could change matching order…



Wait on iRECV ANY returns, but the request remains in queue (and preserve matching in order). **User can MPI_CANCEL** it if it is not needed anymore.

What did we learn?

- You can write an application that survives process failures
- You can use MPI Error handlers to capture the errors
- MPI Communication can continue between non-failed processes
- You can obtain the list (as per the rank local view) of failed processes
- You can restore ANY_SOURCE receives with local-only operations
- You have written an application that continues its computation when processes fail!

Propagating an error under the applications’ control

STOP! WE NEED TO FIX THIS TOGETHER

Summary of new functions

- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

Notification

- MPI_Comm_revoke(comm)
  - Non-collective collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

Propagation

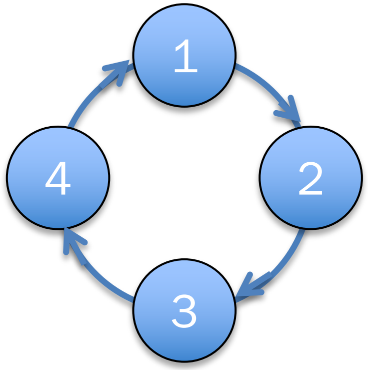
- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return core

Recovery

Regrouping for Plan B

See q04.if_error.c

```
/* Assign left and right neighbors to be rank-1 and rank+1
 * in a ring modulo np */
left   = (np+rank-1)%np;
right  = (np+rank+1)%np;

for( i = 0; i < 10; i++ ) {
…
    /* At every iteration, a process receives from it's 'left' neighbor
     * and sends to 'right' neighbor (ring fashion, modulo np)
     * ... -> 0 -> 1 -> 2 -> ... -> np-1 -> 0 ... */
    rc = MPI_Sendrecv( sarray, COUNT, MPI_DOUBLE, right, 0,
                       rarray, COUNT, MPI_DOUBLE, left , 0,
                       fcomm, MPI_STATUS_IGNORE );
…
    if( rc != MPI_SUCCESS ) {
        /* ???>>> Hu ho, this program has a problem here */
        goto cleanup;
    }
```

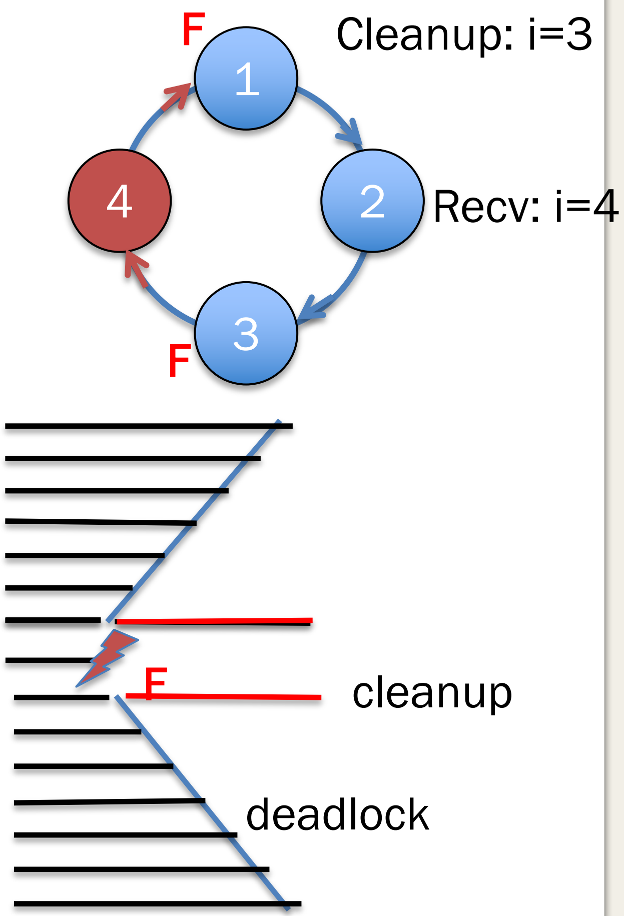


- Run this program. What happens?
- Can you fix it ?

Regrouping for Plan B

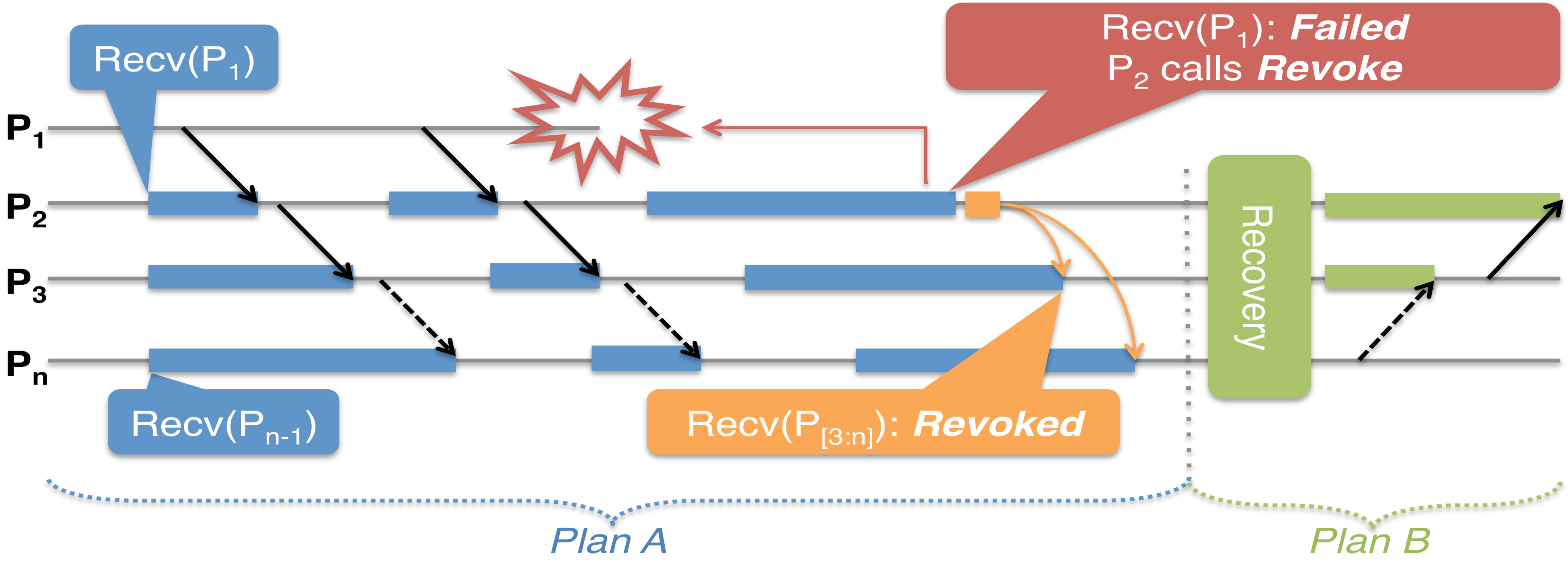
See q04.if_error.c

```
    /* Assign left and right neighbors to be rank-1 and rank+1
     * in a ring modulo np */
    left   = (np+rank-1)%np;
    right  = (np+rank+1)%np;

    for( i = 0; i < 10; i++ ) {
…
        /* At every iteration, a process receives from it's 'left' neighbor
         * and sends to 'right' neighbor (ring fashion, modulo np)
         * ... -> 0 -> 1 -> 2 -> ... -> np-1 -> 0 ... */
        rc = MPI_Sendrecv( sarray, COUNT, MPI_DOUBLE, right, 0,
                           rarray, COUNT, MPI_DOUBLE, left , 0,
                           fcomm, MPI_STATUS_IGNORE );
…
        if( rc != MPI_SUCCESS ) {
            /* ???>>> Hu ho, this program has a problem here */
            goto cleanup;
        }
```



- Run this program. What happens?
- Can you fix it ?

Regrouping for Plan B

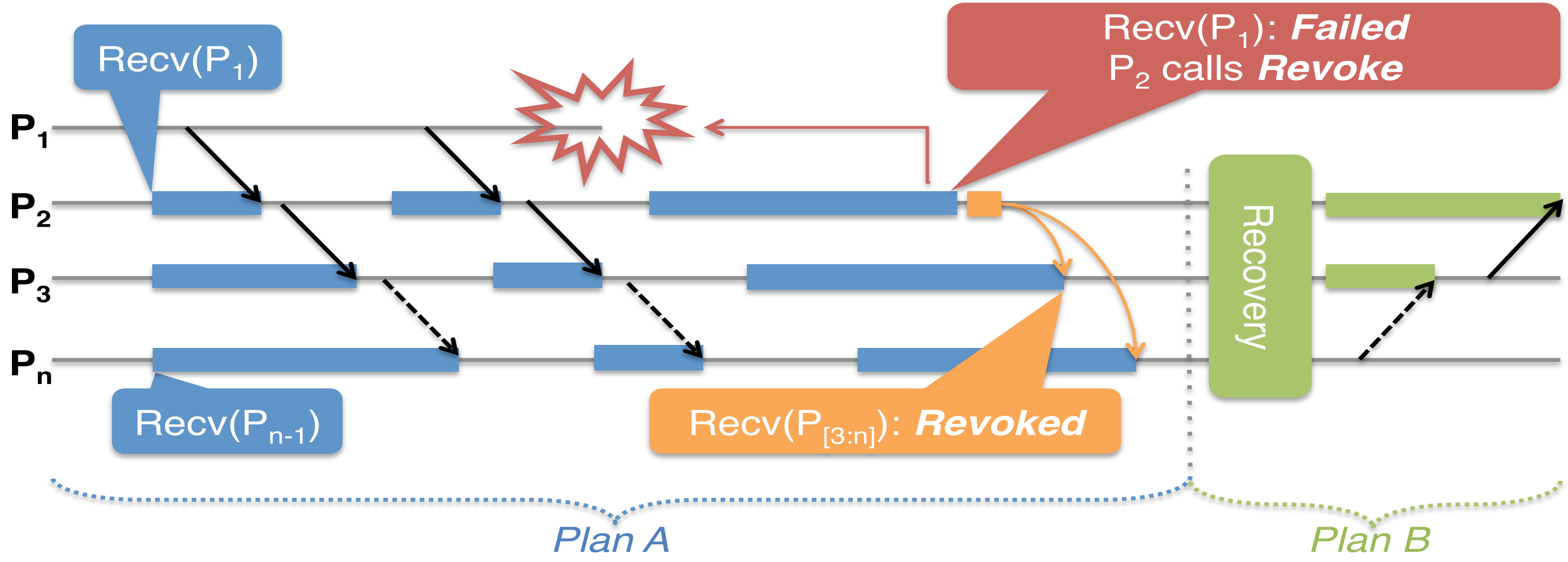


- P1 fails
- P2 raises an error and stop *Plan A* to enter application recovery *Plan B*
- but P3..Pn are stuck in their posted recv
- P2 can unlock them with Revoke ☺
- P3..Pn join P2 in the recovery

MPI_Comm_revoke

- Communicator level failure propagation
- The revocation of a communicator completes all pending local operations
  - A communicator is revoked either after a local MPI_Comm_revoke or any MPI call raise an exception of class MPI_ERR_REVOKED
- Unlike any other concept in MPI it is not a collective call but has a collective scope
- Once a communicator has been revoked all non-local calls are considered local and must complete by raising MPI_ERR_REVOKED
  - Notable exceptions: the recovery functions (agreement and shrink)

Regrouping for Plan B




```
if( rc != MPI_SUCCESS ) {
    /* Ok, some error occurred, force other processes to exit the loop
     * because when I am leaving, I will not match the sendrecv, and
     * that would cause them to deadlock */
    MPIX_Comm_revoke( fcomm );
    goto cleanup;
}
```

See 04.if_error.c

Making order out of chaos

STABILIZING AFTER AN ERROR

Hands On: Fault Tolerant MPI with ULFM

George Bosilca @SC19

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

Summary of new functions

- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

Notification

- MPI_Comm_revoke(comm)
  - Non-collective collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

Propagation

- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return core

Recovery

About non-uniform error reporting

```
35        value = rank/(double)size;
36
37        if( rank == (size/4) ) raise(SIGKILL);
38        MPI_Bcast(&value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
39
40        if( value != 0.0 ) {
41            printf("Rank %d / %d: value from %d is wrong: %g\n",
42                    rank, size, 0, value);
43        }
```

See 05.err_coll.c

Bcast from 0 is disrupted by a failure

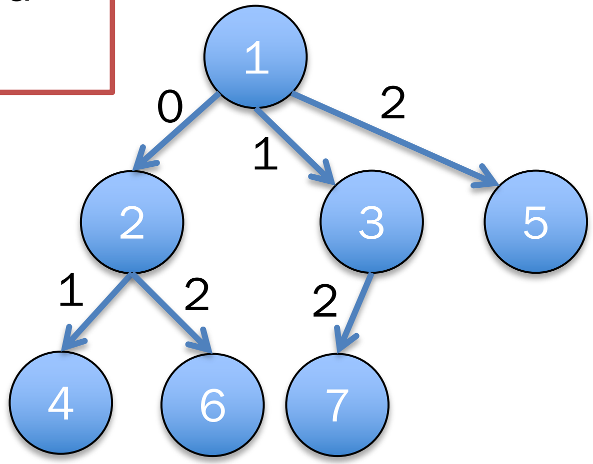
- What processes are going to report an error ?
- Is any process going to display the message line 41 ?
- What if we do an Allreduce instead?

About non-uniform error reporting

See 05.err_coll.c

```
35     value = rank/(double)size;
36
37     if( rank == (size/4) ) raise(SIGKILL);
38     MPI_Bcast(&value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
39
40     if( value != 0.0 ) {
41         printf("Rank %d / %d: value from %d is wrong: %g\n",
42                 rank, size, 0, value);
43     }
```

Bcast from 0 is disrupted by a failure



- Are all processes going to report an error ?
- Is any process going to display the message line 41 ?

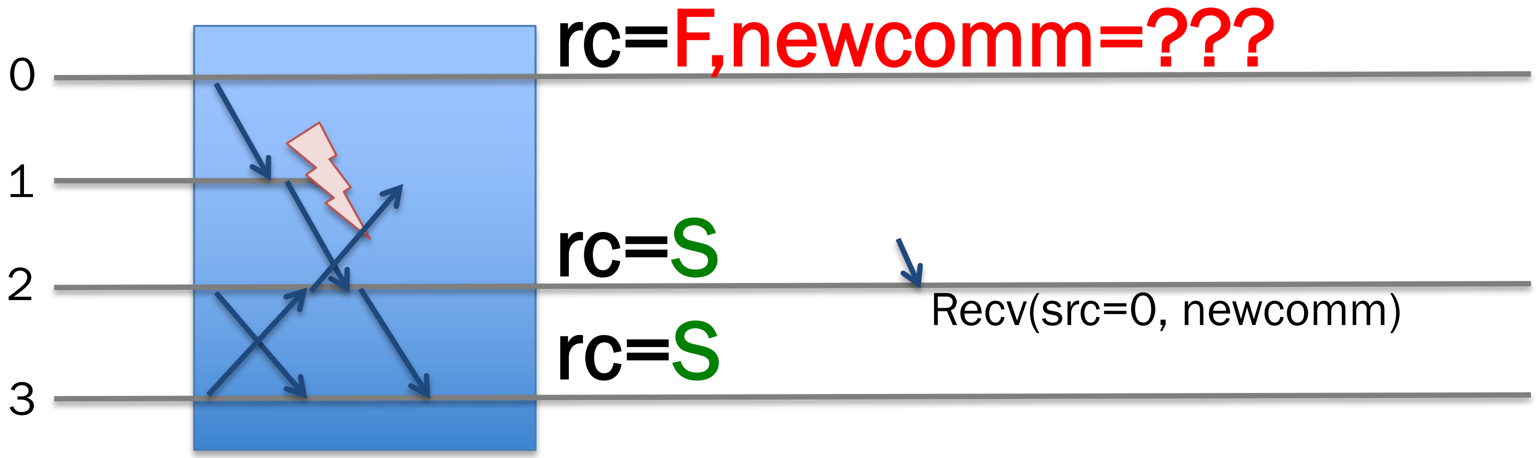
```
bash$ $ULFM_PREFIX/bin/mpirun -np 5 05.err_coll -v
Rank 3 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 1 }
Rank 3 / 5: value from 0 is wrong: 0.6
```

MPI_Bcast internally uses a binomial tree topology 3 (a leaf) was supposed to receive from 1…

0 is the root, it sends to 1, but doesn't see the failure of 1

Bcast failed at rank 3, value has not been updated!

Issue with communicator creation



MPI_Comm_dup w/failure at rank 1 during the operation

- **MPI_Comm_dup** (for example) is a collective
  - Like MPI_Bcast, it may raise an error at some rank and not others
  - When rank 0 sees MPI_ERR_PROC_FAILED, *newcomm* is not created correctly!
  - At the same time, rank 2 creates new*comm* correctly
  - If rank 2 posts an operation with 0, this operation cannot complete (0 cannot post the matching send, it doesn't have the new*comm*)
    - *Deadlock!*

Safe communicator creation

```c
/* Performs a comm_dup, returns uniformly MPIX_ERR_PROC_FAILED or
 * MPI_SUCCESS */
int ft_comm_dup(MPI_Comm comm, MPI_Comm *newcomm) {
    int rc;
    int flag;

    rc = MPI_Comm_dup(comm, newcomm);
    flag = (MPI_SUCCESS==rc);

    if( !flag ) {
        if( rc == MPI_SUCCESS ) {
            MPI_Comm_free(newcomm);
            rc = MPIX_ERR_PROC_FAILED;
        }
    }
    return rc;
}
```

See q06.err_comm_dup.c

MPI_Comm_agree

- Perform a consensus between all living processes in the associated communicator and consistently return a value and an error code to all living processes
- Upon completion all living processes agree to set the output integer value to a bitwise AND operation over all the contributed values
  - Also perform a consensus on the set of known failed processes (!)
  - Failures non acknowledged by all participants keep raising MPI_ERR_PROC_FAILED

Safe communicator creation

```c
/* Performs a comm_dup, returns uniformly MPIX_ERR_PROC_FAILED or
 * MPI_SUCCESS */
int ft_comm_dup(MPI_Comm comm, MPI_Comm *newcomm) {
    int rc;
    int flag;

    rc = MPI_Comm_dup(comm, newcomm);
    flag = (MPI_SUCCESS==rc);
    MPIX_Comm_agree(comm, &flag);
    if( !flag ) {
        if( rc == MPI_SUCCESS ) {
            MPI_Comm_free(newcomm);
            rc = MPIX_ERR_PROC_FAILED;
        }
    }
    return rc;
}
```

See 06.err_comm_dup.c

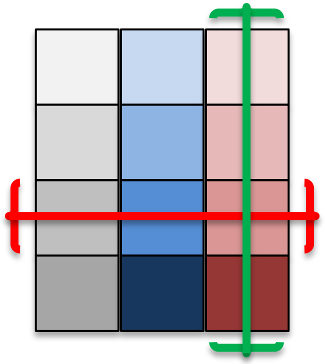
- Solution: MPI_Comm_agree
  - After ft_comm_dup, either all procs have created newcomm, or all procs have returned MPI_ERR_PROC_FAILED
  - Global state is consistent in all cases

Benefit of safety separation

See q07.err_comm_grid2d

```
rc1 = MPI_Comm_split(comm, rank%p, rank, rowcomm);
flag = (MPI_SUCCESS==rc1);
MPIX_Comm_agree(comm, &flag);
if( !flag ) {
    if( rc1 == MPI_SUCCESS ) {
        MPI_Comm_free(rowcomm);
    }
    return MPIX_ERR_PROC_FAILED;
}

rc2 = MPI_Comm_split(comm, rank/p, rank, colcomm);
flag = (MPI_SUCCESS==rc2);
MPIX_Comm_agree(comm, &flag);
if( !flag ) {
    if( rc2 == MPI_SUCCESS ) {
        MPI_Comm_free(colcomm);
    }
    MPI_Comm_free(rowcomm);
    return MPIX_ERR_PROC_FAILED;
}
```

- PxQ 2D process grid
  - A process appears in two communicators
  - A row communicator
  - A column communicator



Benefit of safety separation

See 07.err_comm_grid2d

```
/* Create two communicators, representing a PxP 2D grid of
 * the processes. Either return MPIX_ERR_PROC_FAILED at all ranks,
 * then no communicator has been created, or MPI_SUCCESS and all
 * communicators have been created, at all ranks. */
int ft_comm_grid2d(MPI_Comm comm, int p, MPI_Comm *rowcomm, MPI_Comm *colcomm)
{
…
    rc1 = MPI_Comm_split(comm, rank%p, rank, rowcomm);
    rc2 = MPI_Comm_split(comm, rank/p, rank, colcomm);
    flag = (MPI_SUCCESS==rc1) && (MPI_SUCCESS==rc2);
    MPIX_Comm_agree(comm, &flag);
    if( !flag ) {
        if( rc1 == MPI_SUCCESS ) {
            MPI_Comm_free(rowcomm);
        }
        if( rc2 == MPI_SUCCESS ) {
            MPI_Comm_free(colcomm);
        }
        return MPIX_ERR_PROC_FAILED;
    }
    return MPI_SUCCESS;
}
```

- PxP 2D process grid
  - A process appears in two communicators
  - A row communicator
  - A column communicator
- We Agree only once
  - Better amortization of the cost over multiple operations

Error Insulation

See 09.err_insulation.c

```c
int main(int argc, char *argv[]) {
    MPI_Comm half_comm;
…
    /* Create 2 halfcomms, one for the low ranks, 1 for the high ranks */
    MPI_Comm_split(MPI_COMM_WORLD, (rank<(size/2))? 1: 2, rank, &half_comm);

    if( rank == 0 ) raise(SIGKILL);
    MPI_Barrier(half_comm);

    /* Even when half_comm contains failed processes, we call MPI_Comm_free
     * to give an opportunity for MPI to clean the ressources. */
    MPI_Comm_free(&half_comm);
```

Half_comm inherits the error handler from MPI_COMM_WORLD

Interlude: MPI_Comm_split

- MPI_COMM_SPLIT(comm, color, key, newcomm)
  - Color : control of subset assignment
  - Key : sort key to control rank assignment

rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
color	0	⊥	3	0	3	0	0	5	3	⊥
key	3	1	2	5	1	1	1	2	1	0

3 different colors => 3 communicators

1. {A, D, F, G} with sort keys {3, 5, 1, 1} => {F, G, A, D}
2. {C, E, I} with sort keys {2, 1, 1} => {E, I, C}
3. {H} with sort key {2} => {H}

B and J get MPI_COMM_NULL as they provide an undefined color (MPI_UNDEFINED)

More Insulation

See 09.err_insulation.c

```
int main(int argc, char *argv[]) {
    MPI_Comm half_comm;
…
    /* Create 2 halfcomms, one for the low ranks, 1 for the high ranks */
    MPI_Comm_split(MPI_COMM_WORLD, (rank<(size/2))? 1: 2, rank, &half_comm);

    if( rank == 0 ) raise(SIGKILL);
    MPI_Barrier(half_comm);

    /* Even when half_comm contains failed processes, we call MPI_Comm_free
     * to give an opportunity for MPI to clean the ressources. */
    MPI_Comm_free(&half_comm);
```

Low ranks half_comm:
What will happen?

High ranks half_comm:
What will happen?

More Insulation

See 09.err_insulation.c

```
int main(int argc, char *argv[]) {
    MPI_Comm half_comm;
…
    /* Create 2 halfcomms, one for the low ranks, 1 for the high ranks */
    MPI_Comm_split(MPI_COMM_WORLD, (r                       &half_comm);

    if( rank == 0 ) raise(SIGKILL);
    MPI_Barrier(half_comm);

    /* Even when half_comm contains failed proc                     n_free
     * to give an opportunity for MPI to clean
    MPI_Comm_free(&half_comm);
```

High ranks half_comm has no failure, it works ☺

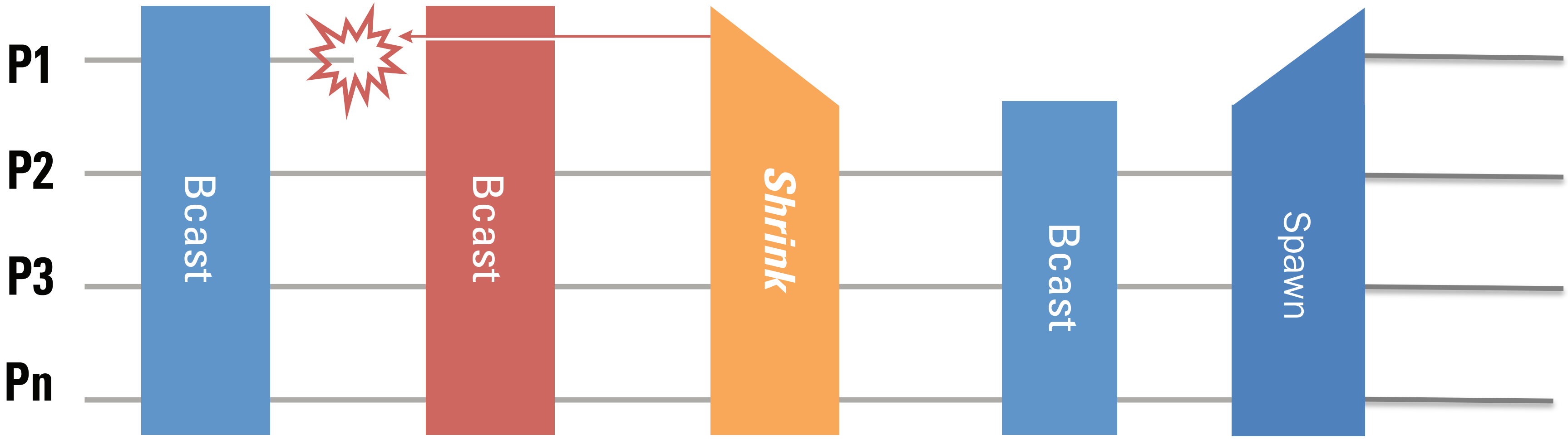
Low ranks half_comm has failed process, we free it anyway


```
$ $ULFM_PREFIX/bin/mpirun -np 10 09.insulation
Rank 1 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead:
{ 0 }
Rank 2 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead:
{ 0 }
Rank 4 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead:
{ 0 }
Rank 3 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead:
{ 0 }
```

Can we fix it? Yes we can!

FIXING THE WORLD

Full capacity recovery



- After a Revoke, our original comm is unusable
- We can Shrink: that create a new comm, but smaller
  - Can be used to do collective and p2p operations, fully functional
- Some application need to restore a world the same size
  - And on top of it, they want the same rank mapping

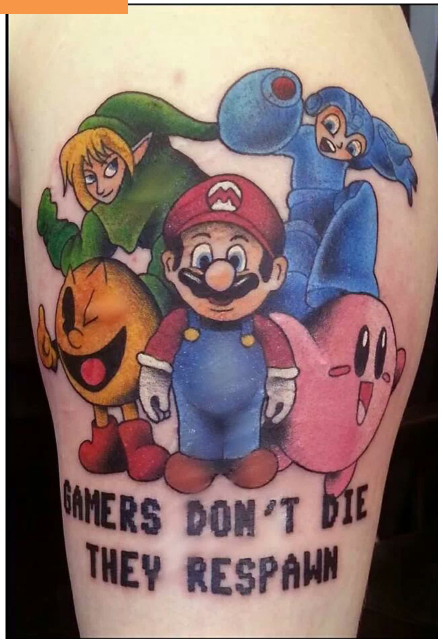
MPI_Comm_shrink

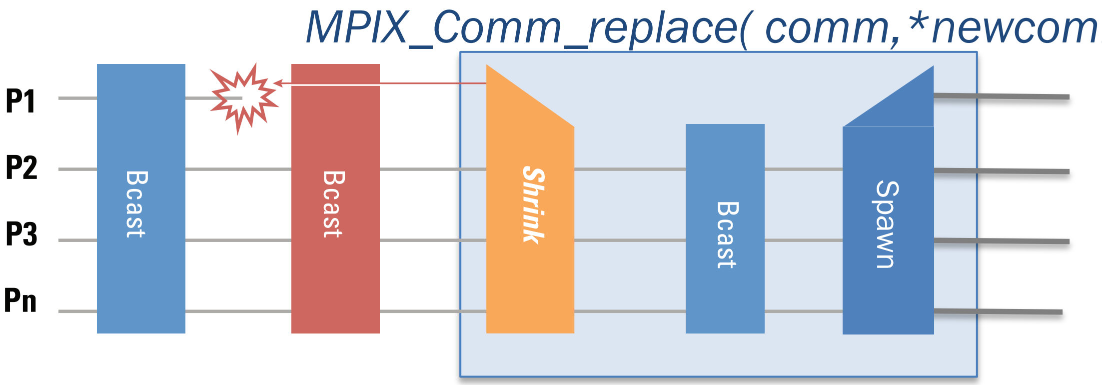
- Creates a new communicator by excluding all known failed processes from the parent communicator
  - It completes an agreement on the parent communicator
  - Work on revoked communicators as a mean to create safe, globally consistent sub-communicators
- Absorbs new failures, it is not allowed to return MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED

Respawning the deads

See 10.respawn

```
int main( int argc, char* argv[] ) {
…
  /* Am I a spare ? */
  MPI_Comm_get_parent( &world );
  if( MPI_COMM_NULL == world ) {
    /* First run: Let's create an initial world,
     * a copy of MPI_COMM_WORLD */
    MPI_Comm_dup( MPI_COMM_WORLD, &world );
…
  } else {
    /* I am a spare, lets get the repaired world */
    MPIX_Comm_replace( MPI_COMM_NULL, &world );
…
    goto joinwork;
  }
```





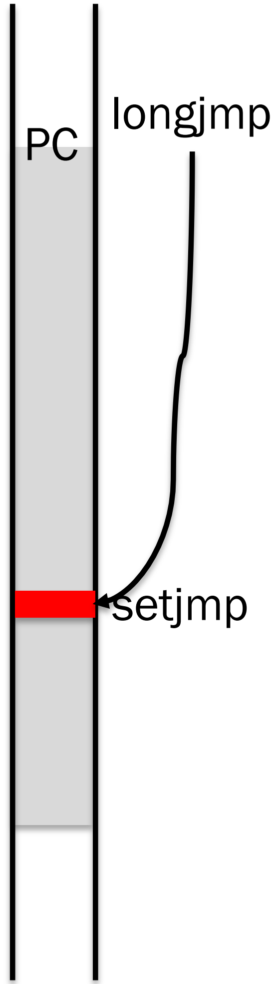
- Avoid the cost of having idling spares
  - We use MPI_Comm_spawn to launch new processes
  - We insert them with the right rank in a new "world"

Respawn in action: buddy C/R

See 12.buddycr.c

```
    MPI_Comm_get_parent( &parent );
    if( MPI_COMM_NULL == parent ) {
        /* First run: Let's create an initial world,
         * a copy of MPI_COMM_WORLD */
        MPI_Comm_dup( MPI_COMM_WORLD, &world );
…
    } else {
        /* I am a spare, lets get the repaired world */
        app_needs_repair(MPI_COMM_NULL);
    }
…
   setjmp(jmpenv);
    while(iteration < max_iterations) {
        /* take a checkpoint */
       if(0 == iteration%2) app_buddy_ckpt(world);
       iteration++;
```

- Do the operation until completion, and nobody else needs repair
- New spawns (obviously) need repair
- Function "app_needs_repair" reloads checkpoints, sets the restart iteration, etc…
- "app_needs_repair" Called upon restart, in the error handler, and before completion



Triggering the Restart

See 12.buddycr.c

```c
static int app_needs_repair(void) {
    MPI_Comm tmp;
    MPIX_Comm_replace(world, &tmp);
    if( tmp == world ) return false;
    if( MPI_COMM_NULL != world) MPI_Comm_free(&world);
    world = tmp;
    app_reload_ckpt(world);
    /* Report that world has changed, and we need to re-execute */
    return true;
}

/* Do all the magic in the error handler */
static void errhandler_respawn(MPI_Comm* pcomm, int* errcode, ...) {
 …
    if( MPIX_ERR_PROC_FAILED != eclass &&
        MPIX_ERR_REVOKED != eclass ) {
        MPI_Abort(MPI_COMM_WORLD, *errcode);
    }
    MPIX_Comm_revoke(*pcomm);
    if(app_needs_repair()) longjmp(jmpenv, 0);
}
```

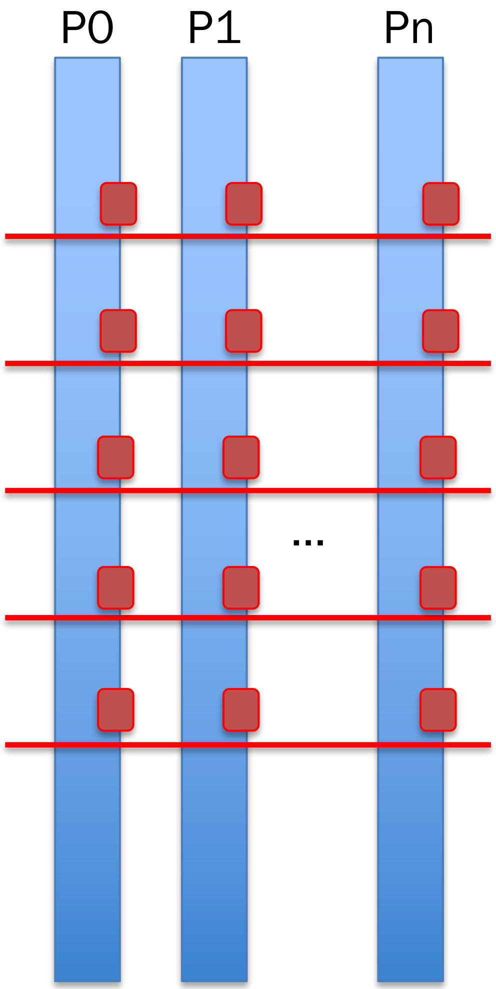
- Upon completion of the spawn and recreation of the new communicator if repairs have been done then we longjmp to skip the remaining of the loop and return to the last coherent version. Keep in mind that longjmp does not restore the variables but leaves them as they were at the moment of the fault.

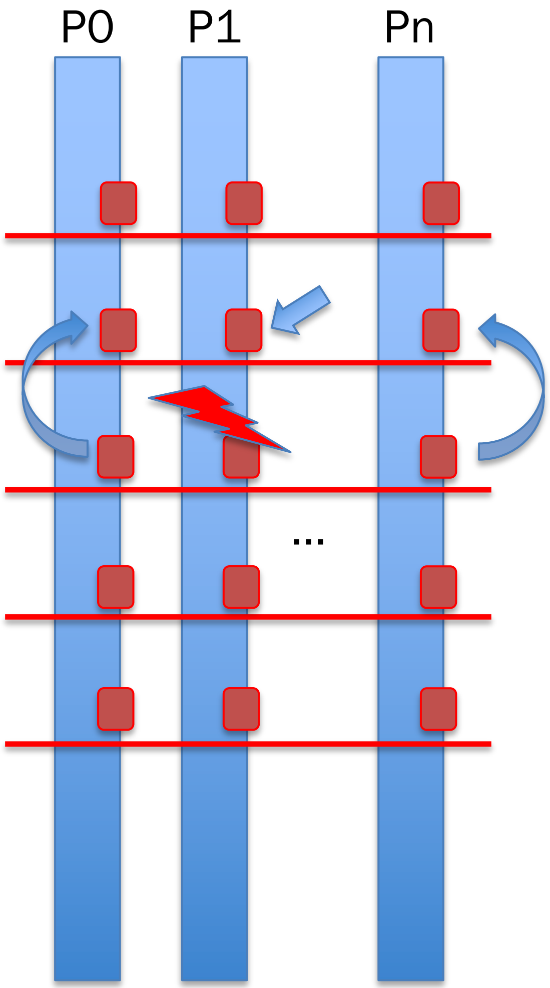
Simple Buddy Checkpoint

```
static int app_buddy_ckpt(MPI_Comm comm) {
    if(0 == rank || verbose) fprintf(stderr, "Rank %04d: checkpointing to %04d after iteration %d\n", rank, rbuddy(rank), iteration);
    /* Store my checkpoint on my "right" neighbor */
    MPI_Sendrecv(mydata_array, count, MPI_DOUBLE, rbuddy(rank), ckpt_tag,
                 buddy_ckpt,   count, MPI_DOUBLE, lbuddy(rank), ckpt_tag,
                 comm, MPI_STATUS_IGNORE);
    /* Commit the local changes to the checkpoints only if successful. */
    if(app_needs_repair()) {
        fprintf(stderr, "Rank %04d: checkpoint commit was not successful, rollback instead\n", rank);
        longjmp(jmpenv, 0);
    }
    ckpt_iteration = iteration;
    /* Memcpy my own memory in my local checkpoint (with datatypes) */
    MPI_Sendrecv(mydata_array, count, MPI_DOUBLE, 0, ckpt_tag,
                 my_ckpt, count, MPI_DOUBLE, 0, ckpt_tag,
                 MPI_COMM_SELF, MPI_STATUS_IGNORE);
    return MPI_SUCCESS;
}
```

See 12.buddycr.c

Transaction-like approaches





- Each transaction is either
  - successful and the execution progresses, or
  - fails, and the entire transaction needs to be re-executed
- This requires a copy of all data necessary for the re-execution readily available

Transaction-like approaches

```
/* save data to be used in the code below */

do {
  /* if not original instance restore the data */

  /* do some extremely useful work */

  /* validate that no errors happened */

} while  (!errors)
```

- Let's not focus on the data save and restore
- Use the agreement to decide when a work unit is valid
- If the agreement succeed the work is correctly completed and we can move forward
- If the agreement fails restore and data and redo the computations
- Use REVOKE to propagate specific exception every time it is necessary (even in the work part)
- Exceptions must be bits to be able to work with the agreement

Transaction-like approaches

```
#define TRY_BLOCK(COMM, EXCEPTION) \
do {                               \
  int __flag = 0xffffffff;         \
  __stack_pos++;                   \
  EXCEPTION = setjmp(&stack_jmp_buf[__stack_pos]);\
  __flag &= ~EXCEPTION;            \
  if( 0 == EXCEPTION ) {

#define CATCH_BLOCK(COMM)          \
    __stack_pos--;                 \
    __stack_in_agree = 1;  /* prevent longjmp */ \
    MPIX_Comm_agree(COMM, &__flag); \
    __stack_in_agree = 0; /* enable longjmp */ \
  }                                \
  if( 0xffffffff != __flag ) {

#define END_BLOCK()                \
  } } while (0);

#define RAISE(COMM, EXCEPTION)     \
  MPIX_Comm_revoke(COMM);          \
  assert(0 != (EXCEPTION));        \
  if(!__stack_in_agree )           \
    longjmp( stack_jmp_buf[__stack_pos],
             (EXCEPTION) ); /* escape */
```

- TRY_BLOCK setup the transaction, by setting a setjmp point and the main if
- CATCH_BLOCK complete the if from the TRY_BLOCK and implement the agreement about the success of the work completion
- END_BLOCK close the code block started by the TRY_BLOCK
- RAISE revoke the communicator and if necessary (if not raised from the agreement) longjmp at the beginning of the TRY_BLOCK catching the if

Transaction-like approaches

```
/* save data1 to be used in the code
below */
transaction1:
TRY_BLOCK(MPI_COMM_WORLD, exception) {

     /* do some extremely useful work */

    /* save data2 to be used in the cod
below */
transaction2:
    TRY_BLOCK(newcomm, exception) {

        /* do more extremely useful wor
*/

    } CATCH_BLOCK(newcomm) {
        /* restore data2 for transaction
2 */
        goto transaction2;
    } END_BLOCK()

} CATCH_BLOCK(MPI_COMM_WORLD) {
    /* restore data1 for transaction 1
```

Transaction 1

Transaction 2

- Skeleton for a 2 level transaction with checkpoint approach
  - Local checkpoint can be used to handle soft errors
  - Other types of checkpoint can be used to handle hard errors
  - No need for global checkpoint, only save what will be modified during the transaction
- Generic scheme that can work at any depth

Transaction-like approaches

See 13.transactions.c

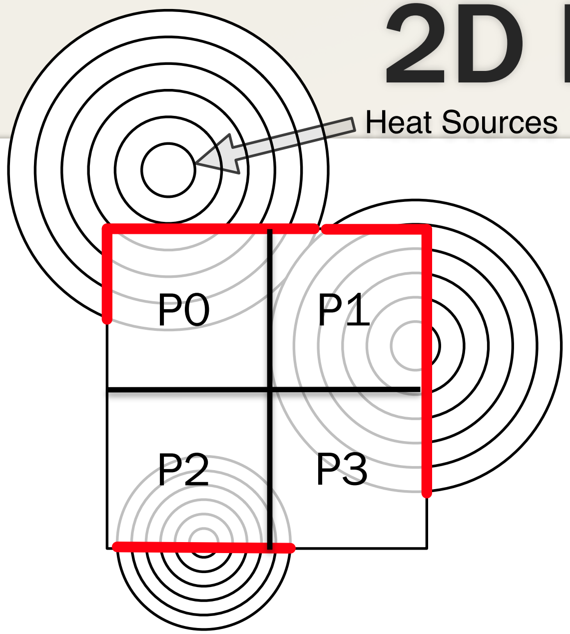
```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

TRY_BLOCK(MPI_COMM_WORLD, exception) {

    int rank, size;

    MPI_Comm_dup(MPI_COMM_WORLD, &newcomm);
    MPI_Comm_rank(newcomm, &rank);
    MPI_Comm_size(newcomm, &size);

    TRY_BLOCK(newcomm, exception) {

        if( rank == (size-1) ) exit(0);
        rc = MPI_Barrier(newcomm);

    } CATCH_BLOCK(newcomm) {
    } END_BLOCK()

} CATCH_BLOCK(MPI_COMM_WORLD) {
} END_BLOCK()
```

Transaction 2

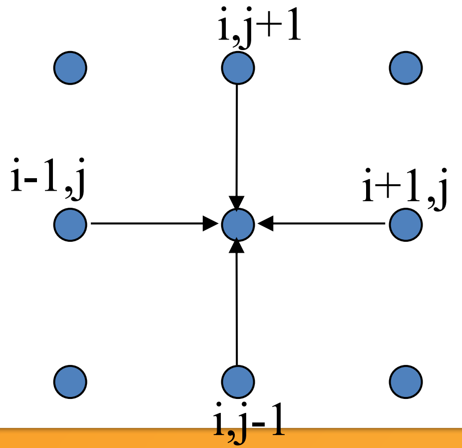
Transaction 1

- A small example doing a simple barrier
- We manually kill a process by brutally calling exit
- What is the correct or the expected output?

2D Heat Propagation (Laplace eq.)

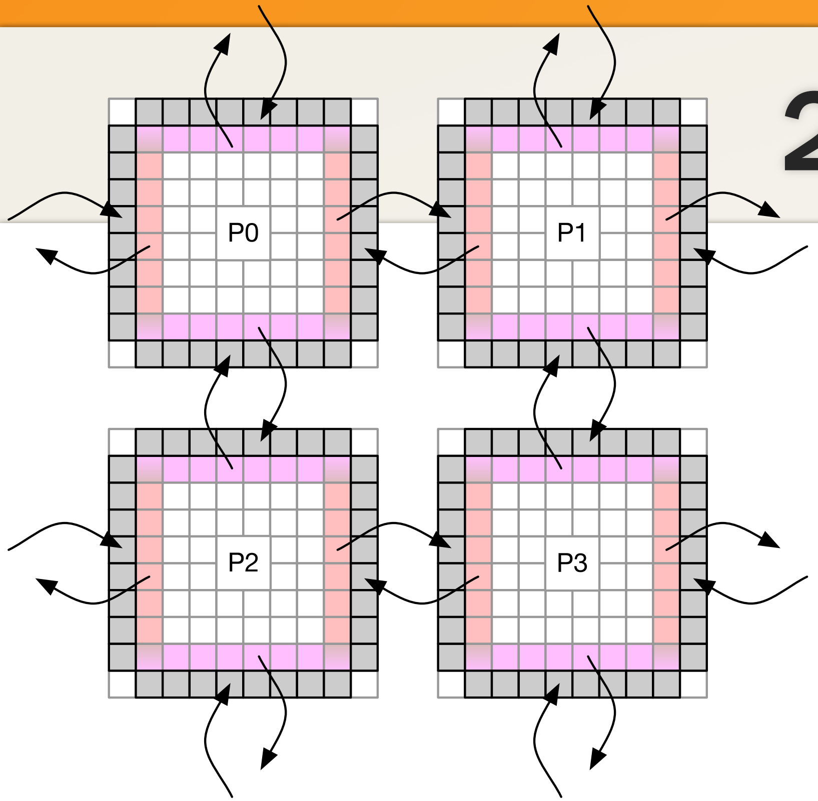


$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$



- The root of many types of scientific challenges
  - The implementation used here is however trivial, and only serve teaching purposes
- We imagine a NxM points space represented as a matrix and distributed on a PxQ grid of processes
  - Each process has (N/P) x (M/Q) elements
  - To facilitate the update each process will surround the part of the space she owns with a ghost region, that role is to hold the data from the last iteration from the neighbor on the direction

2D Heat Propagation (Laplace eq.)

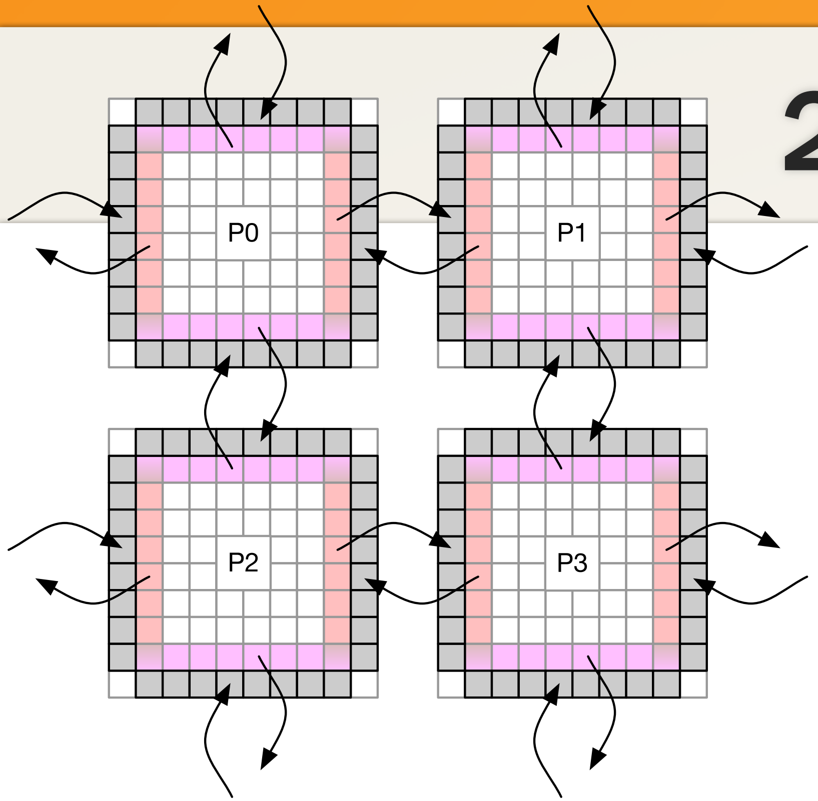


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to save data on the buddy at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/jacobi_cpu_noft.c

```
  set error handlers
restart:
  recover = setjmp()

  build row and column communicators
  if recover { get data from buddy
               goto local_computation }

  do {
    exchange data with neighbors

    if time for buddy chkpt: save local data on buddy

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```

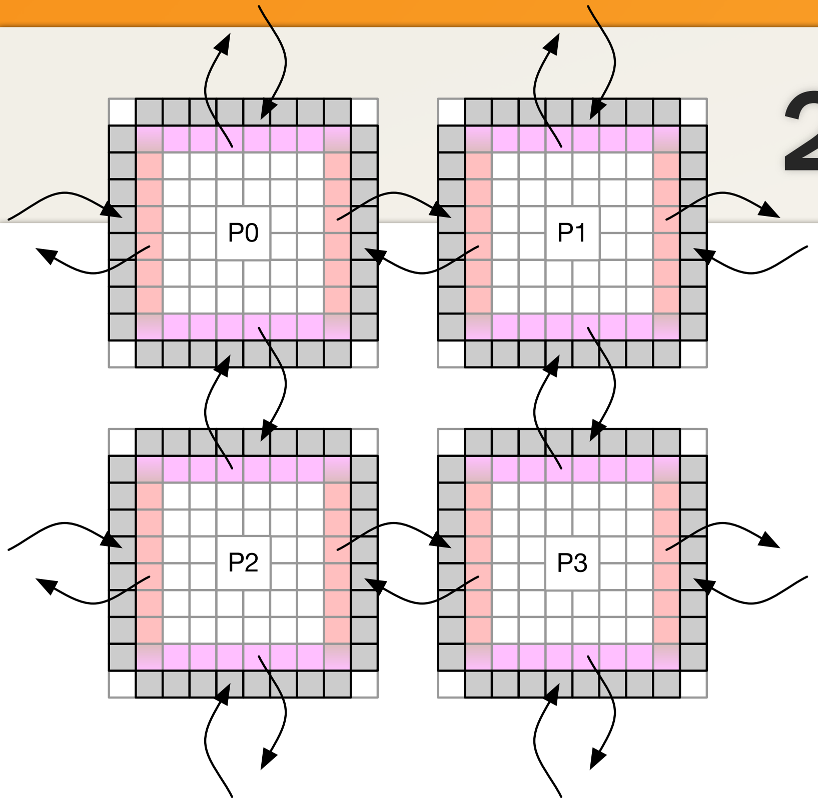
2D Heat Propagation (Laplace eq.)



1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to save data on the buddy at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/jacobi_cpu_bckpt.c

```
  set error handlers
restart:
  recover = setjmp()

  build row and column communicators
  if recover { get data from buddy
               goto local_computation }

  do {
    exchange data with neighbors

    if time for buddy chkpt: save local data on buddy

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```



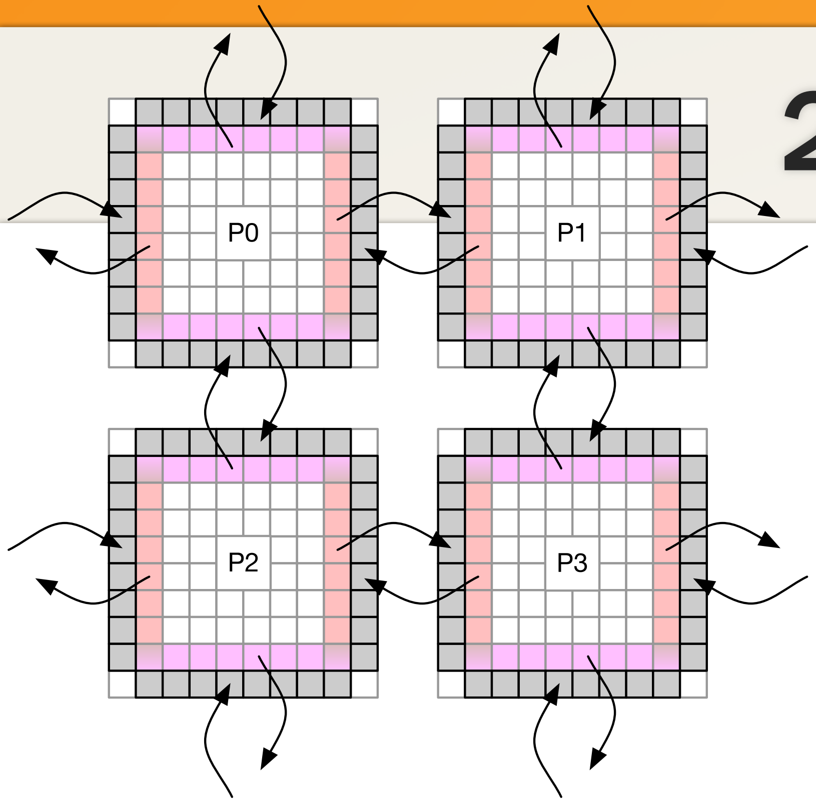
2D Heat Propagation (Laplace eq.)

1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to save data on the buddy at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/jacobi_cpu_bckpt.c

```
  set error handlers
restart:
  recover = setjmp()

  build row and column communicators
  if recover { get data from buddy
               goto local_computation }

  do {
    exchange data with neighbors

    if time for buddy chkpt: save local data on buddy

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```

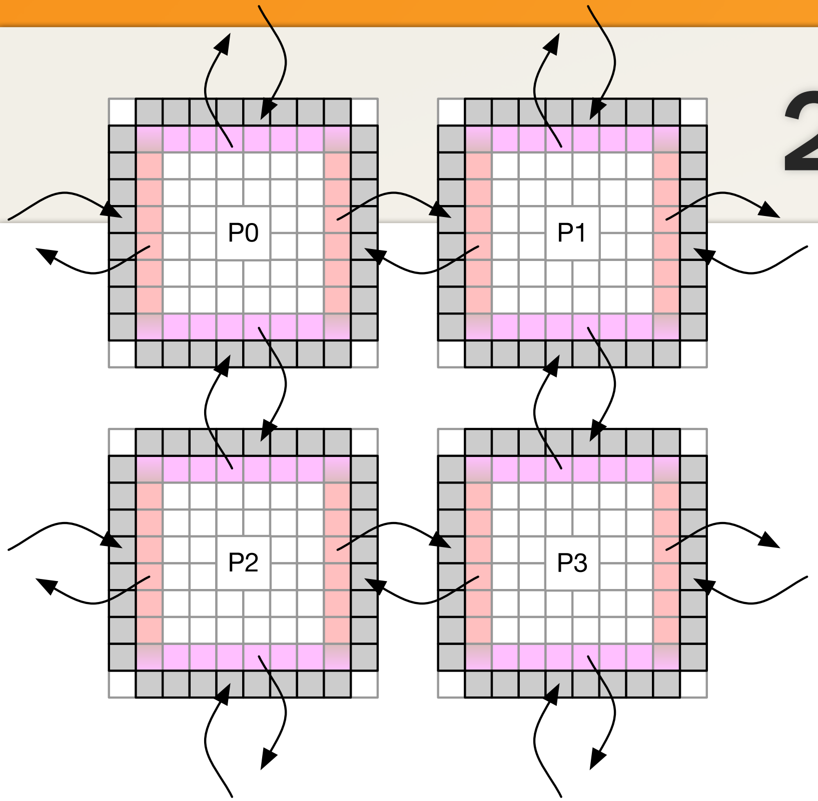
2D Heat Propagation (Laplace eq.)



1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to save data on the buddy at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/jacobi_cpu_bckpt.c

```
  set error handlers
restart:
  recover = setjmp()

  build row and column communicators
  if recover { get data from buddy
               goto local_computation }

  do {
    exchange data with neighbors

    if time for buddy chkpt: save local data on buddy

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```



2D Heat Propagation (Laplace eq.)

1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to save data on the buddy at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

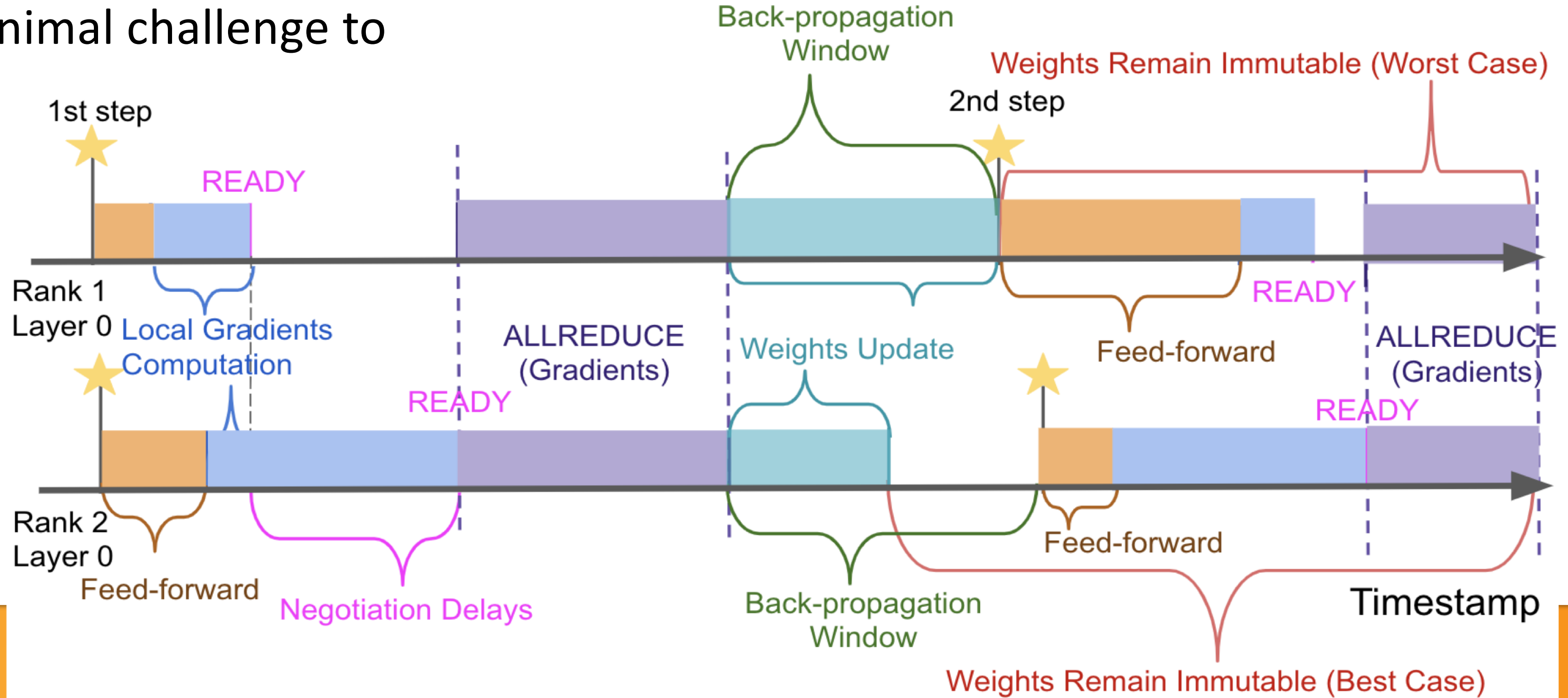
See jacobi/jacobi_cpu_bckpt.c

```
  set error handlers
restart:
  recover = setjmp()

  build row and column communicators
  if recover { get data from buddy
               goto local_computation }

  do {
    exchange data with neighbors

    if time for buddy chkpt: save local data on buddy

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```

ML: Parallel Training: Timeline Analytics

Fine-grain metrics per tensor for timeline analytics

- **Feed-forward** (averaged among all steps)
- **Negotiation delays** (average among all ranks)
- **Back-propagation window per tensor** (averaged; potential opportunities to delay next phase)
- All participants have a copy of the converged gradients, and this is the only information needed to continue the learning process
  - A restarted process can simply receive the gradients from a buddy and with access to the original test input data it can seamlessly continue the execution
  - Minimal challenge to



Beyond examples, what people are doing with it

USER'S RECOVERY STORIES

User Level Failure Mitigation: User Adoption

Fenix Framework/S3D

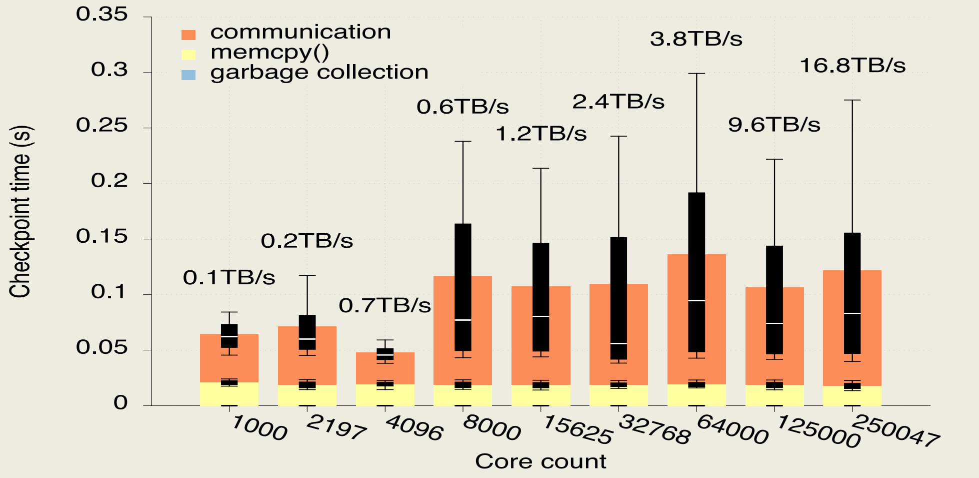


Fig. 3. Checkpoint time for different core counts (8.6 MB/core). The numbers above each test show the aggregated bandwidth (the total checkpoint size over the average checkpoint time).

- Fortran CoArrays "failed images" uses ULFM-RMA to support Fortran TS 18508
- SAP In-memory distributed database
- PHALANX
- Elastic X10

MapReduce

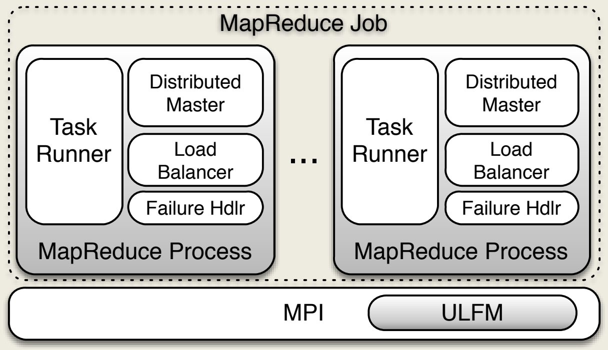
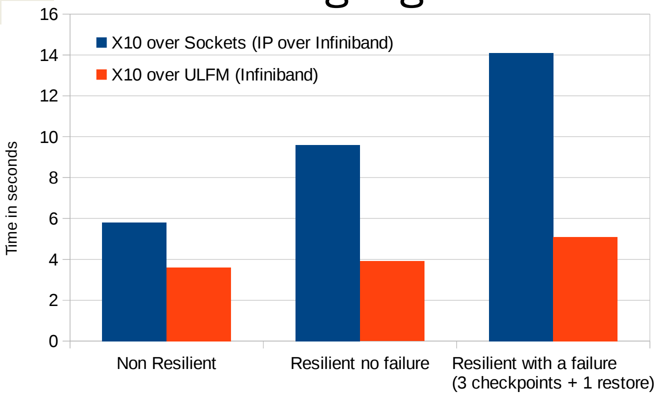


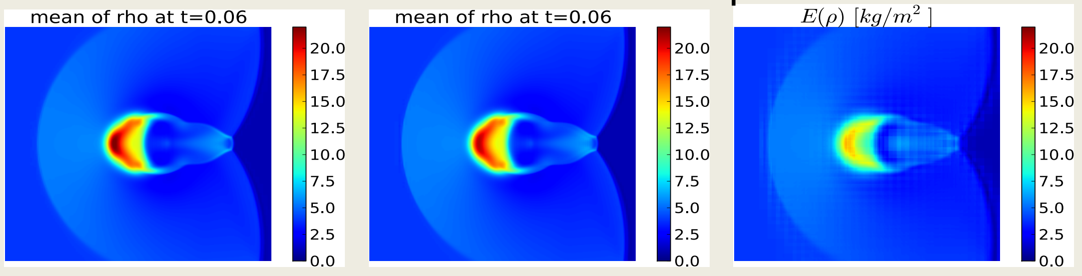
Figure 2: The architecture of FT-MRMPI.

X10 Language



The performance improvement due to using ULFM v1.0 for running the LULESH proxy application [3] (a shock hydrodynamics stencil based simulation) running on 64 processes on 16 nodes with

Domain Decomposition PDE



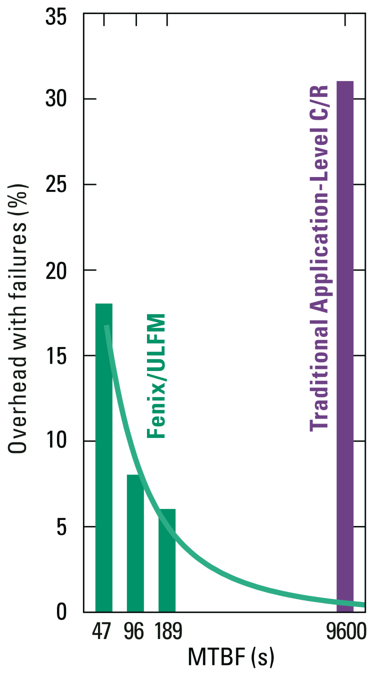
(a) failure-free (b) few failures (c) many failures

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

And many more…

Use cases: Chekpoints w/Fenix in S3D

- S3D is a production, highly parallel method-of-lines solver for PDEs
  - used to perform first-principles-based direct numerical simulations of turbulent combustion
- S3D rendered fault tolerant using Fenix/ULFM
- 35 lines of code modified in S3D in total!
- Order of magnitude performance improvement in failure scenarios
  - thanks to online recovery and in-memory checkpoint advantage over I/O based checkpointing
- Injection of FT layer: addition of a couple of Fenix calls



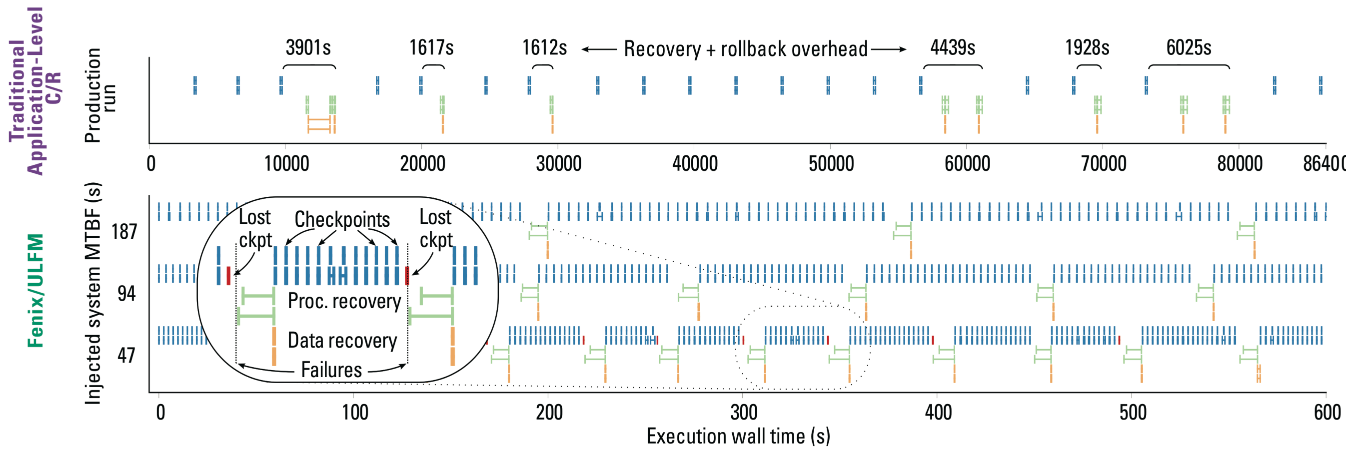


Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

Fenix_Checkpoint_Allocate mark a memory segment (baseptr,size) as part of the checkpoint.
Fenix_Init Initialize Fenix, and restart point after a recovery, status contains info about the restart mode
Fenix_Comm_Add can be used to notify Fenix about the creation of user communicators
Fenix_Checkpoint performs a checkpoint of marked segments

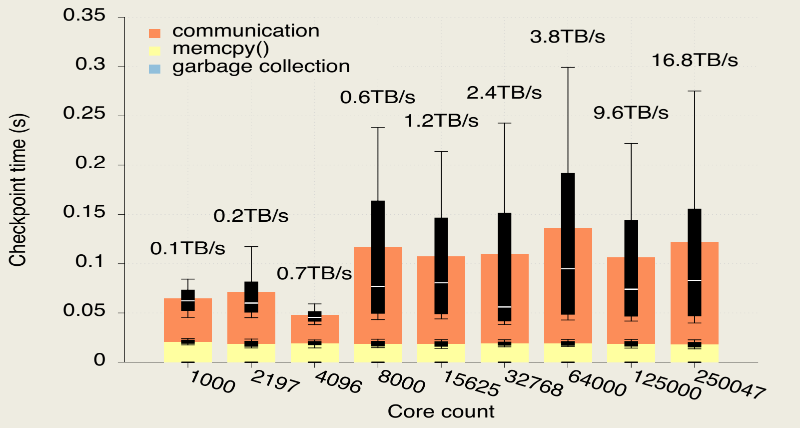


Fig. 3. Checkpoint time for different core counts (8.6 MB/core). The numbers above each test show the aggregated bandwidth (the total checkpoint size over the average checkpoint time).

FRAMEWORKS USING ULFM

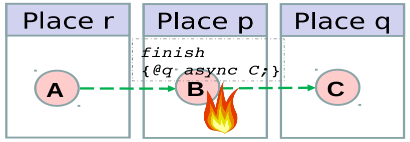
LFLR, FENIX, FTLA, Falanx

Use cases: Languages
Resilient X10

- X10 is a PGAS programming language
  - Legacy resilient X10 TCP based

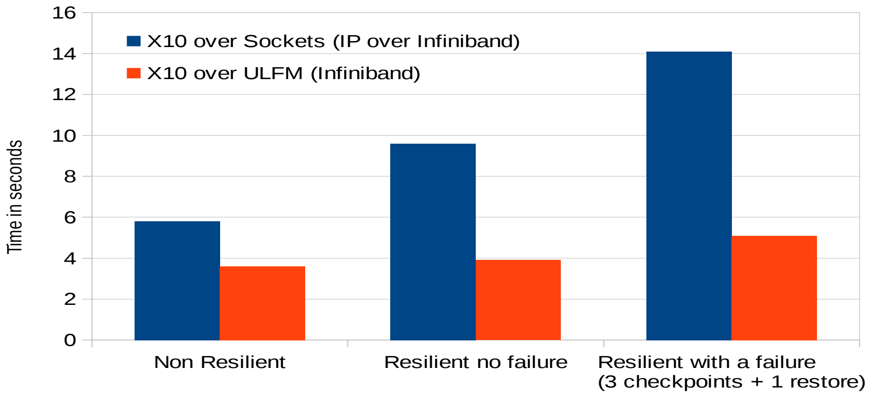
Happens Before Invariance Principle (HBI):
Failure of a place should not alter the happens before relationship between statements at the remaining places.

```
try{ /*Task A*/
 at (p) { /*Task B*/
  finish { at (q) async { /*Task C*/ } }
 }
} catch(dpe:DeadPlaceException){ /*recovery steps*/}
D;
```



By applying the HBI principle, Resilient X10 will ensure that statement D executes after Task C finishes, despite the loss of the synchronization construct (finish) at place p

- MPI operations in resilient X10 runtime
  - Progress loop does MPI_Iprobe, post needed recv according to probes
  - Asynchronous background collective operations (on multiple different comms to form 2d grids, etc).
- Recovery
  - Upon failure, all communicators recreated (from shrinking a large communicator with spares, or using MPI_COMM_SPAWN to get new ones)
  - Ranks reassigned identically to rebuild the same X10 "teams"
- Injection of FT layer
  - Unnecessary, x10 has a runtime that hides all MPI from the application and handles failures internally



The performance improvement due to using ULFM v1.0 for running the LULESH proxy application [3] (a shock hydrodynamics stencil based simulation) running on 64 processes on 16 nodes with

Source: Sara Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, Olivier Tardieu. *Resilient X10 over Fault Tolerant MPI.* In : poster session SC'15, Austin, TX, 2015.

Use cases: Non traditional HPC

Hadoop over MPI

- Non-HPC workflow usually do not consider MPI because it lacks FT

  Judicael A. Zounmevo, Dries Kimpe, Robert Ross, and Ahmad Afsahi. 2013. Using MPI in high-performance computing services. In *Proceedings of the 20th European MPI Users' Group Meeting* (EuroMPI '13). ACM, New York, NY, USA, 43-48.*SE), 2013 IEEE 16th International Conference on*. IEEE, 2013. p. 58-65.

  - ULFM permits high performance exchange in non-HPC runtimes (like Hadoop)

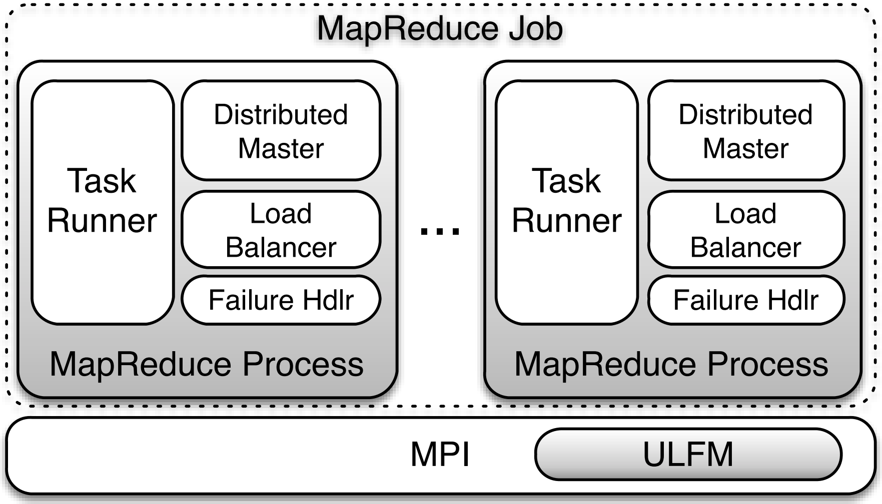


Figure 2: The architecture of FT-MRMPI.

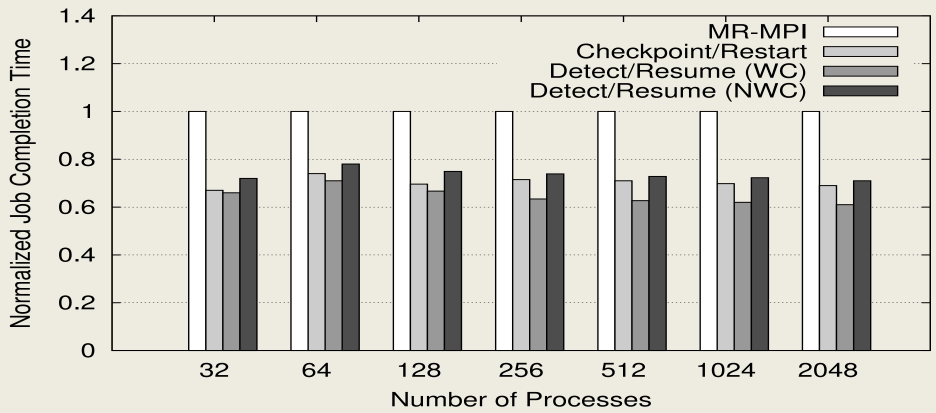


Figure 8: Normalized job completion time of failed and recovery run.

Use cases: Non traditional HPC

SAP Databases

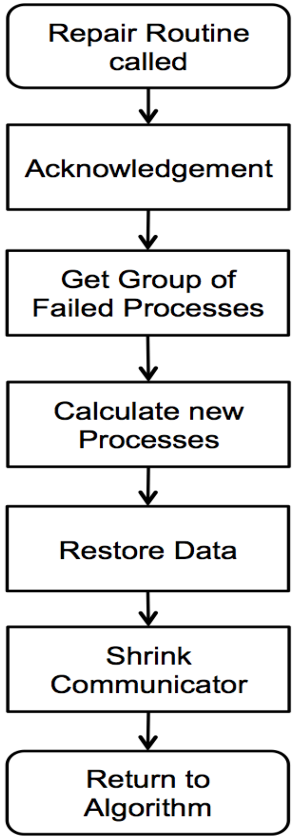


Figure 3.2: Repair Routine

- SAP is a production database system
  - Implemented over MPI for high performance applications
  - Legacy: Fault tolerance based on full-restart
- SAP with ULFM
  - Collective operations consistency protected by agreements
  - Database Request continues in-place after an error

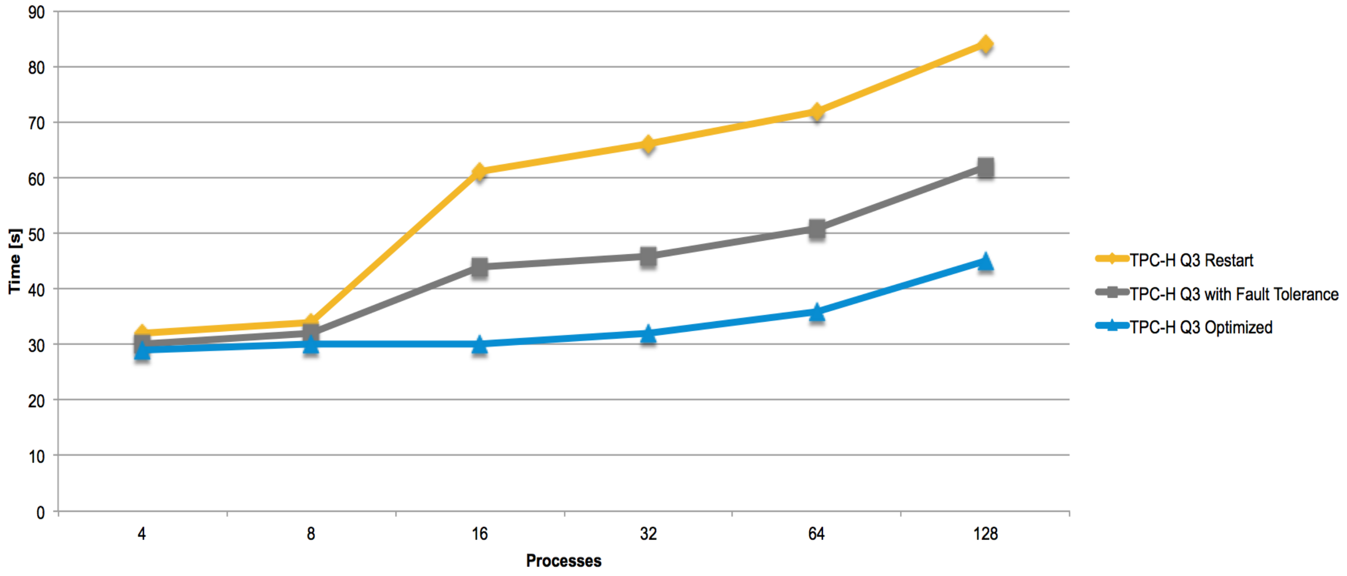
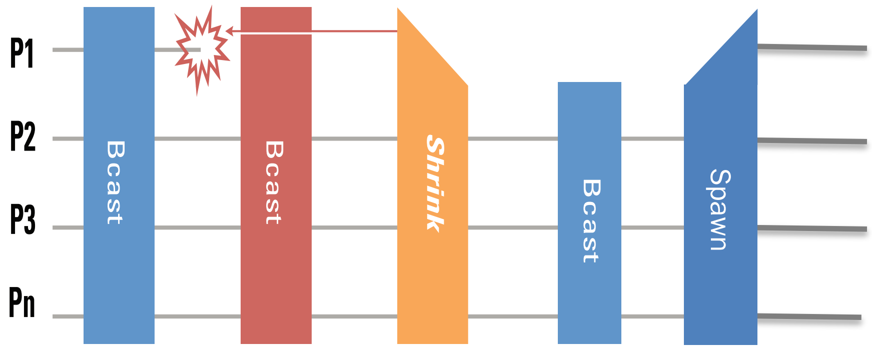


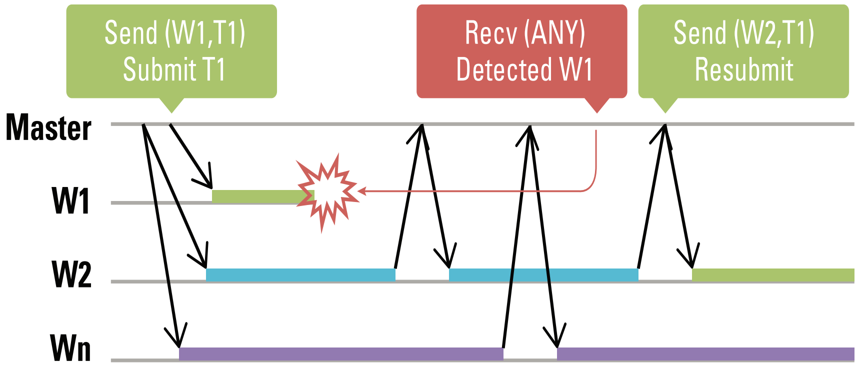
Figure 5.24: Optimization: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 4 (1GB Data per Process)

Source: Fault Tolerant Collective Communication Algorithms for Distributed Database Systems
Fehlertolerante Gruppenkommunikations Algorithmen für verteilte Datenbanksysteme
Master-Thesis von Jan Stengler aus Mainz April 2017

CONCLUSION

ULFM: support for all FT types





- You application is SPMD
  - Coordinated recovery
  - Checkpoint/restart based
  - ABFT
- ULFM can rebuild the same communicators as before the failure!

- Your application is moldable
  - Parameter sweep
  - Master Worker
  - Dynamic load balancing
- ULFM can reduce the cost of recovery by letting you continue to use a communicator in limited mode (p2p only)

Other mechanisms

- Supported but not covered in this tutorial: one-sided communications and files
  - Files: MPI_FILE_REVOKE
  - One-sided: MPI_WIN_REVOKE, MPI_WIN_GET_FAILED
- All other communicator based mechanisms are supported via the underlying communicator of these objects.

Why all these efforts?

Toward Exascale Computing (My Roadmap)

Based on proposed DOE roadmap with MTTI adjusted to scale linearly

Systems	2009	2011	2015	2018
System peak	2 Peta	20 Peta	100-200 Peta	1 Exa
System memory	0.3 PB	1.6 PB	5 PB	10 PB
Node performance	125 GF	200GF	200-400 GF	1-10TF
Node memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node concurrency	12	32	O(100)	O(1000)
Interconnect BW	1.5 GB/s	22 GB/s	25 GB/s	50 GB/s
System size (nodes)	18,700	100,000	500,000	O(million)
Total concurrency	225,000	3,200,000	O(50,000,000)	O(billion)
Storage	15 PB	30 PB	150 PB	300 PB
IO	0.2 TB/s	2 TB/s	10 TB/s	20 TB/s
MTTI	4 days	19 h 4 min	3 h 52 min	1 h 56 min
Power	6 MW	~10MW	~10 MW	~20 MW

Prediction is very difficult

especially about the future

Niels Bohr, Physicist - Nobel Prize Winner

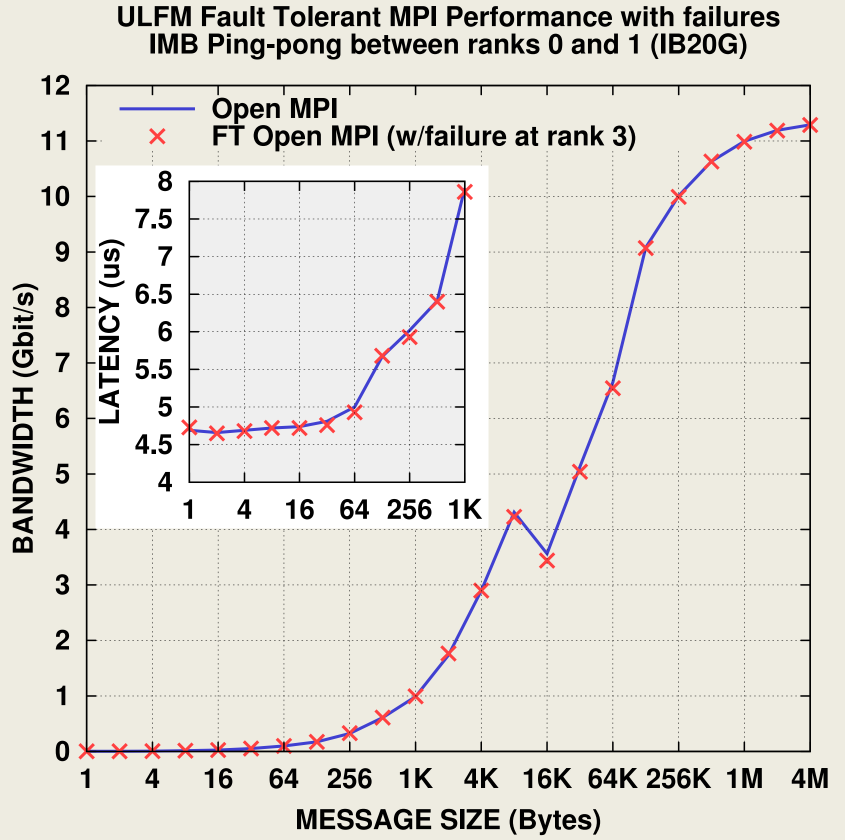
www.Schrijf.be - Tijdnet

- Whatever scenario we are going for our Exascale platforms the MTBF will just keep shrinking

ULFM MPI: Software Infrastructure

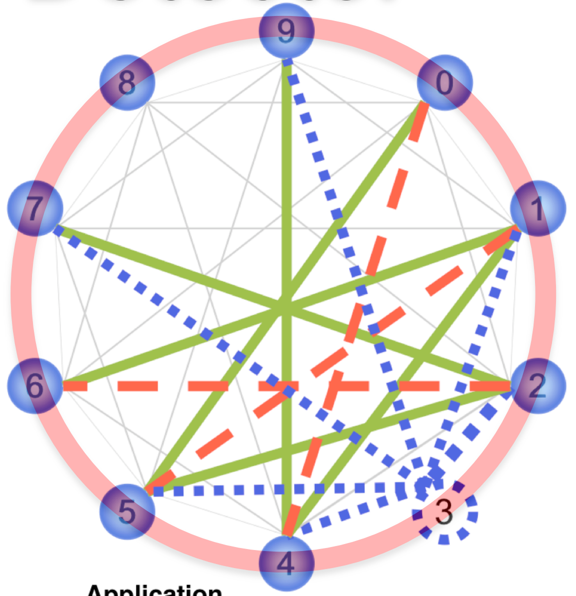
- Implementation in Open MPI, MPICH available
- Very good performance w/o failures
- Open MPI ULFM 4.0.2u1 status
  - In sync with Open MPI master (2 weeks ago)
- New features
  - SC'16 failure detector integrated (threaded detector, RDMA heartbeats optimization, etc.)
  - PMIx notifications taken into account
  - Fault tolerance with 1-copy CMA shared memory
  - Fault tolerance with Non-blocking collective operations
  - Fail gracefully when failure hit during MPI-IO
  - Fail gracefully when failure hit during MPI-RMA
  - Slurm, PBS, support improved
  - Tested on Cori, Edison, Titan, Summit, etc.
  - Failure free performance bump!

Performance w/failures



The failure of rank 3 is detected and managed by rank 2 during the 512 bytes message test. The connectivity and bandwidth between rank 0 and rank 1 are unaffected by failure handling activities at rank 2.

Scalable Failure Detector



f = supported number of overlapping failures

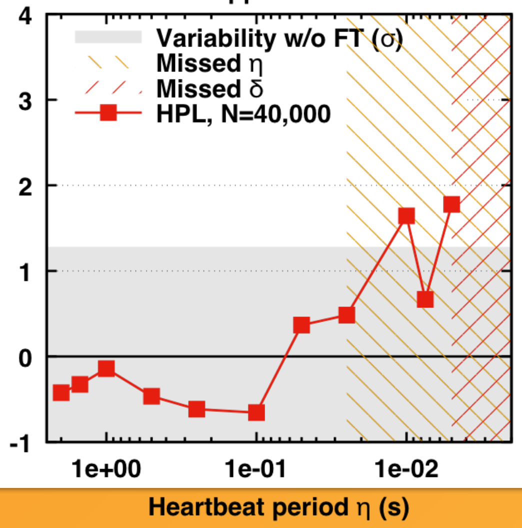
Stabilization Time T(f) = duration of the longest sequence of non stable configurations assuming at most f overlapping faults

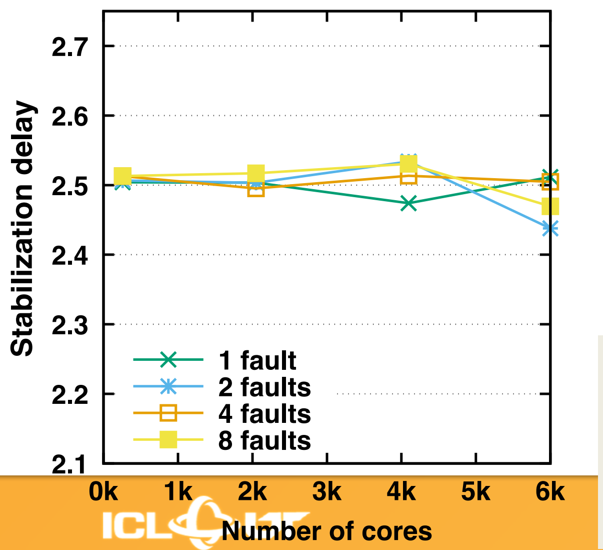
Broadcast Time B(n) $= 8\tau \log n$

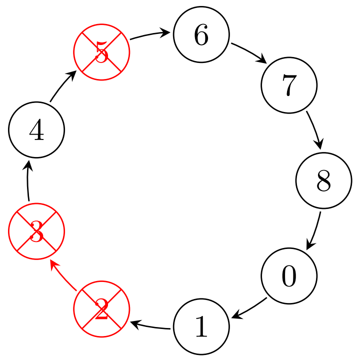
$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{\text{reconnect}} + \underbrace{\frac{f(f+1)}{2} B(n)}_{\text{propagate}}$$

The broadcast algorithm can tolerate up to $\lfloor log(n) \rfloor$ overlapping failures, thus

$$T(f) \sim O((log\ n)^3)$$

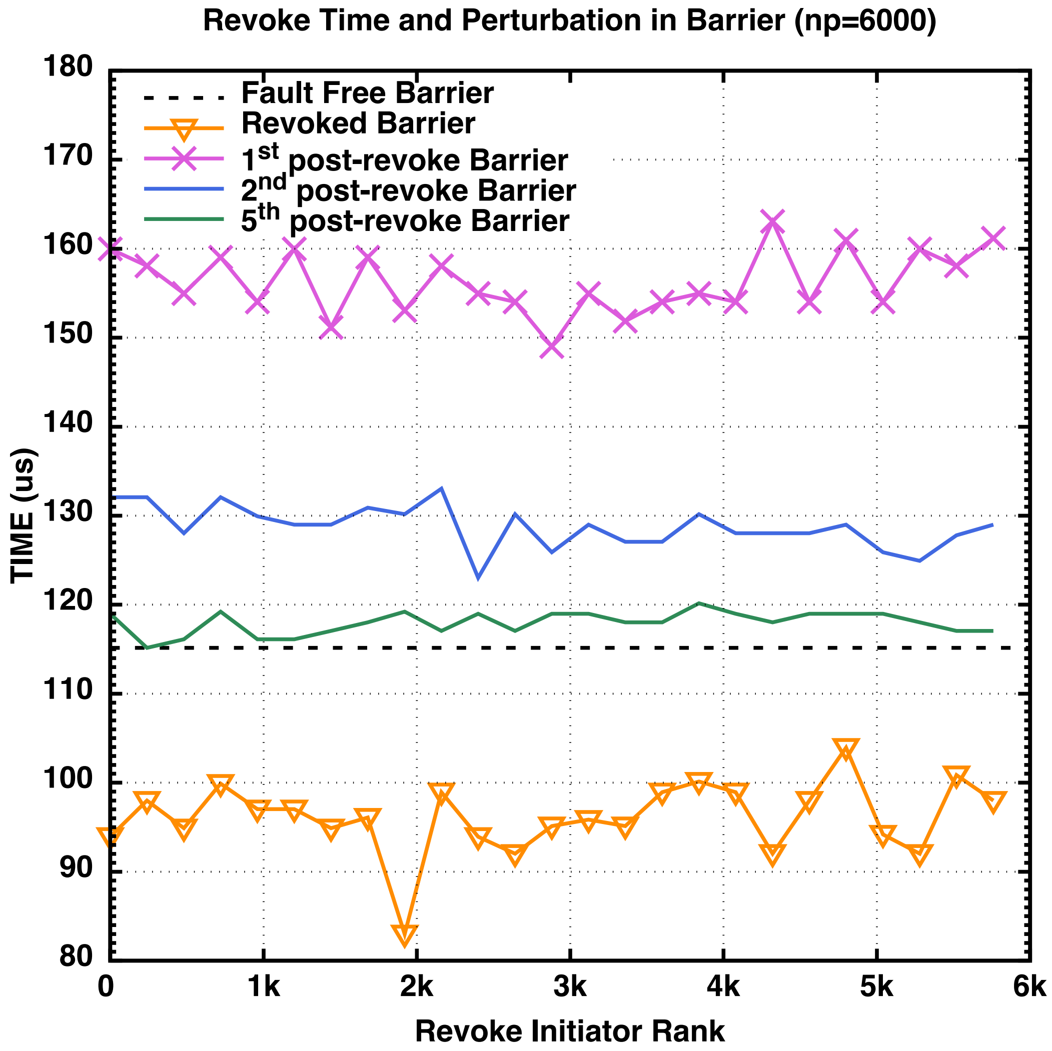






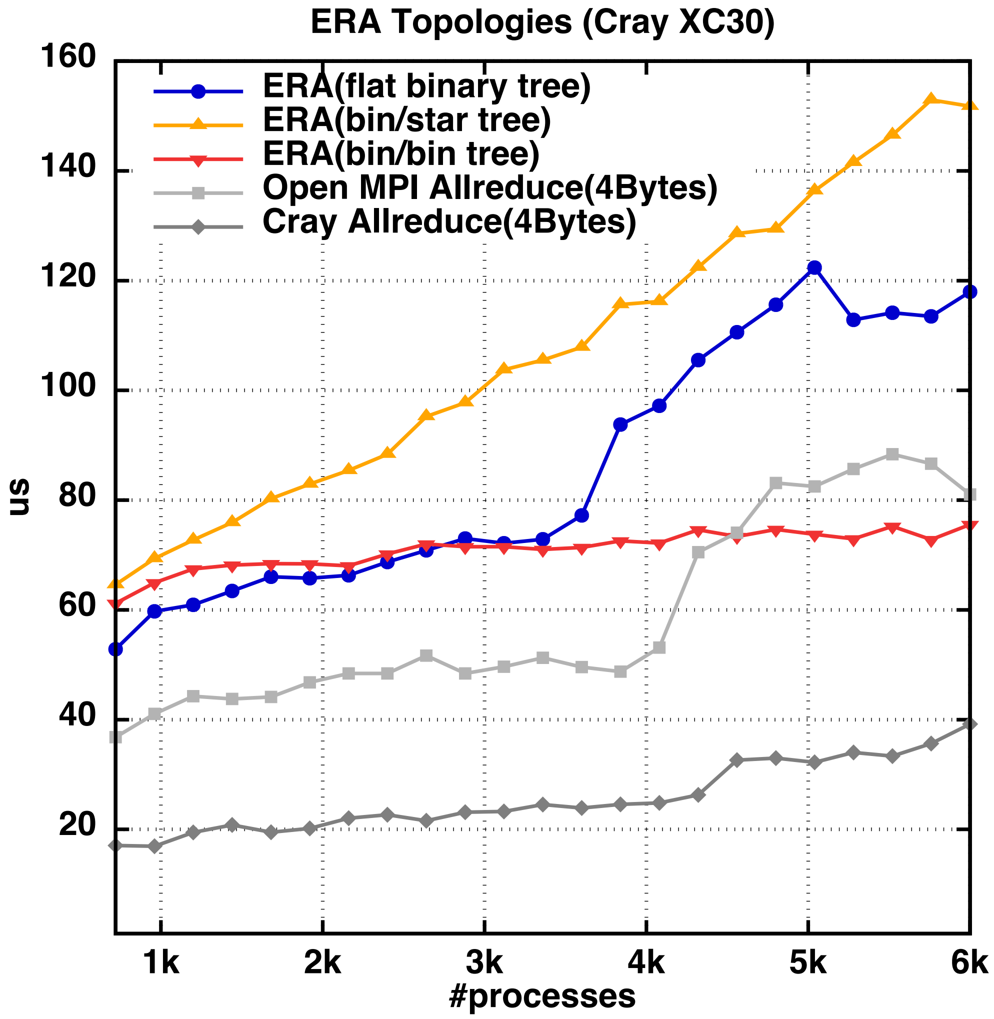
Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J. "Failure Detection and Propagation in HPC systems," SuperComputing, Salt Lake City, UT, November, 2016

Scalable Revocation



- The underlying BMG topology is symmetric and reflects in the revoke which is independent of the initiator
- The performance of the first post-Revoke collective operation sustains some performance degradation resulting from the network jitter associated with the circulation of revoke tokens
- After the fifth Barrier (approximately 700μs), the application is fully resynchronized, and the Revoke reliable broadcast has completely terminated, therefore leaving the application free from observable jitter.

Scalable Agreement



- Early Returning Algorithm: once the decision reached the local process returns, but the decided value remains available for providing to other processes
- The underlying logical topology hierarchically adapts to reflects to network topology
- In the failure-free case the implementation exhibits the theoretically proven logarithmic behavior, similar to an optimized version of MPI_Allreduce

So what is the right approach

- Bad news: there might not be A right approach
- An efficient, scalable and portable approach might be at the frontier of multiple approaches
- So far the algorithm specific approaches seems the most efficient, but they have additional requirements from the programming paradigms
- We need fault tolerance support from the programming paradigms
  - The glue to allow composability if as important as the approaches themselves
- Is ULFM that glue?

What about the development cost?

- ULFM has a steep learning cost compared with system level approaches. But:
  - Parallel programming was considered hard a while back. Today it is almost mainstream (!)
    - Training is key for flatten the learning curve
  - ULFM is a building box, most developers are not supposed to use it directly
    - Instead use domain specific approaches, proposed by the domain scientists as a portable library implemented using the ULFM constructs
- The development cost should be put in balance with the building and ownership cost

Can we fix C/R with hardware?

- NVRAM ? Hardware level triplication? Hardware detection (think ECC++)
- More hardware is not only more expensive, but it also increases
  - The opportunity for failure (the law of big numbers)
  - The cost of ownership (energy, and cooling)
- Why not using this extra hardware to improve the scalability of the application?

You have the answer!

- You learned how to model the behavior of your application and how to interpret the data
- You learned what you can do if you go outsize the box (compose approaches, ULFM, …)
- You know your algorithms and applications

- We are looking forward to hear about your successes !

More info, examples and resources available

http://fault-tolerance.org

Your opinion matters!
File the SC19 tutorial evaluation form
http://bit.ly/sc19-eval

ICL UT
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

How to design your own replace/spare system (not presented live)

ADVANCED CONTENT

Inside MPIX_COMM_REPLACE

See 10.respawn

```c
if( comm == MPI_COMM_NULL ) { /* am I a new process? */
    /* I am a new spawnee, waiting for my new rank assignment
     * it will be sent by rank 0 in the old world */
    MPI_Comm_get_parent(&icomm);
    MPI_Recv(&crank, 1, MPI_INT, 0, 1, icomm, MPI_STATUS_IGNORE);
…
 }
 else {
    /* I am a survivor: Spawn the appropriate number
     * of replacement
    /* First: remove dead processes */
    MPIX_Comm_shrink(comm, &scomm);
    MPI_Comm_size(scomm, &ns);
    MPI_Comm_size(comm, &nc);
    nd = nc-ns; /* number of deads */
    if( 0 == nd ) {
       /* Nobody was dead to start with. We are done here */
…
       return MPI_SUCCESS;
    }
    /* We handle failures during this function ourselves... */
    MPI_Comm_set_errhandler( scomm, MPI_ERRORS_RETURN );
    rc = MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
                        0, scomm, &icomm, MPI_ERRCODES_IGNORE);
```

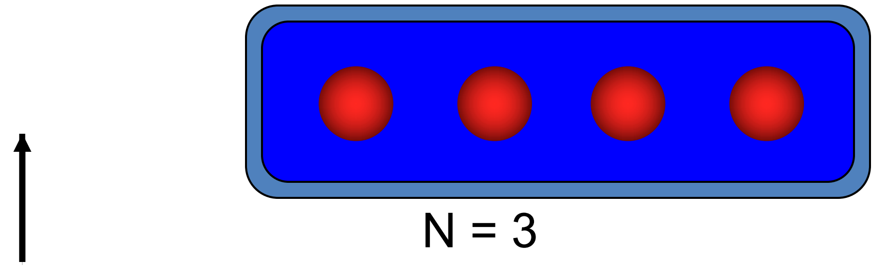
Same as in spare: new guys wait for their rank from 0 in the old world

Spawn *nd* new processes

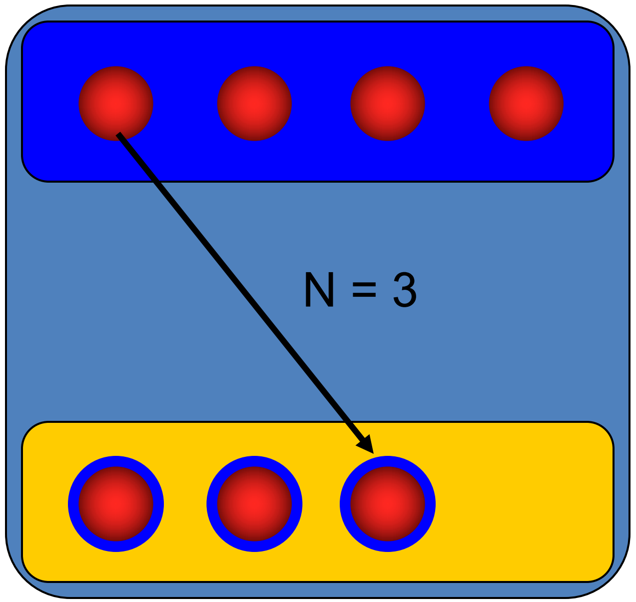
Intercommunicators – P2P

On process 0:
MPI_Send(buf, MPI_INT, 1, n, tag, intercomm)

- Intracommunicator

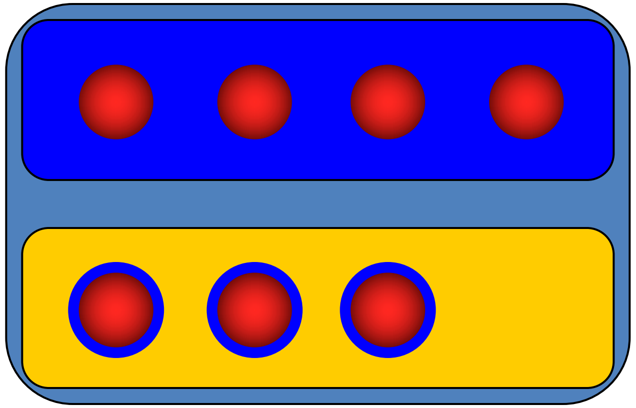


- Intercommunicator



Intercommunicators

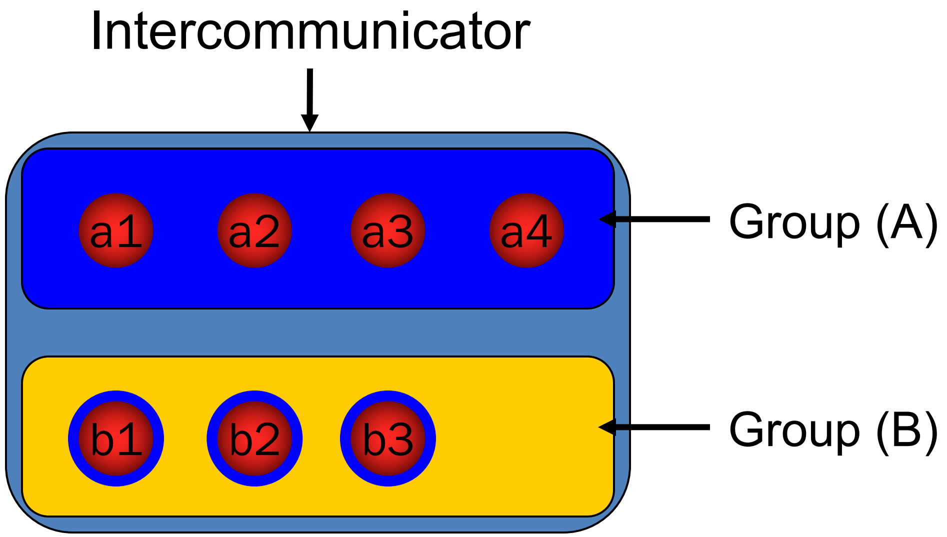
- And what’s a intercommunicator ?



- some more processes
- **TWO** groups
- one communicator

- MPI_COMM_REMOTE_SIZE(comm, size)
  MPI_COMM_REMOTE_GROUP(comm, group)
- MPI_COMM_TEST_INTER(comm, flag)
- MPI_COMM_SIZE, MPI_COMM_RANK return the local size respectively rank

Anatomy of a Intercommunicator



It's not possible to send a message to a process in the same group using this communicator

For any processes from group (A) • (A) is the local group • (B) is the remote group	For any processes from group (B) • (A) is the remote group • (B) is the local group

Inside MPIX_Comm_replace

```
    rc = MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
                        0, scomm, &icomm, MPI_ERRCODES_IGNORE);
    flag = (MPI_SUCCESS == rc);
    MPIX_Comm_agree(scomm, &flag);
    if( !flag ) {
        if( MPI_SUCCESS == rc ) {
            MPIX_Comm_revoke(icomm);
            MPI_Comm_free(&icomm);
        }
        MPI_Comm_free(&scomm);
…
        goto redo;
    }
```

Check if spawn worked (using the shrink comm)

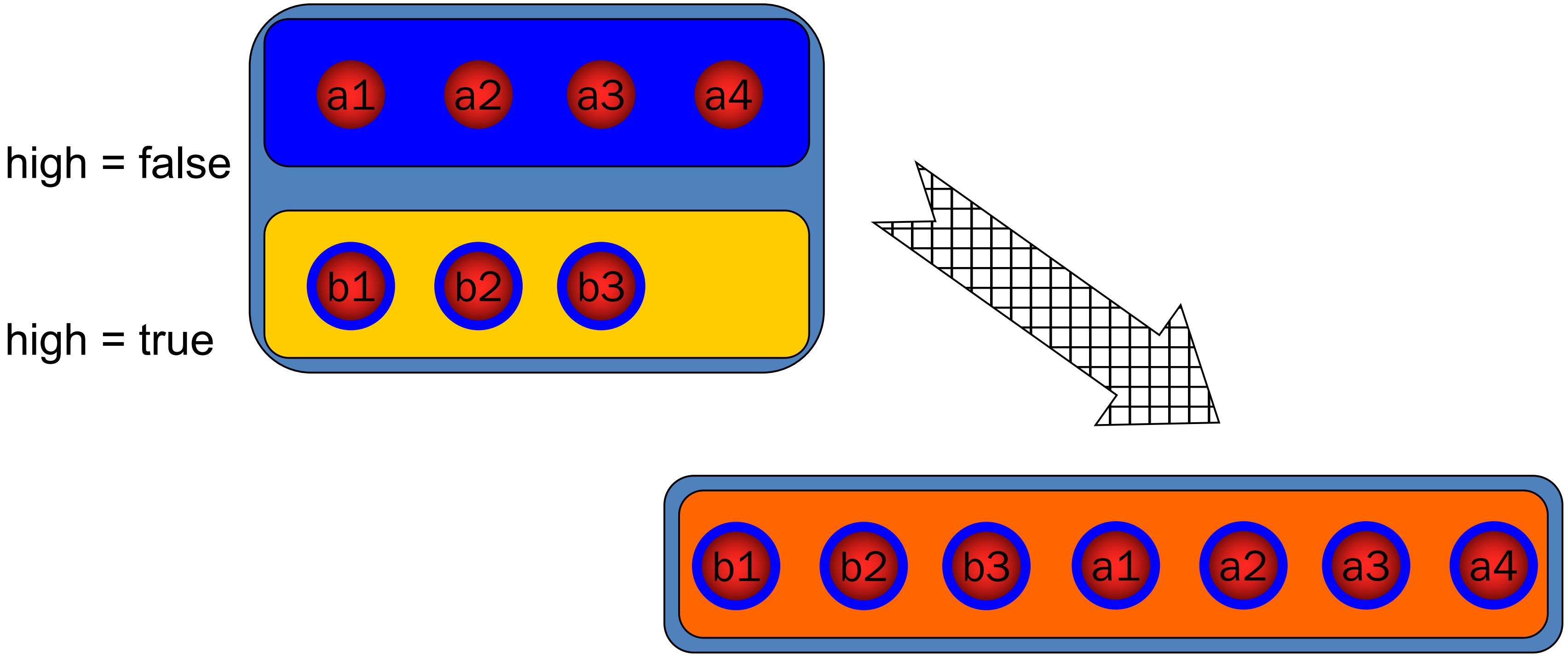
If not, make the spawnees abort with MPI_ERR_REVOKE

See 9.respawn

We need to check if spawn succeeded before proceeding further…

Intercommunicators

- MPI_INTERCOMM_MERGE(intercomm, high, intracomm)
  - Create an intracomm from the union of the two groups
  - The order of processes in the union respect the original one
  - The high argument is used to decide which group will be first (rank 0)



Respawn 3/3

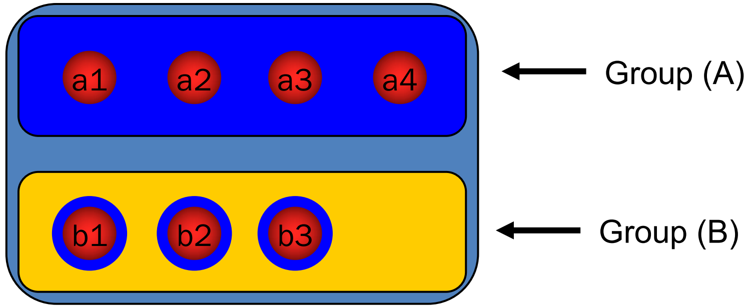
```
    /* Merge the intercomm, to reconstruct an intracomm (we check
     * that this operation worked before we proceed further) */
    rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
    rflag = flag = (MPI_SUCCESS==rc);
    MPIX_Comm_agree(scomm, &flag);
    if( MPI_COMM_WORLD != scomm ) MPI_Comm_free(&scomm);
    MPIX_Comm_agree(icomm, &rflag);
    MPI_Comm_free(&icomm);
    if( !(flag && rflag) ) {
…
        goto redo;
    }
```

Merge the icomm
We are back with an intra

Verify that icomm_mege worked takes 2 agreements

See 10.respawn

- First agree on the local group (a's know about flag provided by a's)
- Second agree on the remote group (a's know about flag provided by b's)
- At the end, all know if both flag and rflag (flag on the remote side) is good



Copy an errhandler

```
/* restore the error handler */
if( MPI_COMM_NULL != comm ) {
    MPI_Errhandler errh;
    MPI_Comm_get_errhandler( comm, &errh );
    MPI_Comm_set_errhandler( *newcomm, errh );
}
```

See 10.respawn

- In the old world, *newcomm* should have the same error handler as *comm*
  - We can copy the errhandler function ☺
  - New spawns do have to set the error handler explicitly (no old *comm* to compy it from)

Rank Reordering

```
        /* remembering the former rank: we will reassign the same
         * ranks in the new world. */
        MPI_Comm_rank(comm, &crank);
        MPI_Comm_rank(scomm, &srank);
        /* the rank 0 in the scomm comm is going to determine the
         * ranks at which the spares need to be inserted. */
        if(0 == srank) {
            /* getting the group of dead processes:
             *   those in comm, but not in scomm are the deads */
            MPI_Comm_group(comm, &cgrp);
            MPI_Comm_group(scomm, &sgrp);
            MPI_Group_difference(cgrp, sgrp, &dgrp);
            /* Computing the rank assignment for the newly inserted spares
*/
            for(i=0; i<nd; i++) {
                MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
                /* sending their new assignment to all new procs */
                MPI_Send(&drank, 1, MPI_INT, i, 1, icomm);
            }
```

See 11.respawn_reorder

Working with spares

- First use case:
  - We deploy with mpirun –np p*q+s, where s is extra processes for recovery
  - Upon failure, spare processes join the work communicator

Split the spares out of "world", the work communicator

```c
/* Let's create an initial world, a copy of MPI_COMM_WORLD w/o
 * the spare processes */
spare = (rank>np-SPARES-1)? MPI_UNDEFINED : 1;
MPI_Comm_split( MPI_COMM_WORLD, spare, rank, &world );

/* Spare process go wait until we need them */
if( MPI_COMM_NULL == world ) {
    do {
        MPIX_Comm_replace( MPI_COMM_WORLD, MPI_COMM_NULL, &world );
    } while(MPI_COMM_NULL == world );
    MPI_Comm_size( world, &wnp );
    MPI_Comm_rank( world, &wrank );
    goto joinwork;
}
```

We will define (ourselves) MPIX_Comm_replace, a function that fix the world

See ex3.0.shrinkspares.c

Working with spares

```
int MPIX_Comm_replace(MPI_Comm worldwspares, MPI_Comm comm, MPI_Comm
*newcomm) {
…
    /* First: remove dead processes */
    MPIX_Comm_shrink(worldwspares, &shrinked);
    /* We do not want to crash if new failures come... */
    MPI_Comm_set_errhandler( shrinked, MPI_ERRORS_RETURN );
    MPI_Comm_size(shrinked, &ns); MPI_Comm_rank(shrinked, &srank);

    if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
        /* not enough processes to continue, aborting. */
        MPI_Comm_size(comm, &nc);
        if( nc > ns ) MPI_Abort(worldwspares, MPI_ERR_PROC_FAILED);

        /* remembering the former rank: we will reassign the same
         * ranks in the new world. */
        MPI_Comm_rank(comm, &crank);
        /* >>??? is crank the same as srank ???<<< */
    } else { /* I was a spare, waiting for my new assignment */
    }
    printf("This function is incomplete! The comm is not repaired!\n");
```

Shrink MPI_COMM_WORLD

- A look at what we need to do…

See ex3.0.shrinkspares.c

Assigning ranks to spares

See ex3.1.shrinkspares_reorder.c

```
if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
…
  /* remembering the former rank: we will reassign the same
   * ranks in the new world. */
  MPI_Comm_rank(comm, &crank);

  /* the rank 0 in the shrinked comm is going to determine the
   * ranks at which the spares need to be inserted. */
  if(0 == srank) {
    /* getting the group of dead processes:
     *   those in comm, but not in shrinked are the deads */
    MPI_Comm_group(comm, &cgrp); MPI_Comm_group(shrinked, &sgrp);
    MPI_Group_difference(cgrp, sgrp, &dgrp); MPI_Group_size(dgrp, &nd);
    /* Computing the rank assignment for the newly inserted spares */
    for(i=0; i<ns-(nc-nd); i++) {
      if( i < nd ) MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
      else drank=-1; /* still a spare */
      /* sending their new assignment to all spares */
      MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
    }
…
  }
} else { /* I was a spare, waiting for my new assignment */
  MPI_Recv(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
}
```

Inserting the spares in world

```
if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
…
  /* remembering the former rank: we will reassign the same
   * ranks in the new world. */
  MPI_Comm_rank(comm, &crank);
…
      /* sending their new assignment to all spares */
      MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
…
} else { /* I was a spare, waiting for my new assignment */
  MPI_Recv(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
}
/* Split does the magic: removing spare processes and reordering ranks
 * so that all surviving processes remain at their former place */
rc = MPI_Comm_split(shrinked, crank<0?MPI_UNDEFINED:1, crank, newcomm);
…
flag = MPIX_Comm_agree(shrinked, &flag);
MPI_Comm_free(&shrinked);
if( MPI_SUCCESS != flag ) {
  if( MPI_SUCCESS == rc ) MPI_Comm_free( newcomm );
  goto redo;
}
return MPI_SUCCESS;
```

Send, Recv or Split could have failed due to new failures…
If any new failure, redo it all

See ex3.1.shrinkspares_reorder.c