Resilient applications using MPI-level constructs

SC'16 Fault Tolerant MPI Tutorial

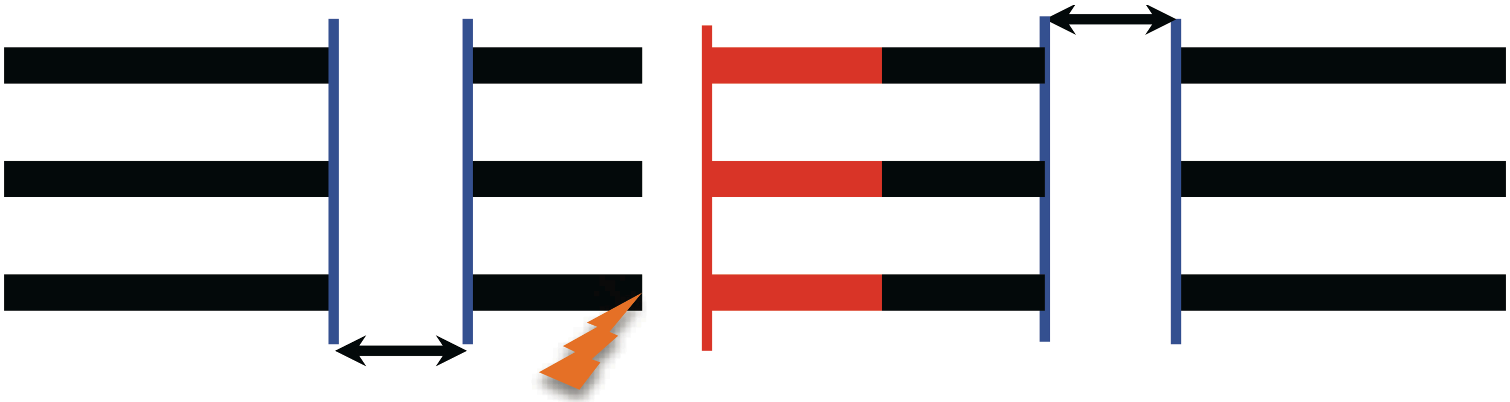
ICL UT
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

Getting the latest examples

- Slides:http://fault-tolerance.org/fault-tolerance-tutorial/sc16-tutorial/
- Examples: http://fault-tolerance.org/downloads/tutorial-sc16.tgz
  - mpirun -np 8 -am ft-enable-mpi ./my-app
- Run with ULFM-1.1 (or better) http://fault-tolerance.org/2015/11/14/ulfm-1-1-release/
- Docker Image for ULFM 1.1 http://fault-tolerance.org/downloads/docker-ftmpi.sc16tutorial.tar.xz

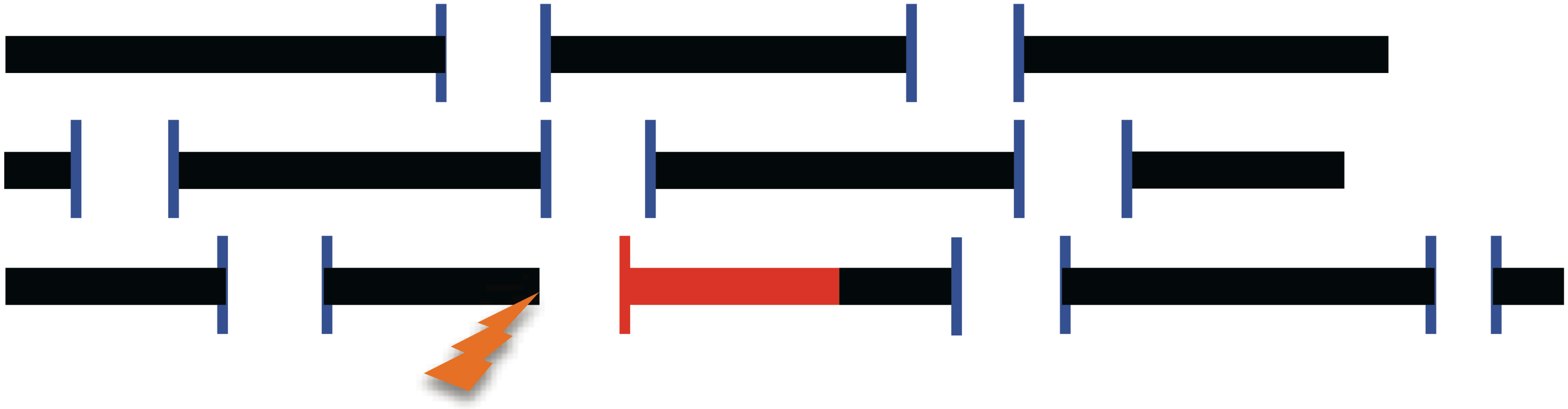
Backward recovery: C/R

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

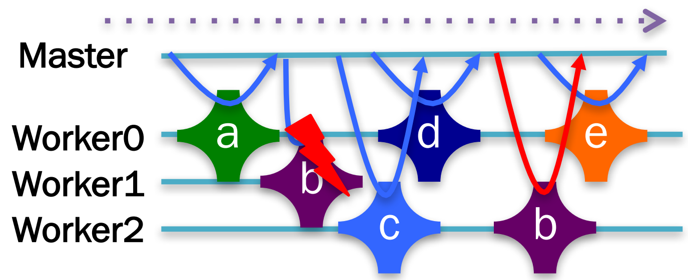
Uncoordinated C/R

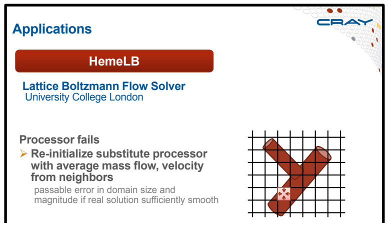


- Checkpoints taken independently
- Based on variants of Message Logging
- 1 fails, 1 rollback
- Can be implemented w/o a standardized user API
- Benefit from ULFM: implementation becomes portable across multiple MPI libraries

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) ☹





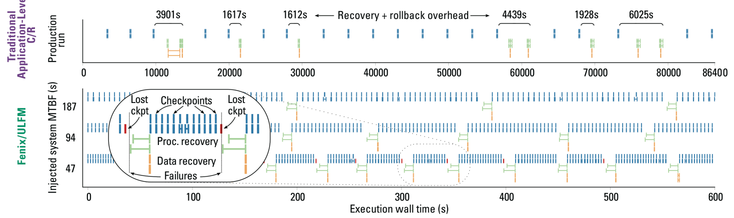
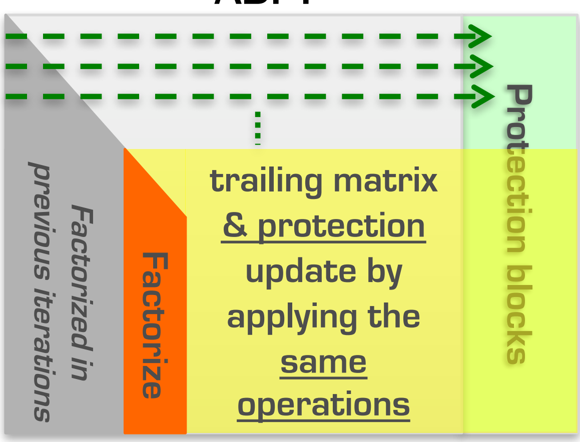


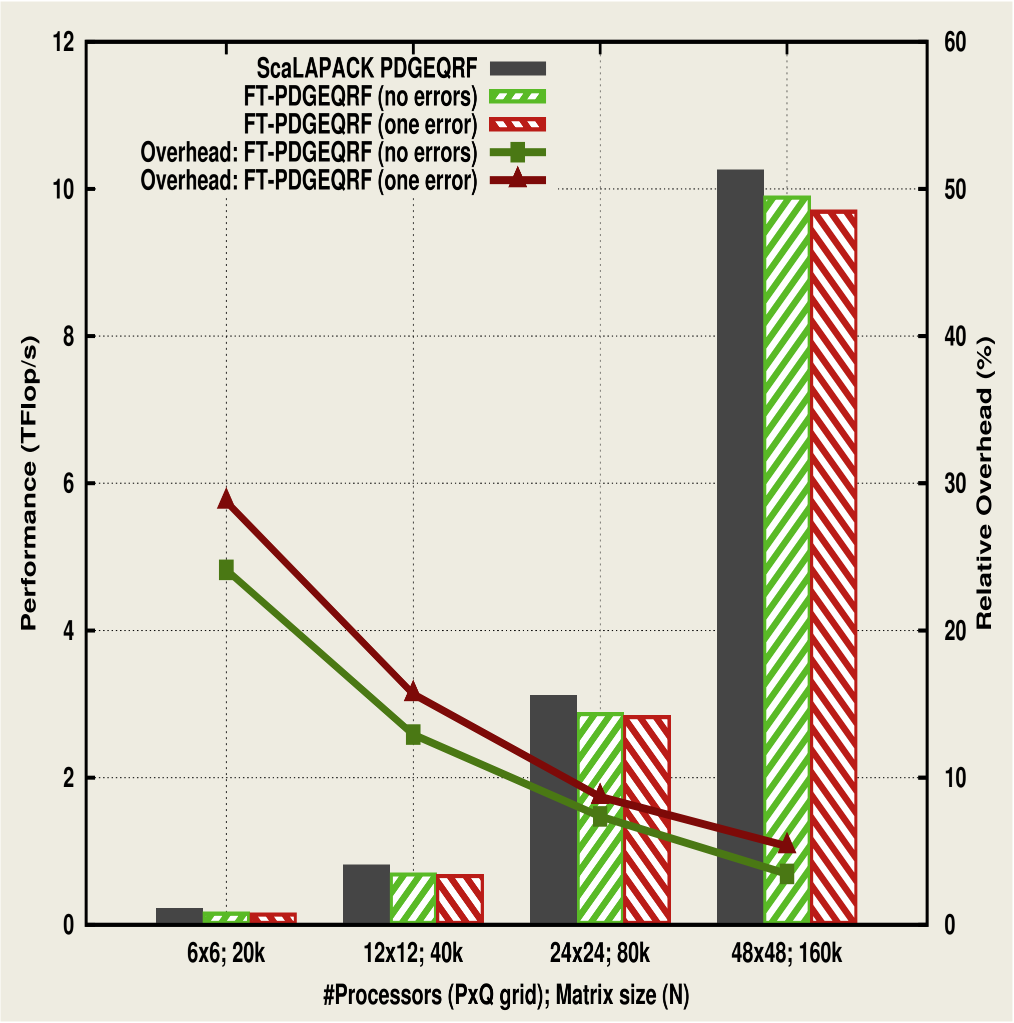
Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

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





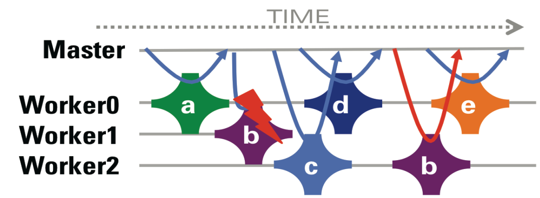
An API for diverse FT approaches

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.

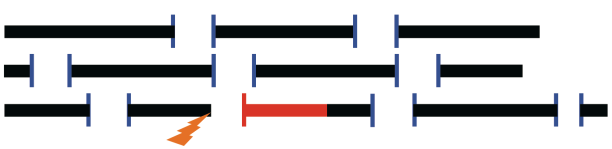
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

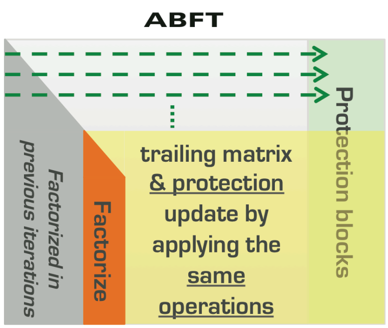
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

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

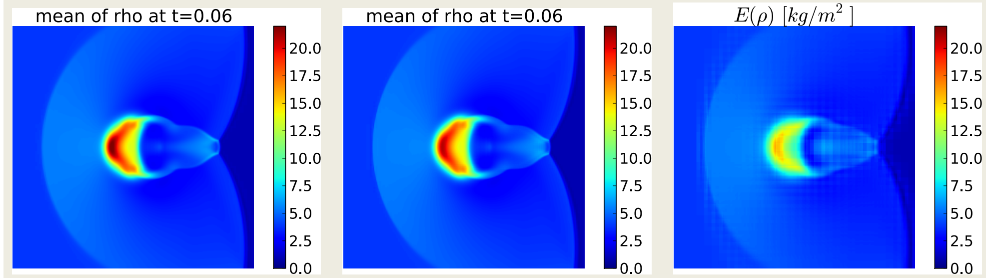


User Level Failure Mitigation: a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures

Credits: ETH Zurich

ULFM-based Applications

- **ORNL**: Molecular Dynamic simulation, C/R in memory with Shrink
- **UAB**: transactional FT programming model
- **Tsukuba**: Phalanx Master-worker framework
- **Georgia University**: Wang Landau Polymer Freezing and Collapse, localized subdomain C/R restart
- **Sandia, INRIA, Cray**: PDE sparse solver
- **Cray**: CREST miniapps, PDE solver Schwartz, PPStee (Mesh, automotive), HemeLB (Lattice Boltzmann)
- **ETH Zurich**: Monte-Carlo, on failure the global communicator (that contains spares) is shrunk, ranks reordered to recreate the same domain decomposition
- ...



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

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

FRAMEWORKS USING ULFM

LFLR, FENIX, FTLA, Falanx

S3D - Fenix

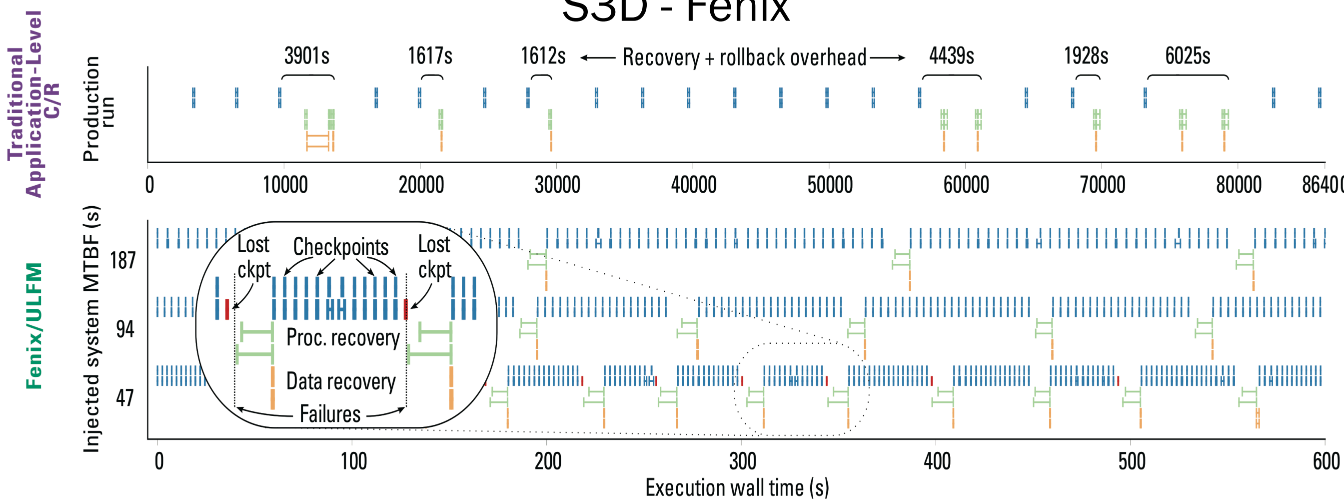


Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

Credits: ETH Zurich

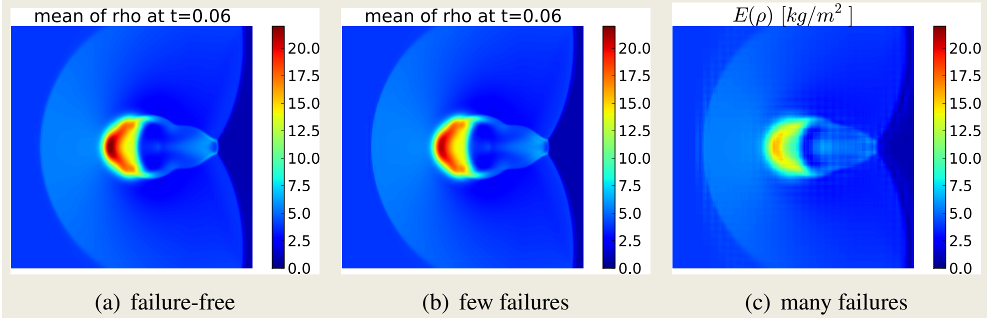


Figure 5. Results of the FT-MLMC implementation for three different failure scenarios.

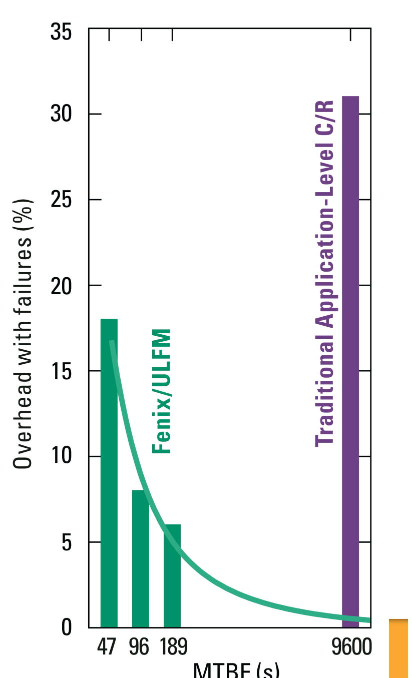
ULFM-based Applications

- **ORNL**: Molecular Dynamic simulation, C/R in memory with Shrink
- **UAB**: transactional FT programming model
- **Tsukuba**: Phalanx Master-worker framework
- **Georgia University**: Wang Landau Polymer Freezing and Collapse, localized subdomain C/R restart
- **Sandia, INRIA, Cray:** PDE sparse solver
- **Cray**: CREST miniapps, PDE solver Schwartz, PPStee (Mesh, automotive), HemeLB (Lattice Boltzmann)
- **ETH Zurich**: Monte-Carlo, on failure the global communicator (that contains spares) is shrunk, ranks reordered to recreate the same domain decomposition
- Programming Model Resilient X10 ...

FRAMEWORKS USING ULFM

LFLR, FENIX, FTLA, Falanx

S3D - Fenix



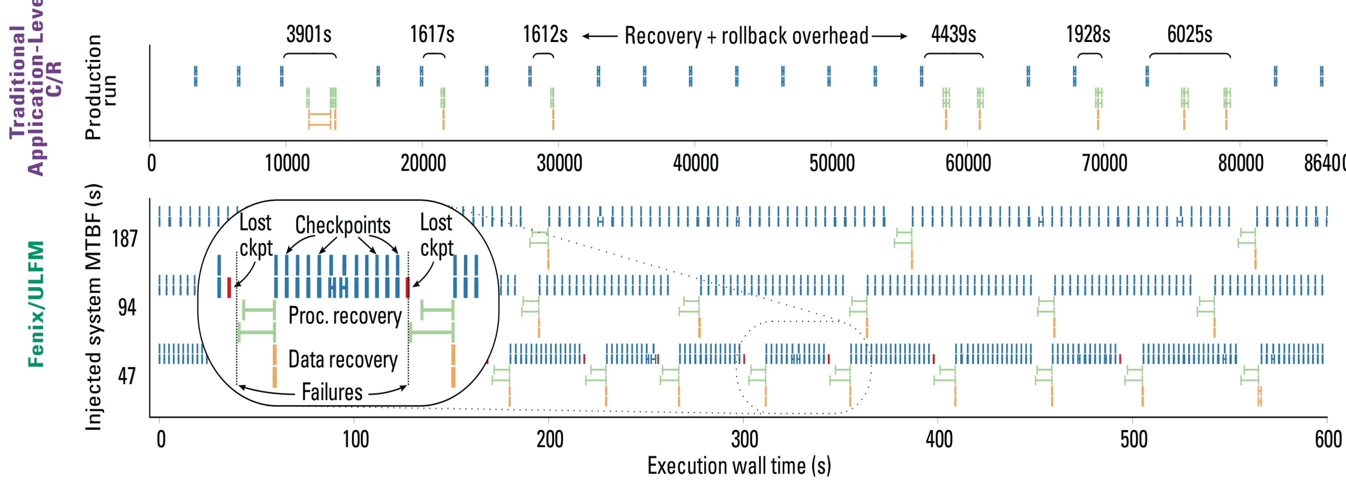


Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

Part rationale, part examples

ULFM MPI API

What is the status of FT in MPI today?

- 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

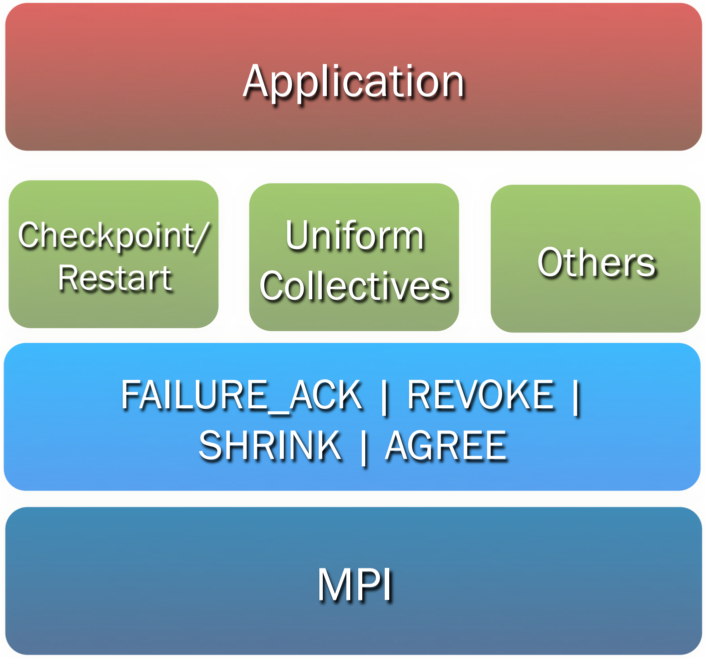
- MPI_Comm_create_errhandler(errh, errhandler_fct)
  - Declare an error handler with the MPI library
- MPI_Comm_set_errhandler(comm, errh)
  - Attach a declared error handler to a communicator
  - Newly created communicators inherits the error handler that is associated with their parent
  - Predefined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

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

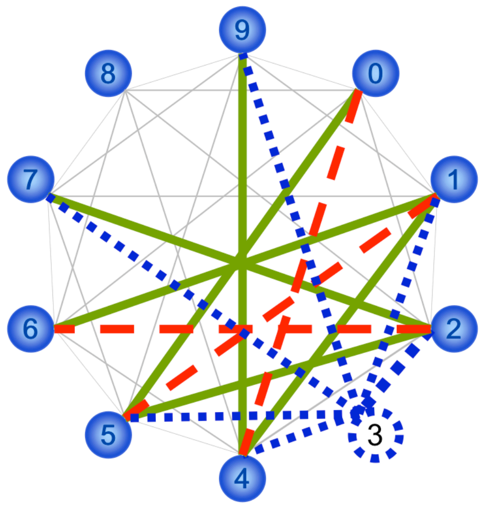


Not all recovery strategies require all of these features, that's why the interface splits notification, propagation and recovery.

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

Failure Notification

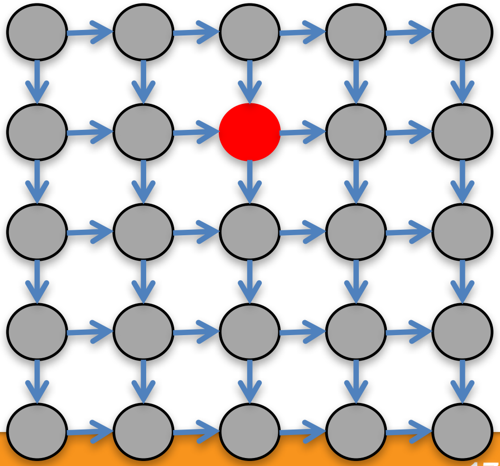
- MPI stands for scalable parallel applications it would be unreasonable to expect full connectivity between all peers
- The failure detection and notification should have a neighboring scope: only processes involved in a communication with the failed process might detect the failure



- 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?

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

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

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_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

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

Failure Discovery

- 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_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)

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

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

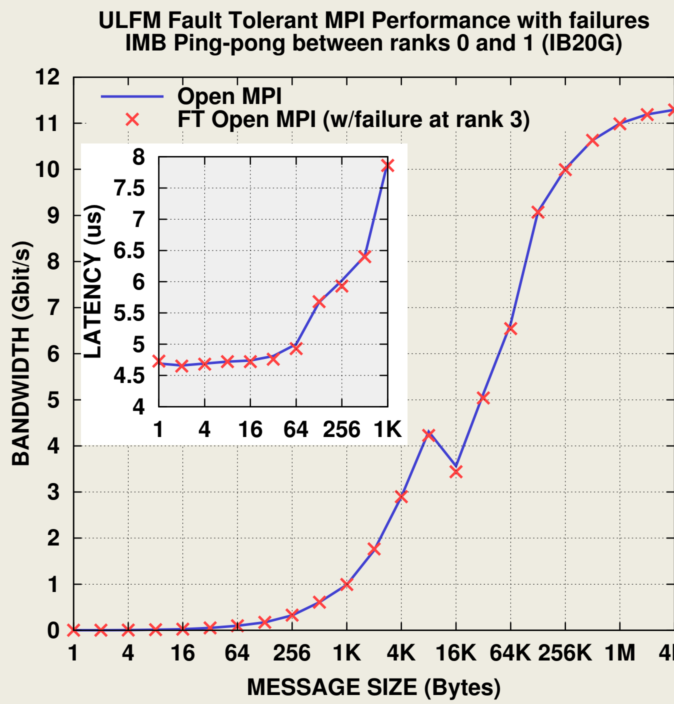
Other mechanisms

- 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.

ULFM MPI: Software Infrastructure

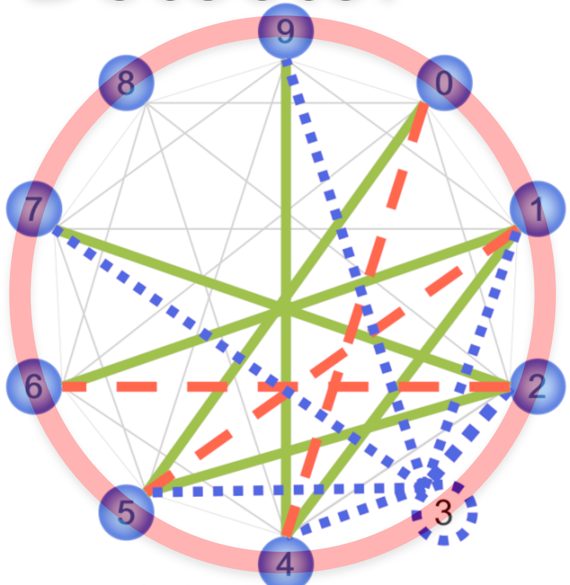
- Implementation in Open MPI available
  - ANL working on MPICH implementation, close to release
- Very good performance w/o failures
- Optimization and performance improvements of critical recovery routines are close to release
  - New revoke
  - New Agreement

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

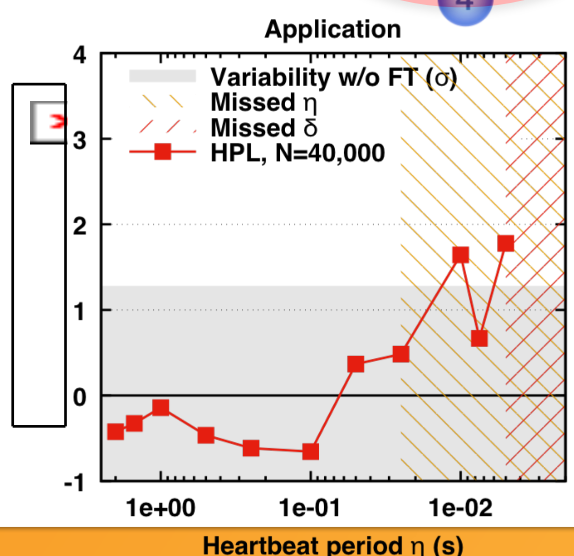
Stabilization Time T(f) = duration of the longest sequence of non stable configurations assuming at most f overlapping faults

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

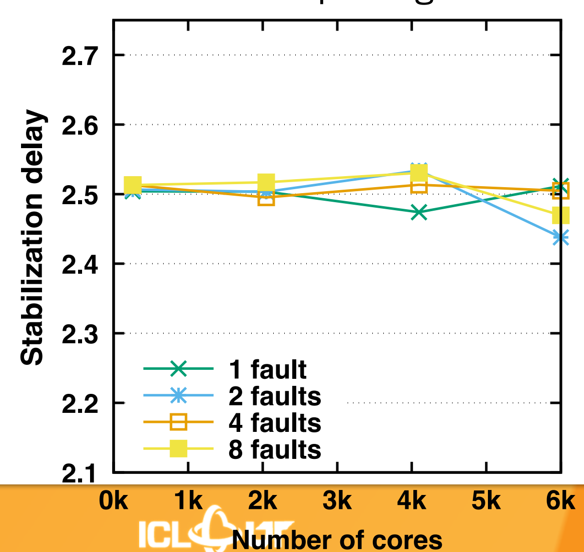
$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{\text{reconnect}} + \underbrace{\frac{f(f+1)}{2} B(n)}_{\text{propagate}}$$

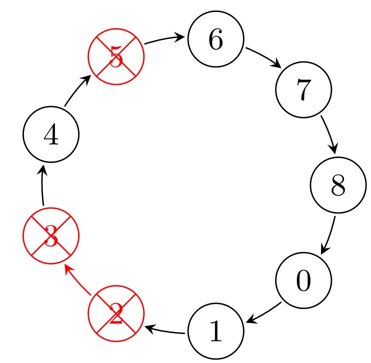
The broadcast algorithm can tolerate up to $\lfloor log(n) \rfloor$ overlapping failures, thus

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



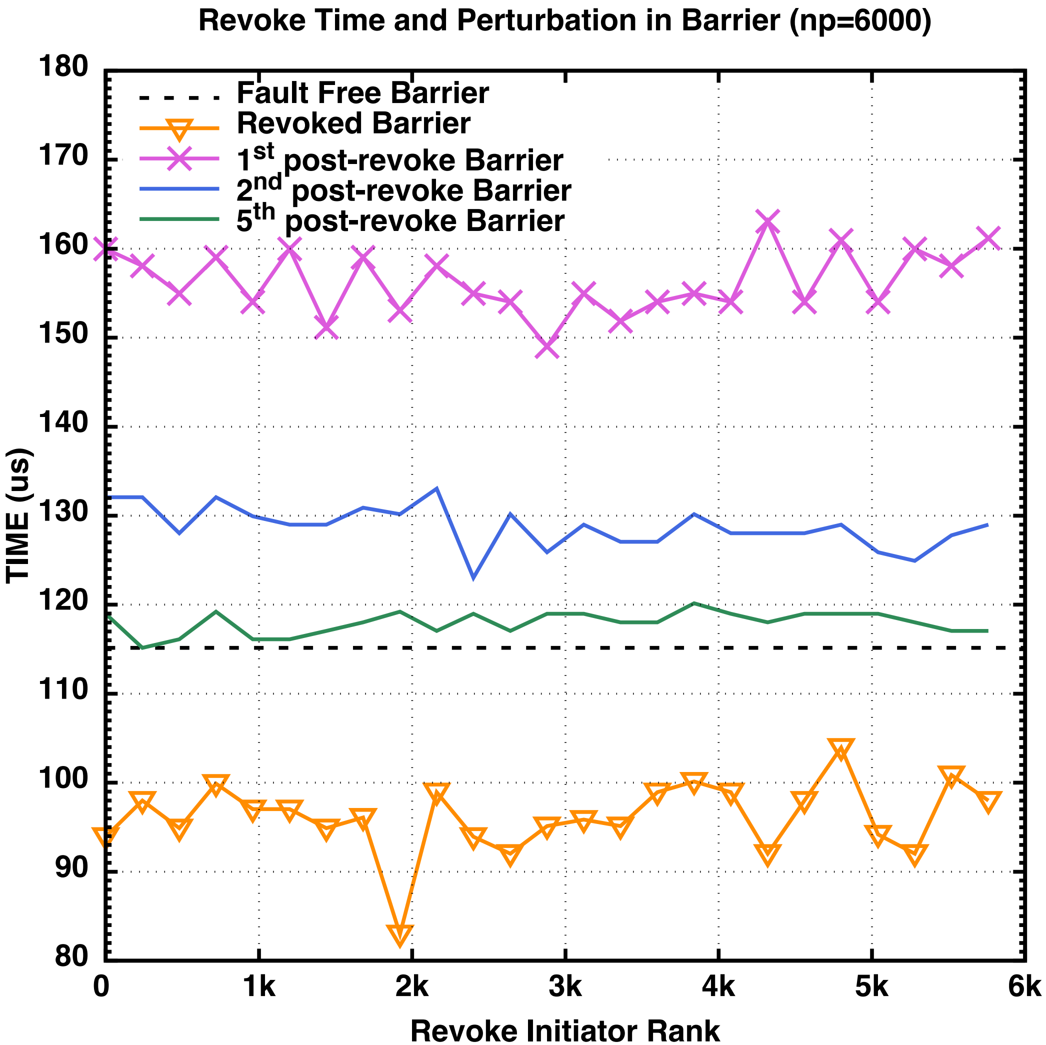
Timeout for suspecting a failure 2.5s





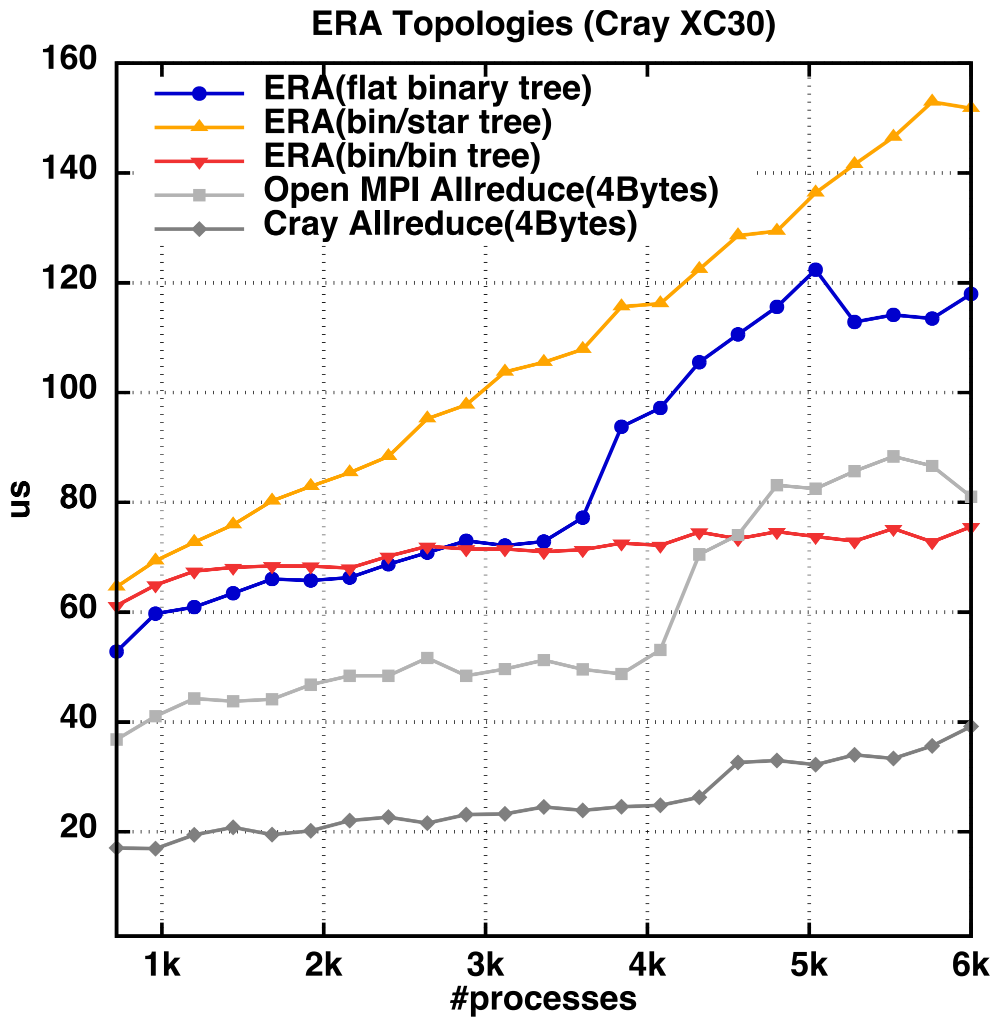
Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J. "Failure Detection and Propagation in HPC systems," SuperComputing, Salt Lake City, UT, November, 2016

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.

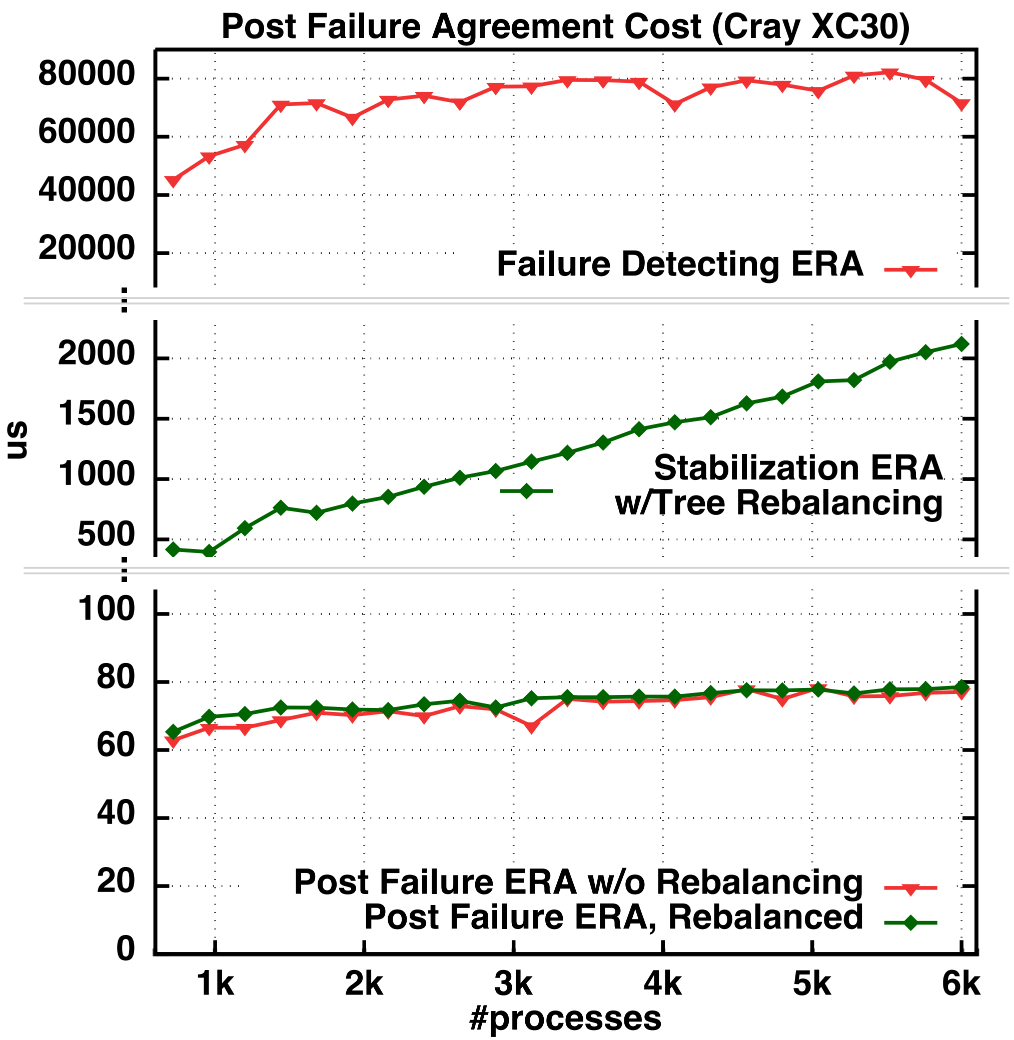
Scalable Agreement



- 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

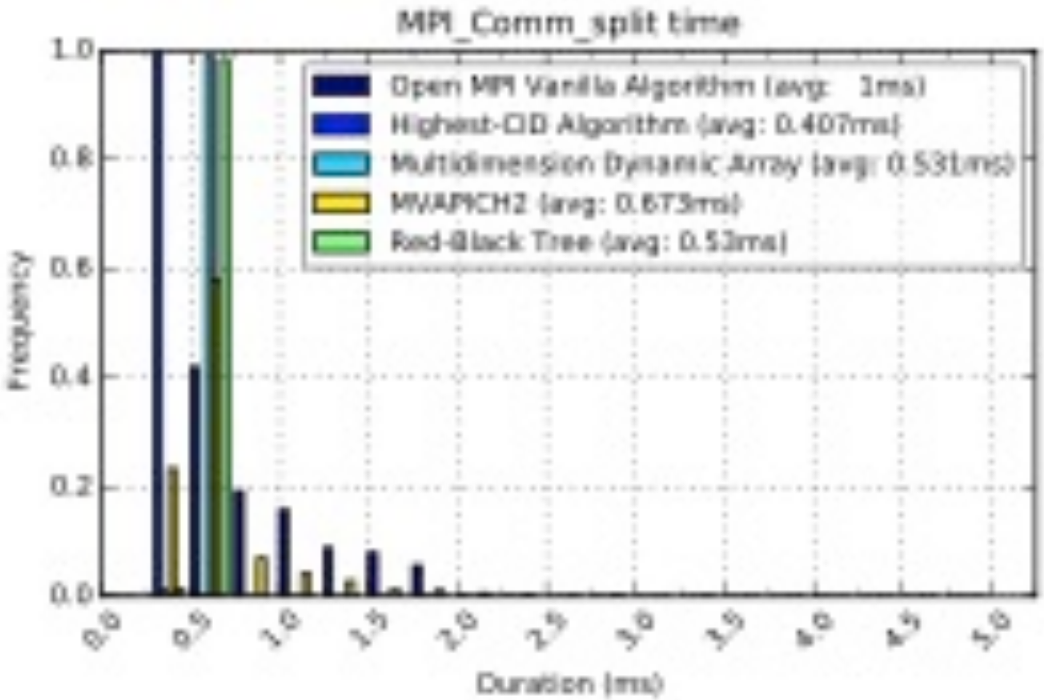
NICS Darter (Cray XC30)

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
- The optional rebalancing step is not justified until the topology degenerates enough to need it.

Scalable CID Allocation

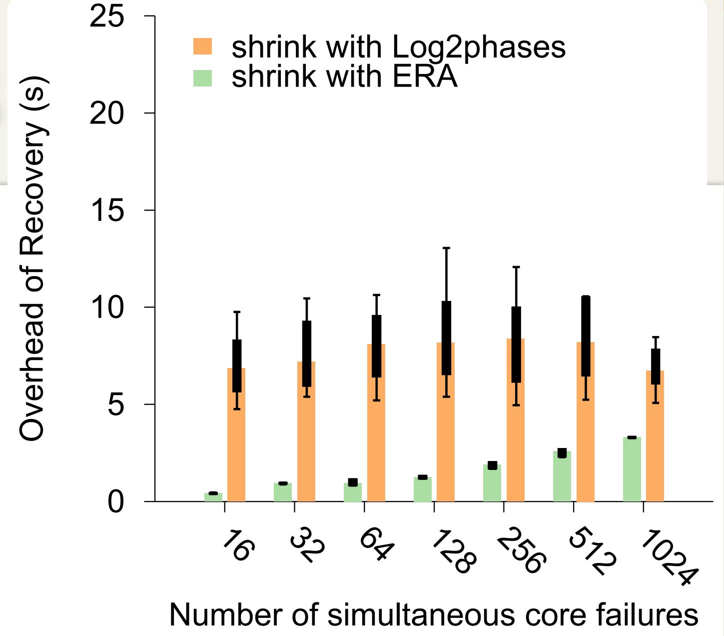


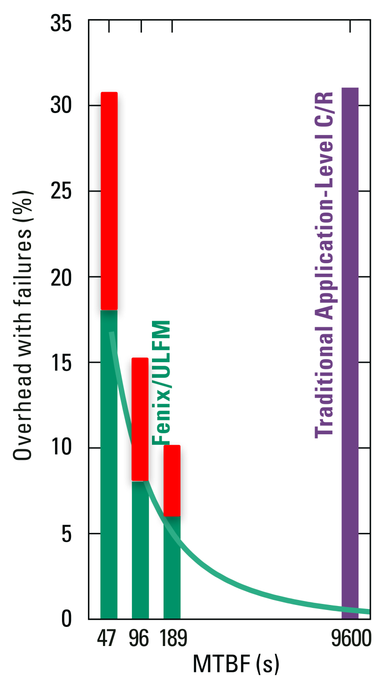
- Typical communicator recreation usage in ULFM: MPI_COMM_SPLIT to replace the processes in the same order as originally
- 128 max processes per communicator

- Default Open MPI algorithm loops round an MPI_ALLREDUCE operations using the smallest CID available. MVAPICH does a bit-array reduction by limiting the number of available communicators to 2048.
- Highest-CID loops using the maximum used CID instead
- Guarantees a CID allocation in 1 step if not multi-threading conflicts, but the sparsity of the CID might be problematic
- Different CID storage algorithms: A red-black tree and a 4-byte multidimensional array

Impact on Applications

- “Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems” – Tue 4:30PM Room 18CD





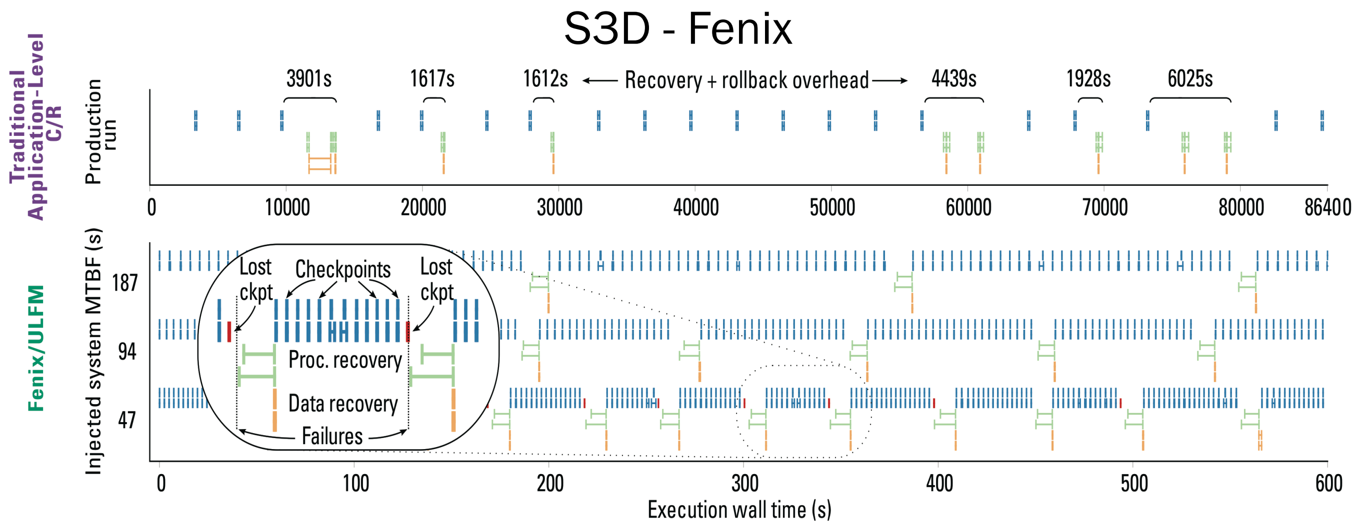


Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

Part rationale, part examples

ULFM MPI API

Hands On: Fault Tolerant MPI with ULFM

Aurelien Bouteiller @SC16

A failure, you say?

Installing and using the Docker

- ULFM Open MPI branch packaged in a docker

1. Install Docker: You can install Docker quickly, either by downloading one of the official builds from http://docker.io for MacOS and Windows, or by installing Docker from your Linux or MAcOS package manager (i.e. yum install docker, apt-get docker-io, brew/port install docker-io).
2. In a terminal, Run `$ docker hello-world` to verify that the docker installation works.
3. Load the pre-compiled ULFM Docker machine into your Docker installation
   ```**xzcat <ftmpi_ulfm-1.1.xz | docker load**```. On MacOS, the system provided `zcat` can handle xz archives
4. Source the docker aliases which will redirect the "make" and "mpirun" command in the local shell to execute in the Docker machine.
   ```**sh$. dockervars.sh load**```.
5. Go to the tutorial examples directory. You can now type `make` to compile the examples using the Docker provided "mpicc", and you can execute the generated examples in the Docker machine using `**mpirun -am ft-enable-mpi -np 10 example**`. Note the special `**-am ft-enable-mpi**` parameter;

- Complete examples (corrections) are name a*, incomplete examples are named [0-9]*

Bye bye, world

See 0.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

- This program will abort (default error handler)
- What do we need to do to make if fault tolerant?

Bye bye, world, but orderly

See 1.errreturns.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 2.errhandler.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 2.errhandler.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 2.errhander.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

Still in 2.errhandler.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 2.errhandler.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

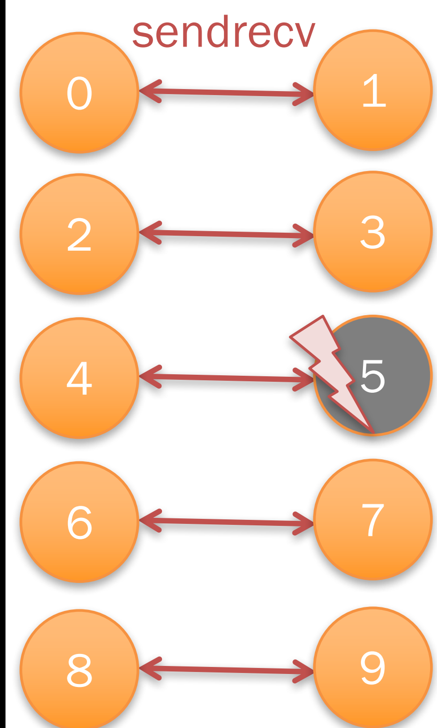
Get the group of marked failed processes

Translate the failed group member's ranks, in comm

Insulation from irrelevant failures

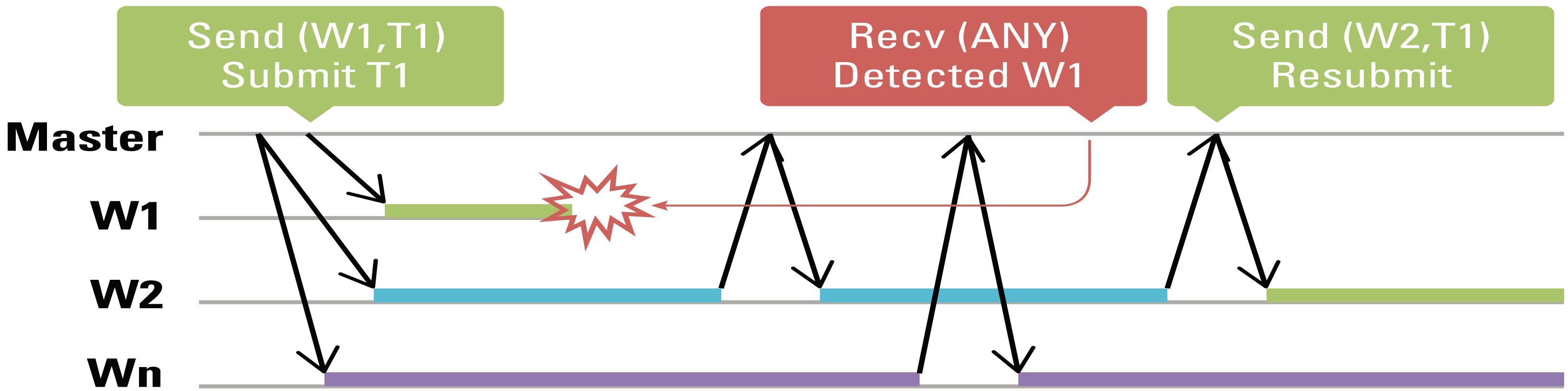
See 3.undisturbed.c

```
    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);
```



Can you guess what happens?

Continuing through errors

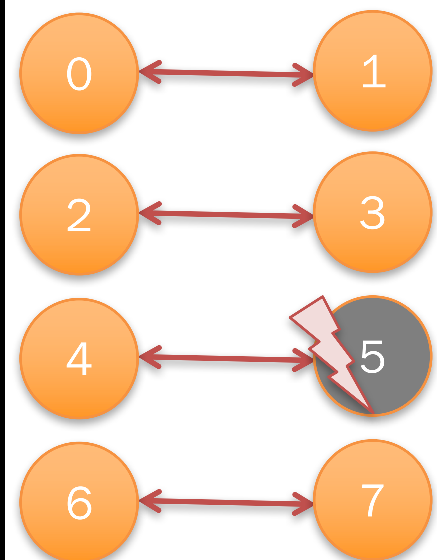


- Error notifications do not break MPI
  - App can continue to communicate on the communicator
  - More errors may be raised if the op cannot complete (typically, most collective ops are expected to fail), but p2p between non-failed processes works
- In this Master-Worker example, we can continue w/o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

Insulation from irrelevant failures

See 3.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 -am ft-enable-mpi  -np 10 ex0.5.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!

What [didn't] caused the error?

See ex0.8.recv_deadlock.c

```c
#include <mpi.h>
#include <mpi-ext.h>
… /* we use the same error handler as before */
if( rank == 0 ) {
    raise(SIGKILL);
    MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else {
    rc = MPI_Recv(&unused, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
    if( (MPI_SUCCESS == rc) && (rank < (size - 1)) )
        MPI_Send(&unused, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
}
printf("Rank %d/%d leaving (after receiving %d)\n", rank, size, unused);…
```

Assume the process dies before sending the message

MPIX_ERR_PROC_FAILED on rank 1. No further propagation of the data.

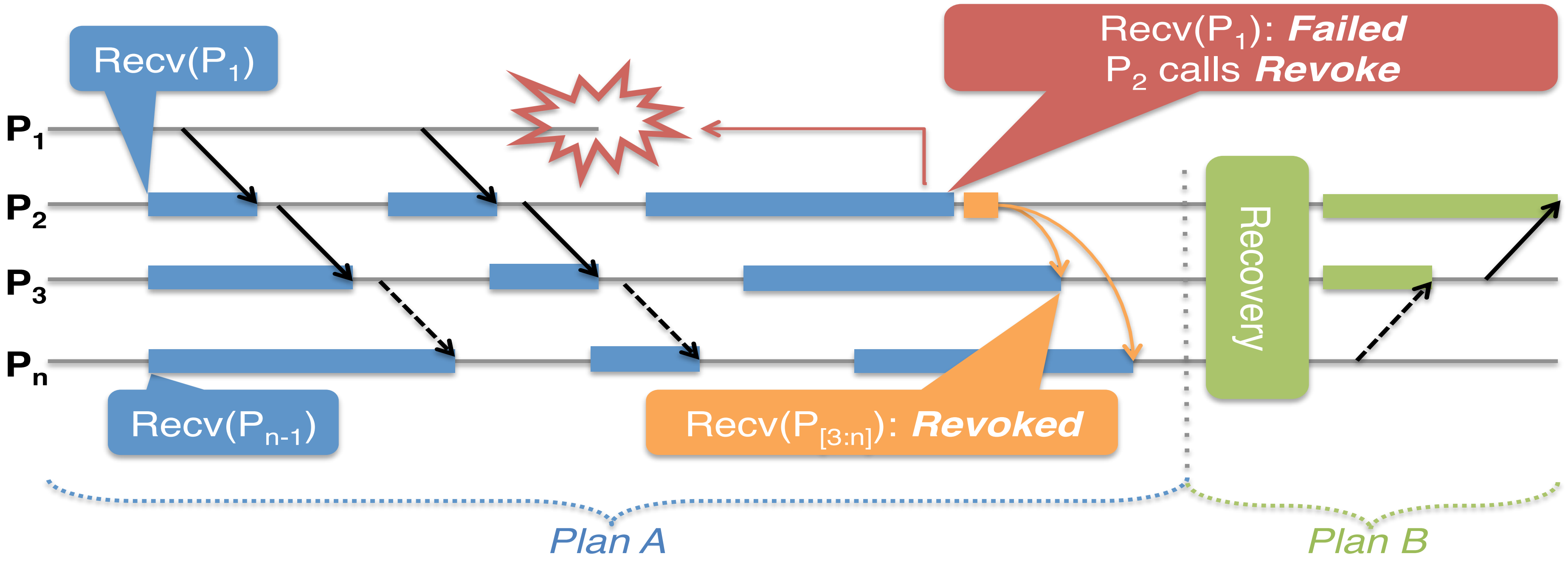
- When a sender fails
  - The corresponding receive cannot complete properly anymore
  - If we want to handle the failure, that particular recv must be interrupted

- All **MPI operations** must complete (possibly in error) when a failure prevents their normal completion
- Recv from non failed processes should complete normally

Lets keep it neat and tidy

STABILIZING AFTER AN ERROR

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

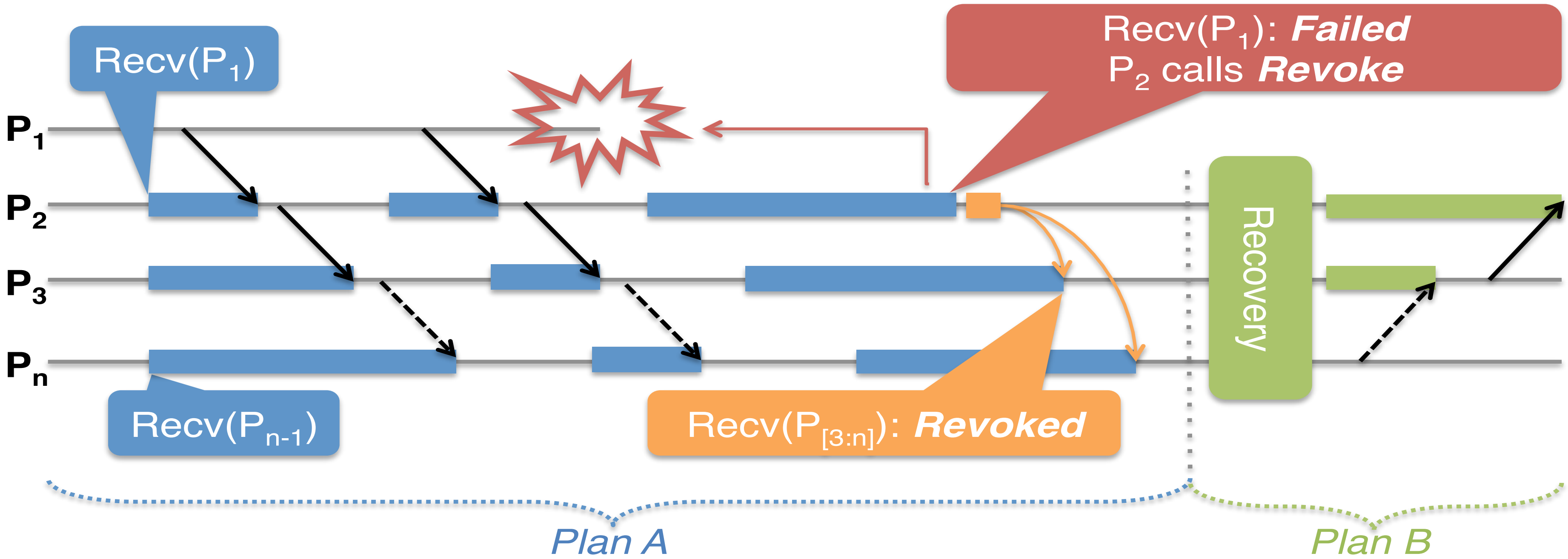
Regrouping for Plan B

See 4.iferror.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;
        }
```

- What needs to be added here to fix this program?

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 a4.iferror.c

More on non-uniform error reporting

See 5.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 ?

More on non-uniform error reporting

See 5.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 -am ft-enable-mpi -np 5 ex0.7.report_nonuniform -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!

Who caused the error

Try ex0.4.report_many

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                        errh);

if( rank > (size/2) ) raise(SIGKILL);
MPI_Barrier(MPI_COMM_WORLD);
```

Same program, but we inject more failures…

- Are all ranks going to trigger the error handler?
- For those that do, will they all print the same thing?

Who caused the error

Try ex0.4.report_many

```
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                            errh);

    if( rank > (size/2) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
```

Same program, but we inject more failures…

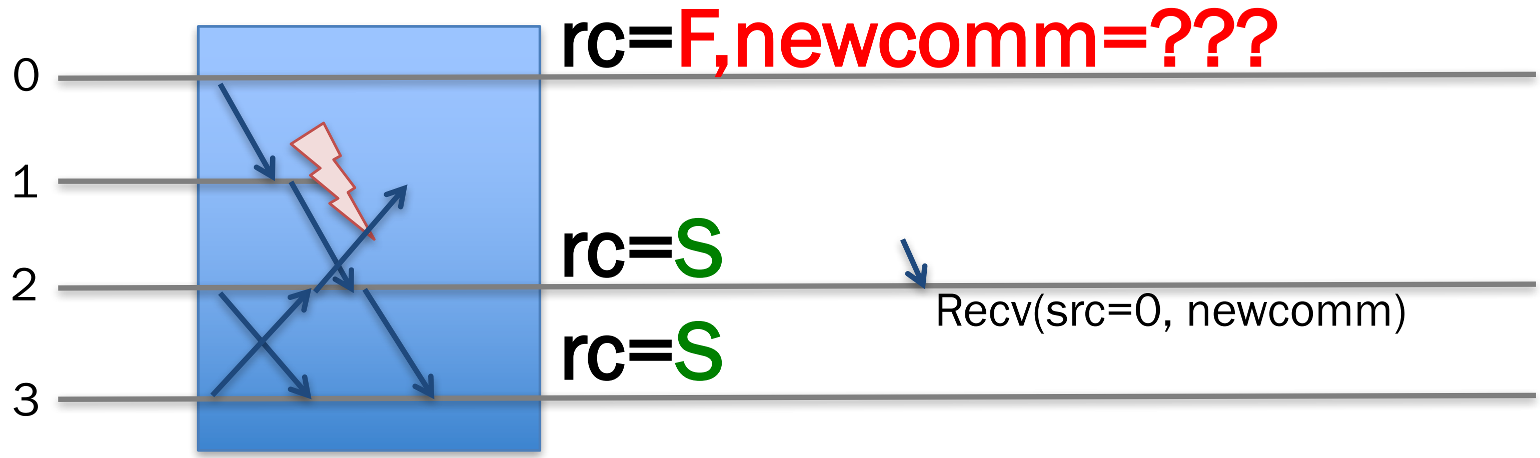
- Are all ranks going to trigger the error handler?
- For those that do, will they all print the same thing?

```
bash$ $ULFM_PREFIX/bin/mpirun -am ft-enable-mpi -np 5 ex0.4.report_many -v
Rank 2 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: {
4 }
Rank 1 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 2 found dead: {
3 4 }
Rank 0 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 2 found dead: {
4 3 }
```

All survivors reported an error, but not necessarily about the same failed ranks, or not seen in the same order

(it may take several trials to see it at small scale, at large scale, it’s almost all the time…)

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 uniformely 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 6.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

```c
/* 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;
}
```

See 7.err_comm_grid2d

- 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

Dealing with MPI_ANY_SOURCE

See 8.err_any_source.c

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

Error Insulation

See 9.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 9.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 9.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 ☺

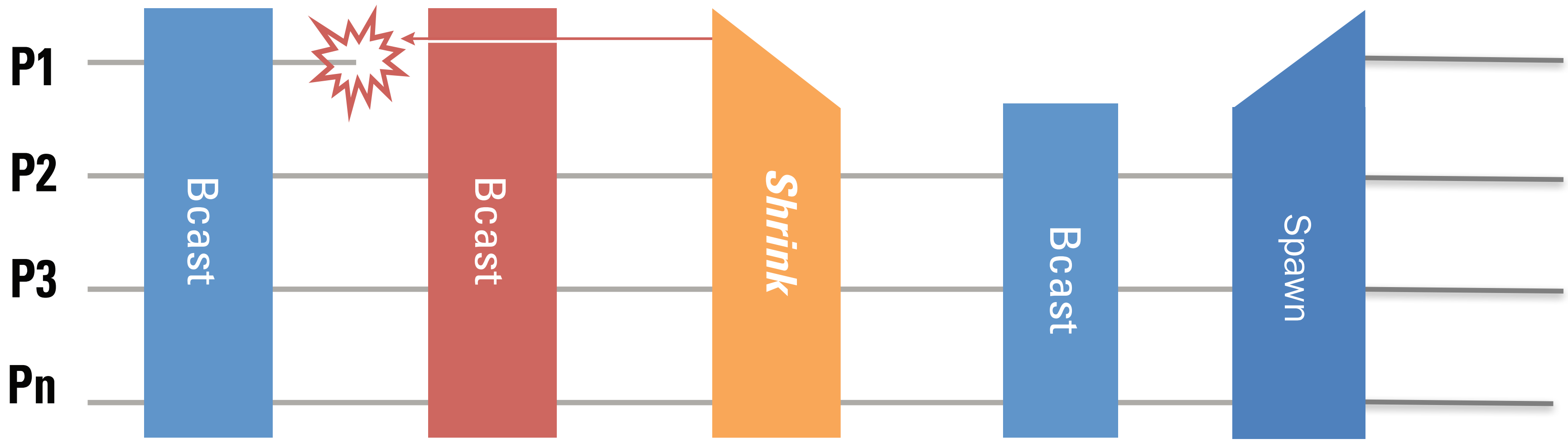
Low ranks half_comm has failed process, we free it anyway


```
$ $ULFM_PREFIX/bin/mpirun -am ft-enable-mpi -np 10 ex0.6.undisturbed2
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

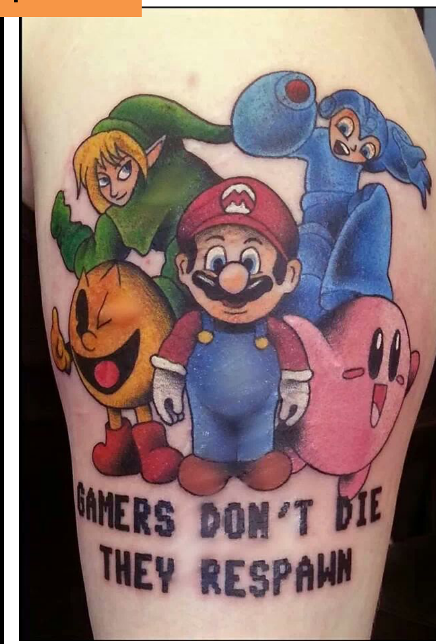


- After a Revoke, our original comm is unusable
- We can Shrink: that create a new comm, but smaller
  - Can be used to do collective and p2p operations, fully functional
- Some application need to restore a world the same size
  - And on top of it, they want the same rank mapping

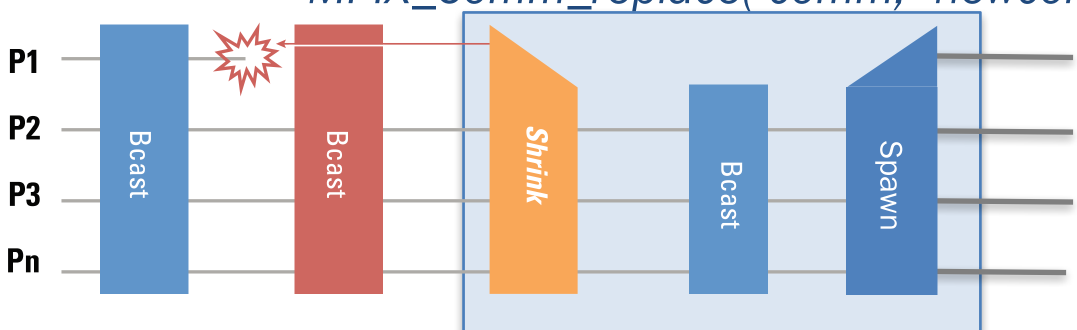
Respawning the deads

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;
  }
```



*MPIX_Comm_replace(comm,*newcomm)*



- 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);
}
```

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 succesful, rollback instead\n", rank);
        longjmp(jmpenv, 0);
    }
    ckpt_iteration = iteration;
    /* Memcopy 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

Inside MPIX_COMM_REPLACE

See 10.respawn

```
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);
```

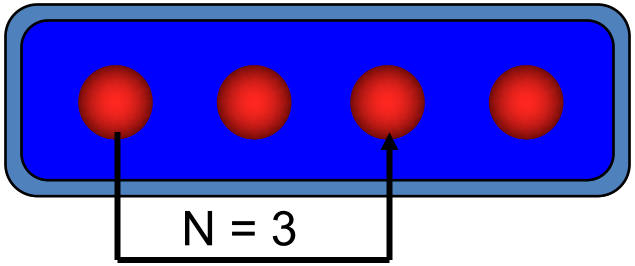
Same as in spare: new guys wait for their rank from 0 in the old world

Spawn *nd* new processes

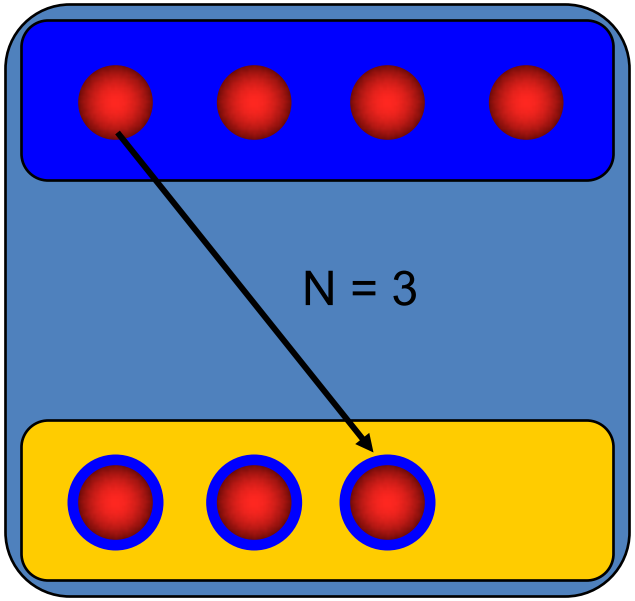
Intercommunicators – P2P

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

- Intracommunicator

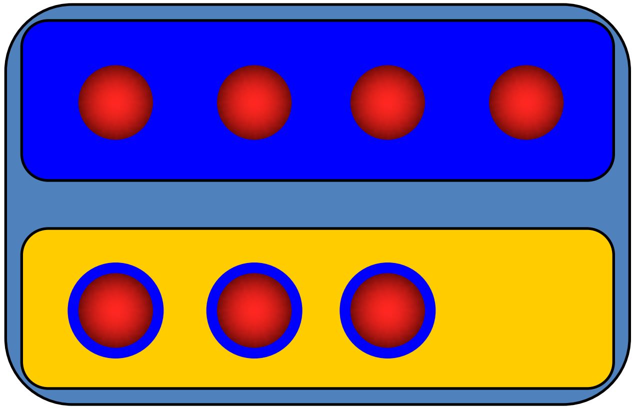


- Intercommunicator



Intercommunicators

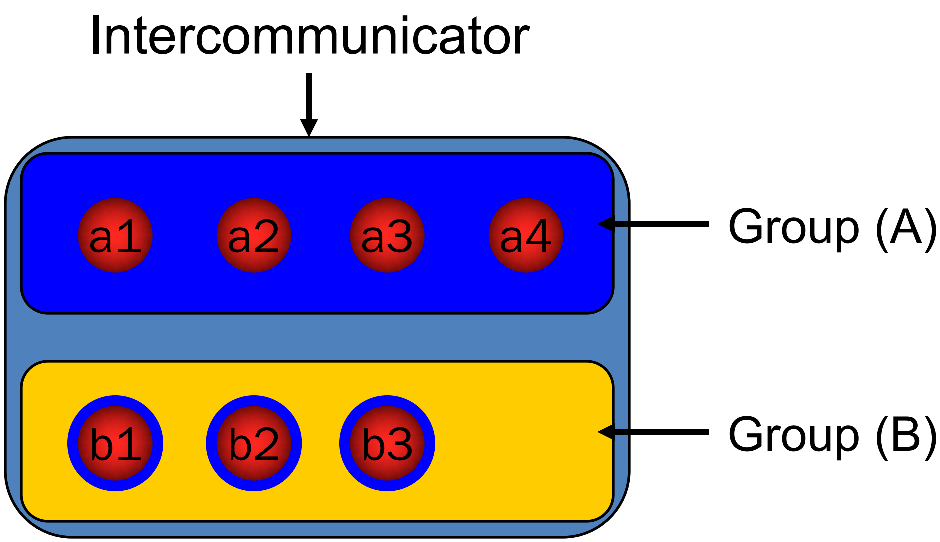
- And what's a intercommunicator ?



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

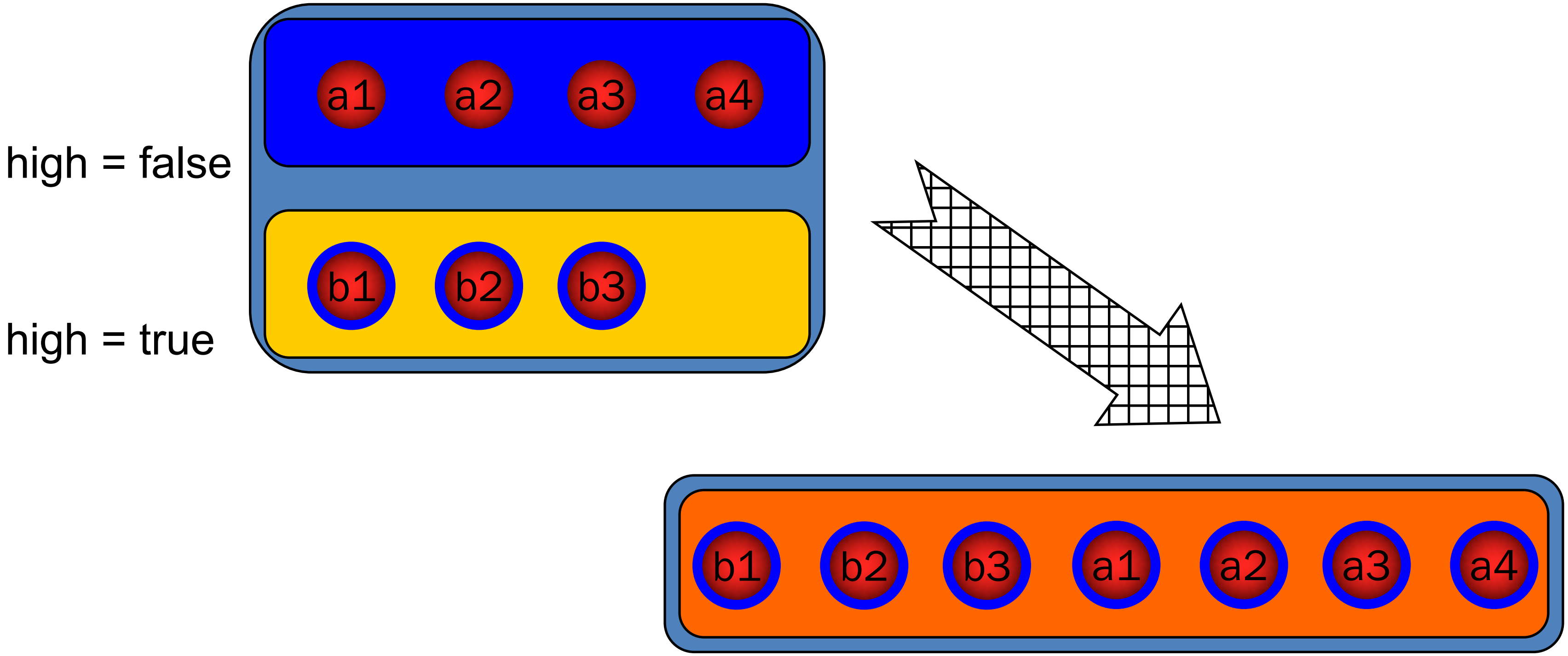
If not, make the spawnees abort with MPI_ERR_REVOKE

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

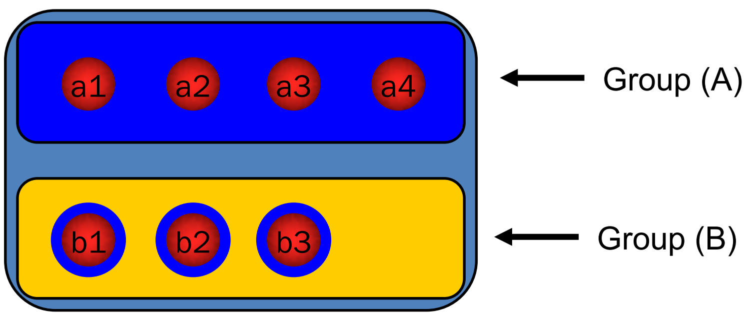
```
/* Merge the intercomm, to reconstruct an intracomm (we check
 * that this operation worked before we proceed further) */
rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
rflag = flag = (MPI_SUCCESS==rc);
MPIX_Comm_agree(scomm, &flag);
if( MPI_COMM_WORLD != scomm ) MPI_Comm_free(&scomm);
MPIX_Comm_agree(icomm,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

```c
        /* 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

Split the spares out of “world”, the work communicator

- 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

```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

```c
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

See ex3.0.shrinkspares.c

- A look at what we need to do…

Assigning ranks to spares

See ex3.1.shrinkspares_reorder.c

```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

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 */
\
    OMPI_Comm_agree(COMM, &__flag); \
    __stack_in_agree = 0; /* enable longjmp */ \
  } \
  if( 0xffffffff != __flag ) {

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

#define RAISE(COMM, EXCEPTION) \
  OMPI_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 code
below */
transaction2:
    TRY_BLOCK(newcomm, exception) {

        /* do more extremely useful work */

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

} CATCH_BLOCK(MPI_COMM_WORLD) {
    /* restore data1 for transaction 1 */
    goto transaction1;
} END_BLOCK()
```

Transaction 2

Transaction 1

- 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

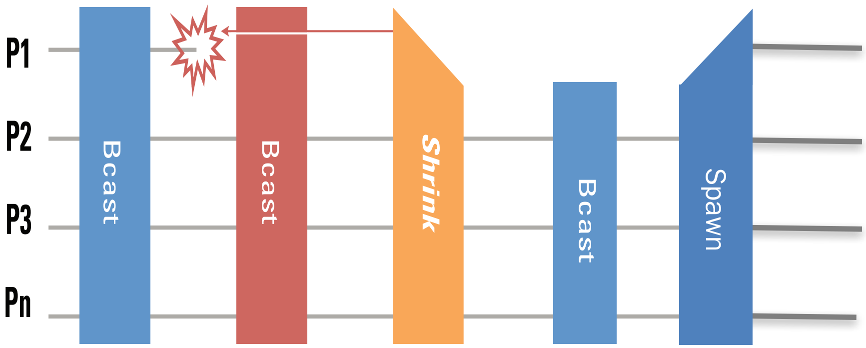
```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

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()
```

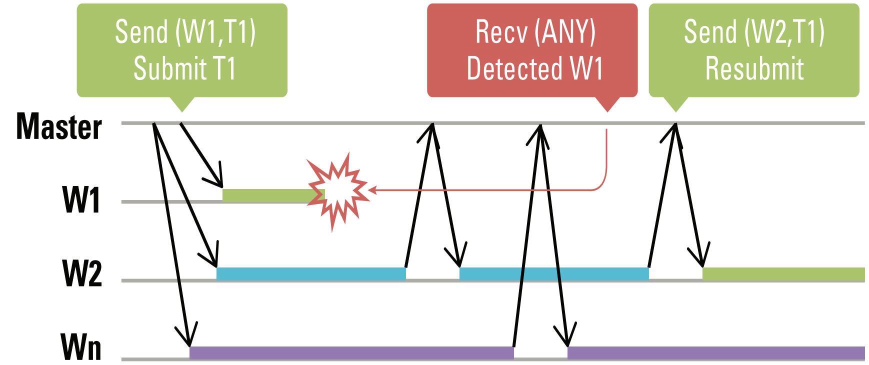
Transaction 2

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?

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)

CONCLUSION

Why all these efforts?

> **Prediction is very difficult**
>
> especially about the future
>
> Niels Bohr, Physicist - Nobel Prize Winner
>
> www.Schrijf.be - Tijdnet

Toward Exascale Computing (My Roadmap)

Based on proposed DOE roadmap with MTTI adjusted to scale linea

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

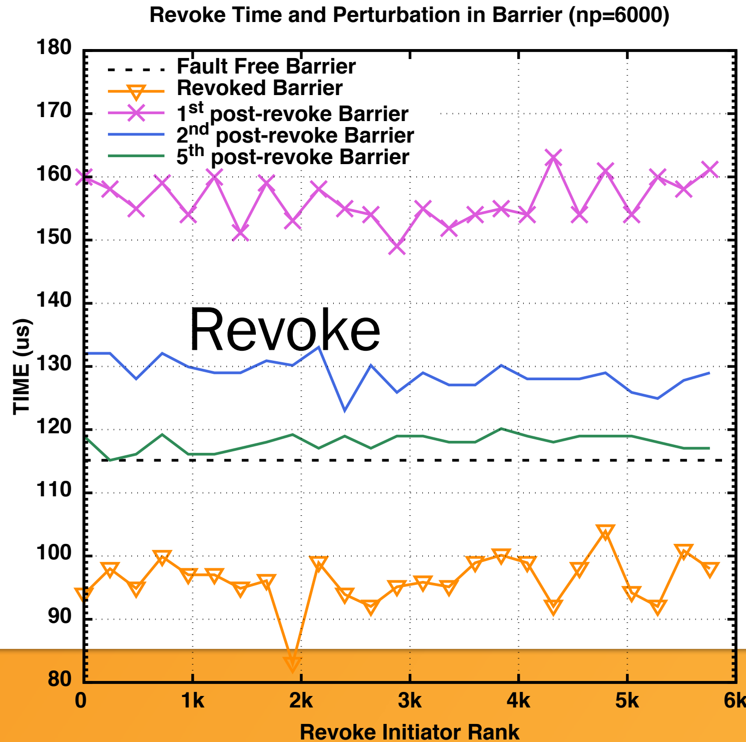
- Whatever scenario we are going for our Exascale platforms the MTBF will just keep shrinking

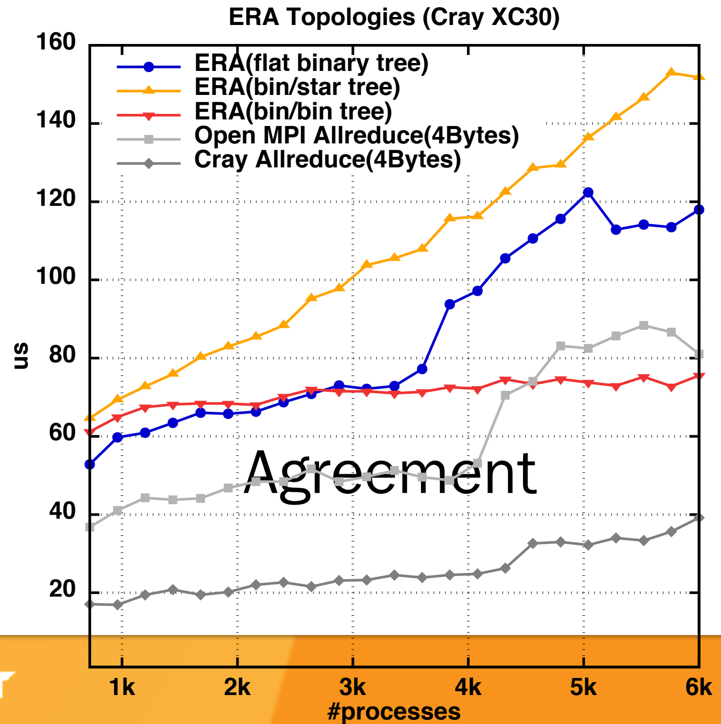
So what is the right approach

- 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 is the cost?

- Let's first look at the ULFM constructs costs
- No modification of the fault-free path in the library
- The rest depend on the application, but now you have the tools required to build the corresponding models





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

ULFM@SC'15:

TUTORIAL "Fault Tolerance for HPC: Theory and Practice" Sun. 8:30am, Room 18D

PAPER "Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems" Tue. 4:30pm, Room 18CD

TALK "Fault Tolerant MPI applications with ULFM" Wed. 2:30pm, UT NICS Booth

BOF "Fault Tolerant MPI Applications with ULFM" Wed. 3:30pm, Room 13A

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE