Application-driven Fault-Tolerance for High Performance Distributed Computing

George Bosilca, UTK
Bogdan Nicolae, ANL
Franck Cappelle, ANL
And many more

EuroPar 2018 Conference
Turin, Italy

Preparing for the hands-on

https://fault-tolerance.org/2018/08/27/europar18-tutorial/

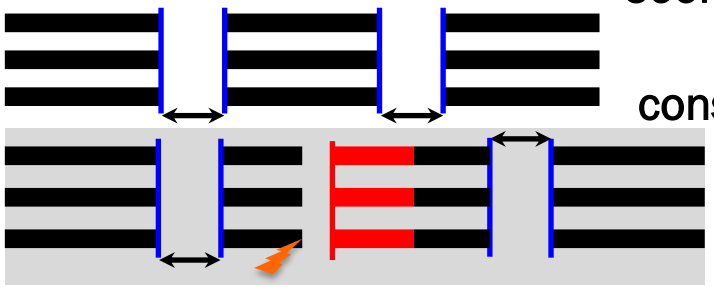
- Source code of all examples:

https://fault-tolerance.org/downloads/europar18-handson.tgz

- Run with The ULFM Docker image
  1. Install Docker (requires external download)
  2. docker pull bnicolae/veloc-tutorial (requires external download)
  3. Download source code tarball (requires external download)
  4. Open terminal and source dockervars.sh
  5. mpirun -np 10 example

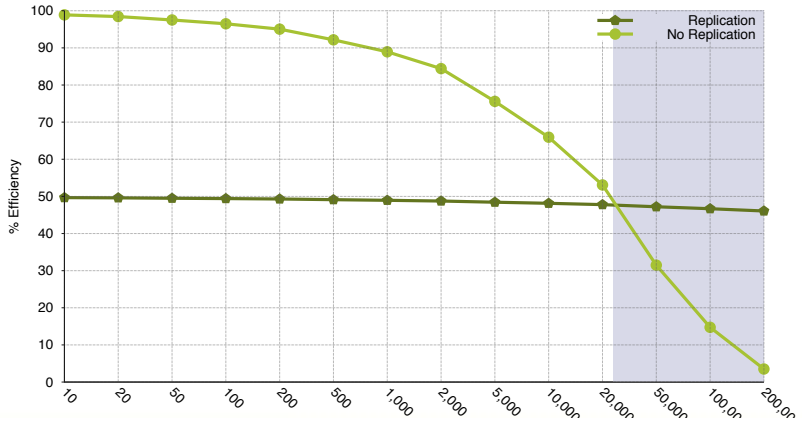
Fault Tolerance: many solutions

- Rollback Recovery
  - Not only the legacy approach
  - Checkpoint/Restart based
  - Many possible improvements (in memory, buddy, async, hierarch,...)
- Forward Recovery
  - Replication *(the only system level Forward Recovery)*
  - Master-Worker with resubmission
  - Iterative methods, Naturally fault tolerant algorithms
  - Algorithm Based Fault Tolerance



Coordinated checkpoint (with blocking, constant checkpoints)

- No checkpoint, no message logging
  - Overhead due to synchronizations between replicas
- Benefit: Possible soft error detection and recovery



Evaluating the Viability of Process Replication Reliability for Exascale Systems -- Kurt Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Patrick G. Bridges, Dorian Arnold and Rolf Riesen – SC'11

Fault Tolerance: many solutions

- Rollback Recovery
  - Not only the legacy approach
  - Checkpoint/Restart based
  - Many possible improvements (in memory, buddy, async, hierarch,…)
- Forward Recovery
  - Replication *(the only system level Forward Recovery)*
  - Master-Worker with resubmission
  - Iterative methods, Naturally fault tolerant algorithms
  - Algorithm Based Fault Tolerance

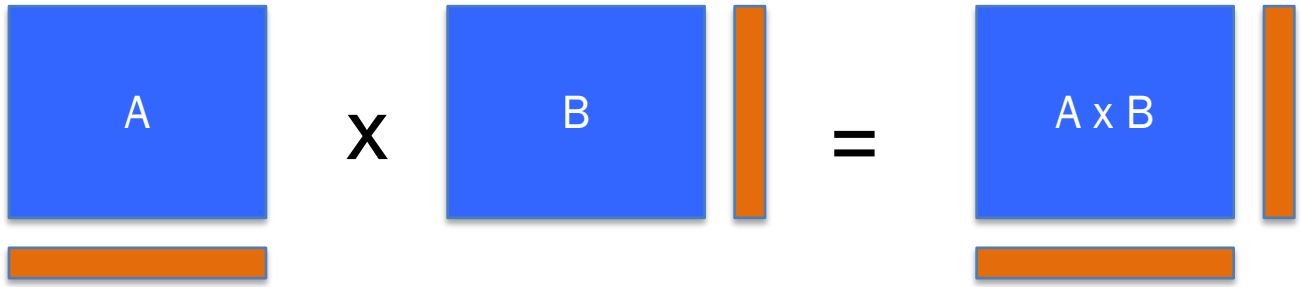
- Any technique that permit the application to continue without rollback

No checkpoint I/O overhead
No rollback, no loss of completed work
May require (sometime expensive, like replicates) protection/recovery operations, *but generally more scalable than checkpoint*
Often requires in-depths algorithm rewrite (in contrast to automatic system based C/R)

"Why is not everybody doing this already, then?"

Algorithm Based Fault Tolerance (ABFT)

- *Takes advantage of existing mathematical relationship(s)*
  - *Introduced (cheaply, if possible) by ABFT*
- *KH Huang & Jacob Abraham, ABFT for Matrix Operations, IEEE Trans. Computers. 01/1984;*
- *Matrix extended to contain additional information.*
  - *Extra column or row contains checksum.*
- *Matrix algorithm designed to operate on the data and the encoded checksum.*
  - *Checksum invariant during the course of the algorithm.*
  - *No checkpoint needed.*



$$\begin{bmatrix} A \\ G^T A \end{bmatrix} \times \begin{bmatrix} B & BG \end{bmatrix} = \begin{bmatrix} AB & ABG \\ G^T AB & G^T ABG \end{bmatrix}$$

G^T $G^T A$ *and* BG *are the check sums.*

Algorithm Based Fault Tolerance (ABFT)

- *Takes advantage of existing mathematical relationship(s)*
- *K*
  C
- *M*
  a
- *M*
  o
  e

For Dense Linear Algebra Factorizations (POTRF, QR, LU)

Memory Overhead

Matrix M x N, Blocks mb x nb,
Process grid p x q

$$O(\frac{F}{q} \times M \times N)$$

- Matrix is extended with 2F columns every q columns a

N.B. Usually F << q
Relative overheads in F/q
e.g. **2** simultaneous faults on 192x**192** process grid => 1% memory overhead

Computation Overhead

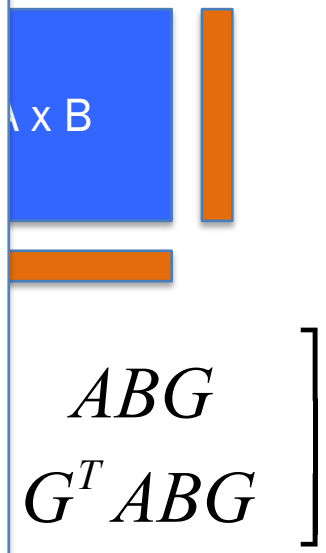
F: maximum number of ***simultaneous*** failures tolerated

$$O(\frac{F}{q} \times M^3)$$

flops for the checksum update, and

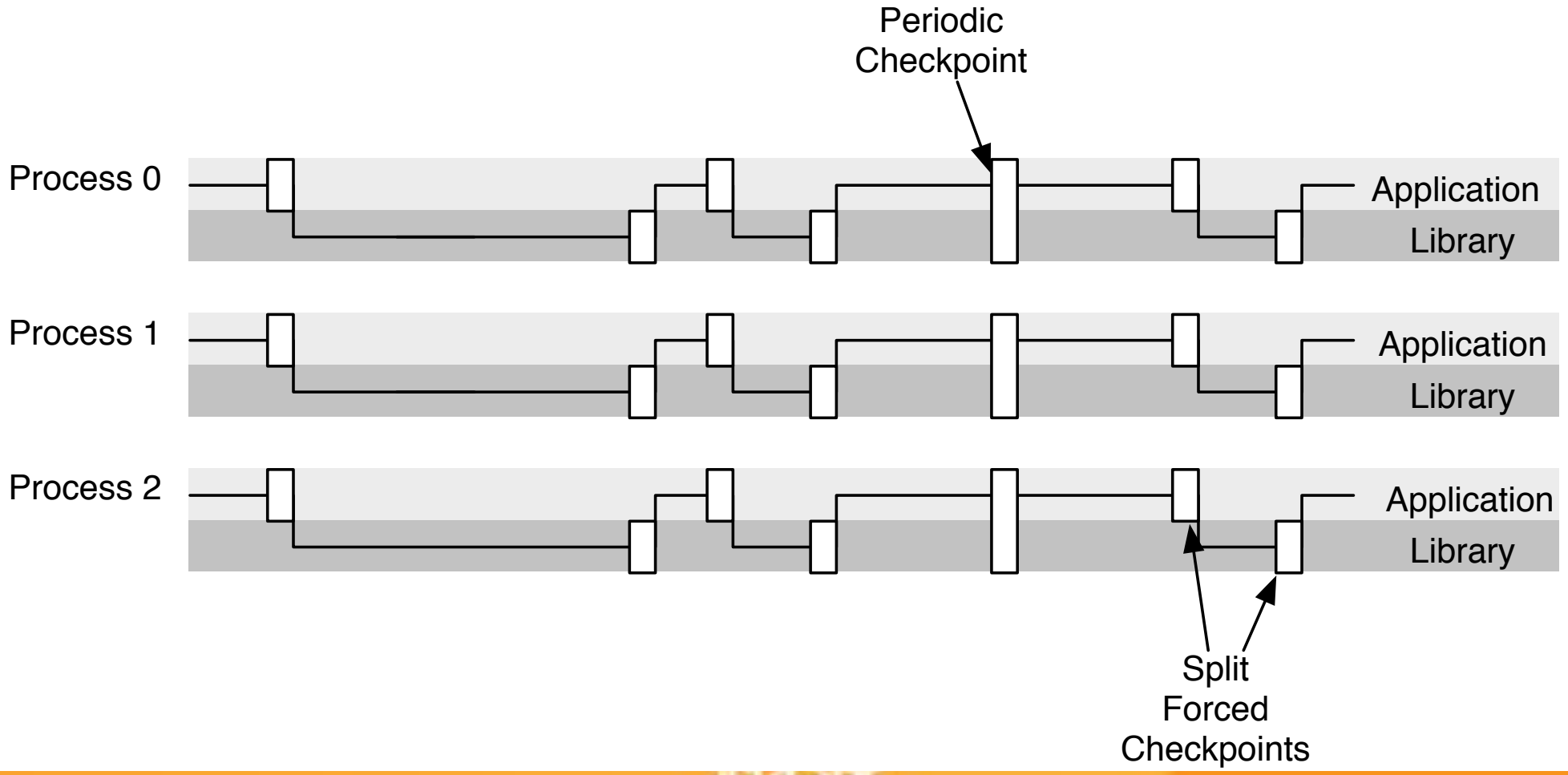
$$O(MN)$$

flops for the checksum creation.
Less than 5% computational overhead



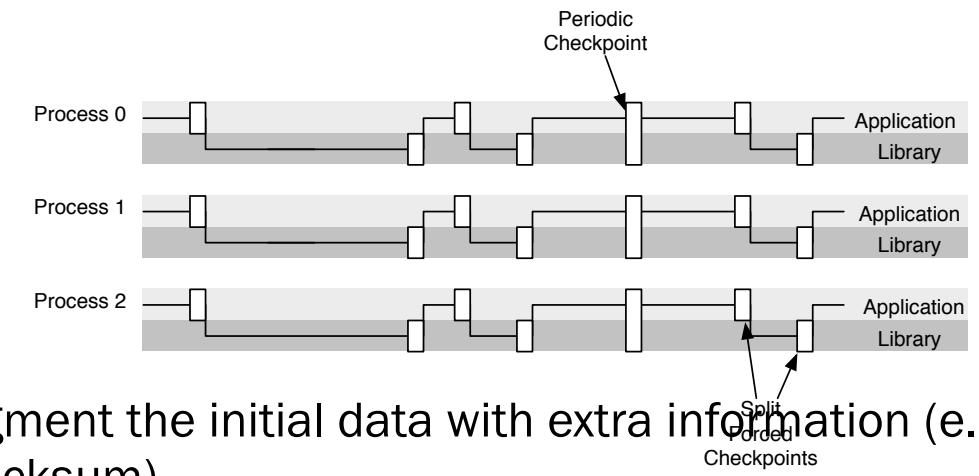
Mixed resilient solutions (model)

- An iterative application using a resilient library
  - Protect the application with traditional checkpoint/restart
  - Protect the library with new techniques (ABFT)

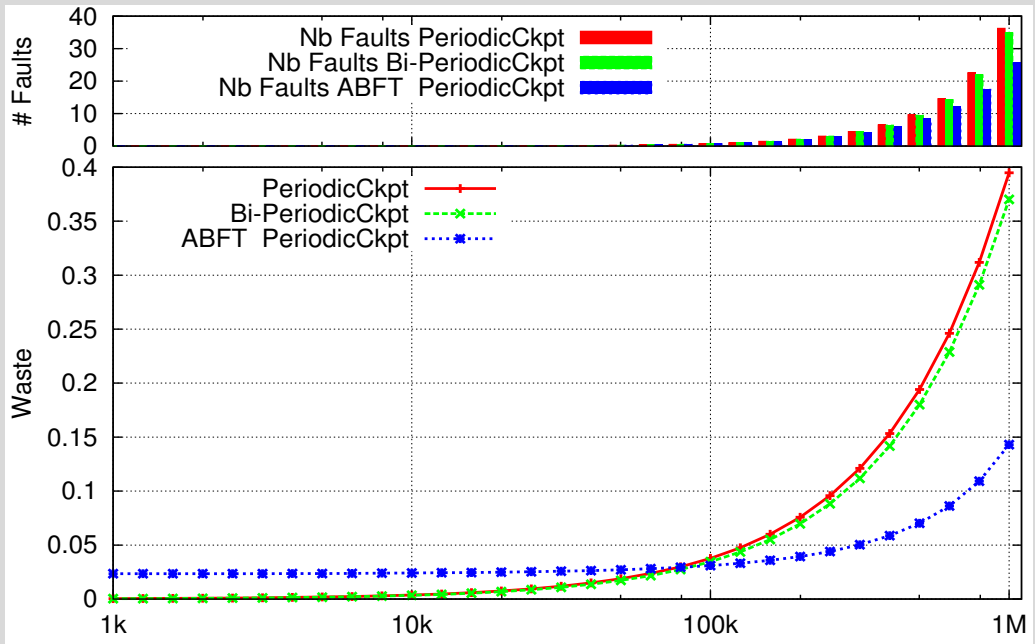


Mixed resilient solutions (model)

- An iterative application using a resilient library
  - Protect the application with traditional checkpoint/restart
  - Protect the library with new techniques (ABFT)



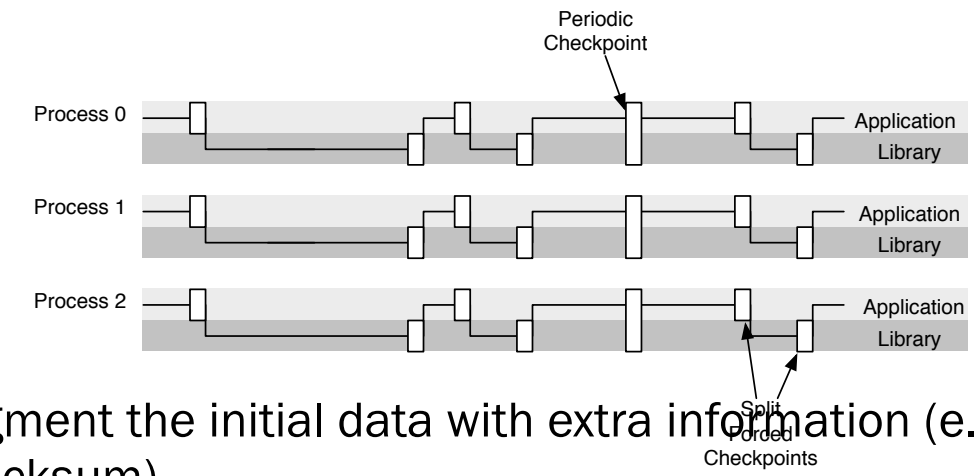
- Augment the initial data with extra information (e.g. checksum)
  - Maintain this extra information through the algorithm
  - Allow soft and hard error survival
- Library using ABFT: dense and sparse LA, matrix-matrix multiplications, one-sided and two-sided factorizations, CG, GMRES



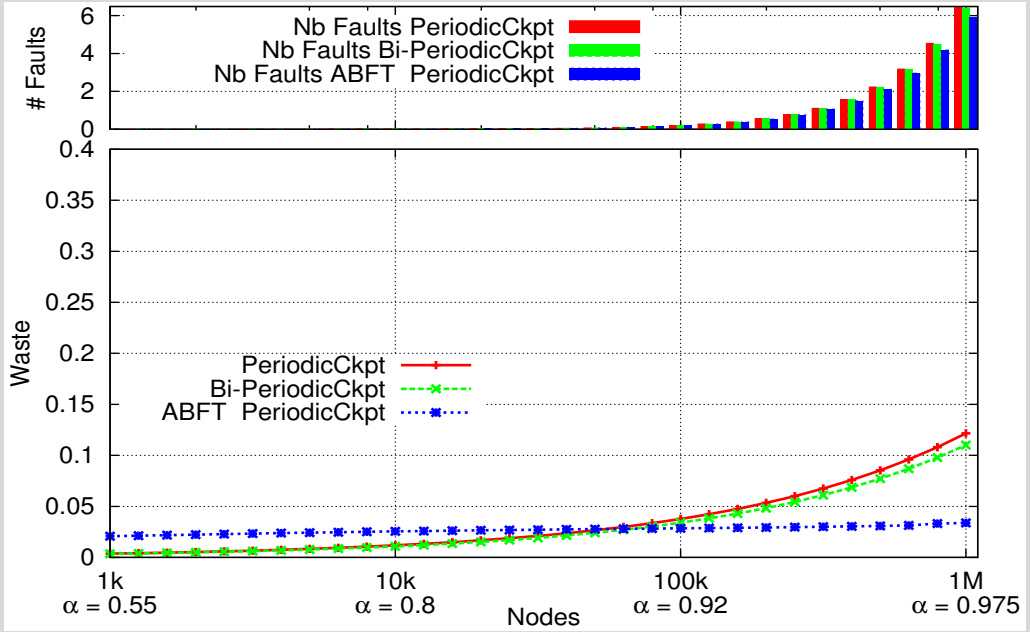
≠ Exascale machine: same comp increase
Memory per component remains constant
Problem size increases (O(√n): matrix based)
μ at $n=10^5$ 1 day is O(1/n)
C (=R) at $n=10^5$ is 1m, is in O(n)
80% in library, 20% in application

Mixed resilient solutions (model)

- An iterative application using a resilient library
  - Protect the application with traditional checkpoint/restart
  - Protect the library with new techniques (ABFT)



- Augment the initial data with extra information (e.g. checksum)
  - Maintain this extra information through the algorithm
  - Allow soft and hard error survival
- Library using ABFT: dense and sparse LA, matrix-matrix multiplications, one-sided and two-sided factorizations, CG, GMRES



≠ Exascale machine: same comp increase
Memory per component remains constant
Problem size increases (O(√n): matrix based)
μ at $n=10^5$ 1 day is O(1/n)
C (=R) at $n=10^5$ is 1m, independent of n (O(1))
80% in library, 20% in application

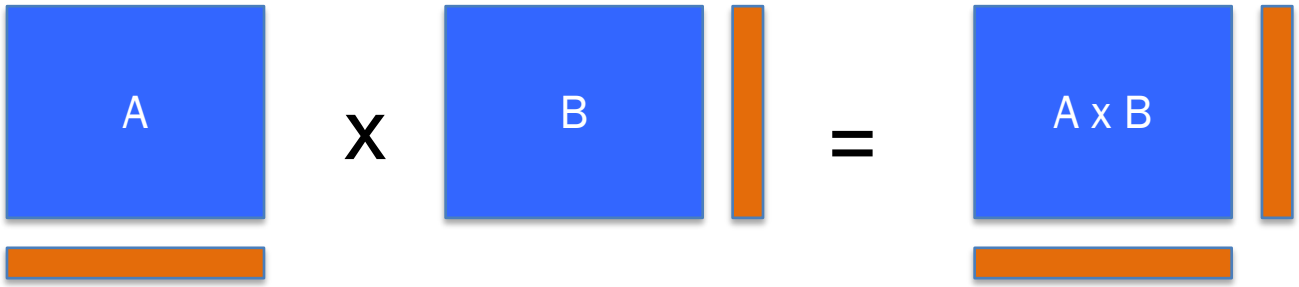
VeloC: Very Low Overhead Transparent Multilevel Checkpoint/Restart

Franck Cappello

Bogdan Nicolae

Algorithm Based Fault Tolerance (ABFT)

- *Takes advantage of existing mathematical relationship(s)*
  - *Introduced (cheaply, if possible) by ABFT*
- *KH Huang & Jacob Abraham, ABFT for Matrix Operations, IEEE Trans. Computers. 01/1984 for systolic array*
- *Matrix extended to contain additional information.*
  - *Extra column or row contains checksum.*
- *Matrix algorithm designed to operate on the data and the encoded checksum.*
  - *Checksum invariant during the course of the algorithm.*
  - *No checkpoint needed.*



$$\begin{bmatrix} A \\ G^T A \end{bmatrix} \times \begin{bmatrix} B & BG \end{bmatrix} = \begin{bmatrix} AB & ABG \\ G^T AB & G^T ABG \end{bmatrix}$$

G^T $G^T A$ *and* BG *are the check sums.*

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

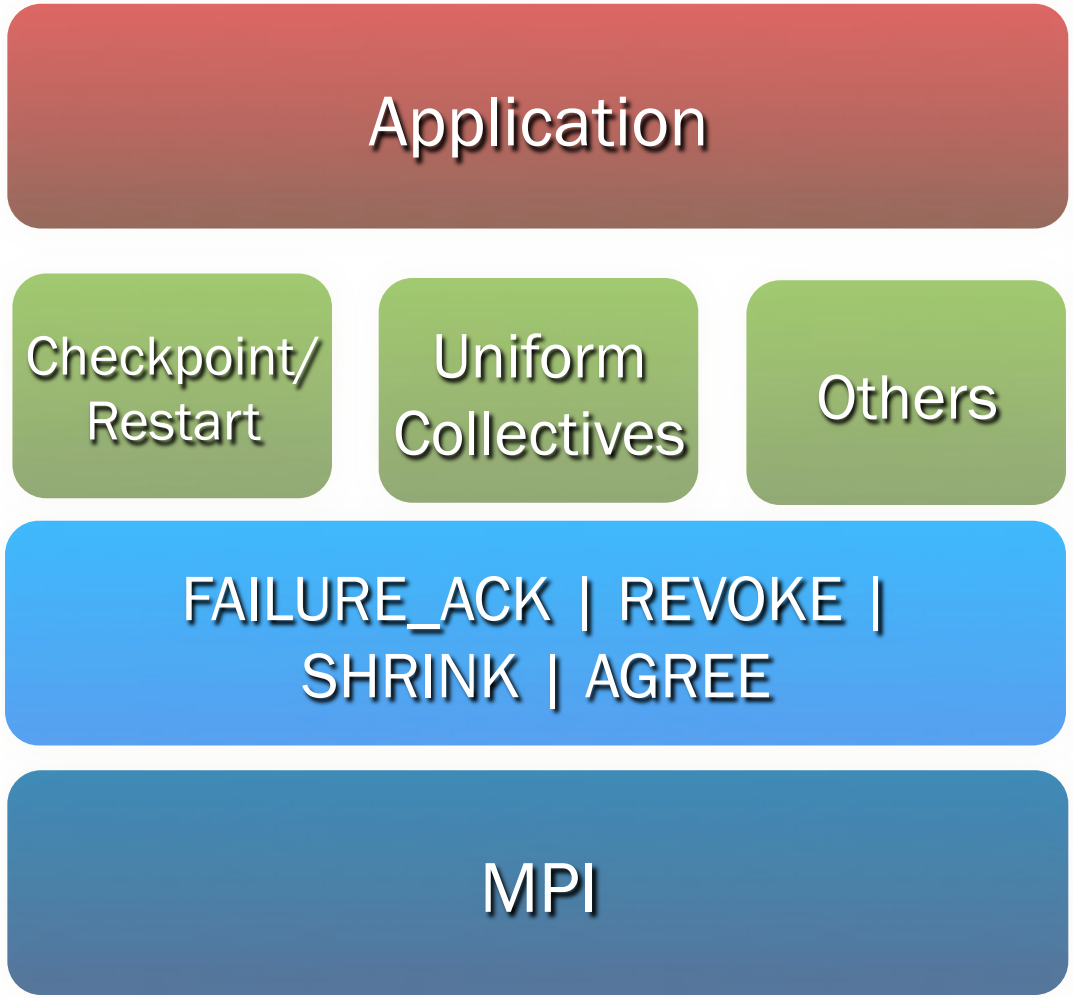
- Can be attached to all objects allowing data transfers: communicators, windows and files
- Allow for minimalistic error recovery: the standard suggests only non-MPI related actions, and no collective operations
- All newly created MPI objects inherit the error handler from their parent
  - A global 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 *, ...);
MPI_Comm_create_errhandler(errh, errhandler_fct);
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

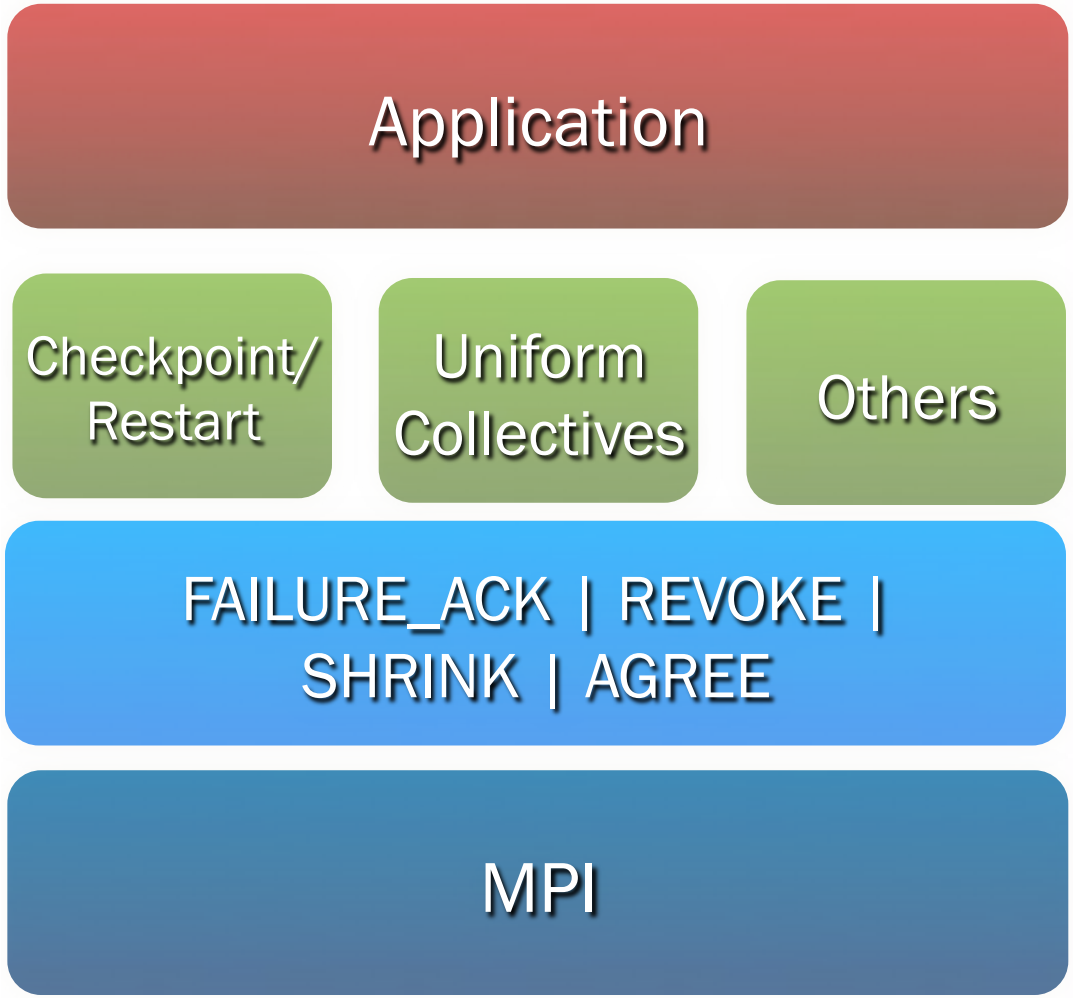


Minimal Feature Set for a Resilient MPI

- Failure Notification
- Error Propagation
- Error Recovery

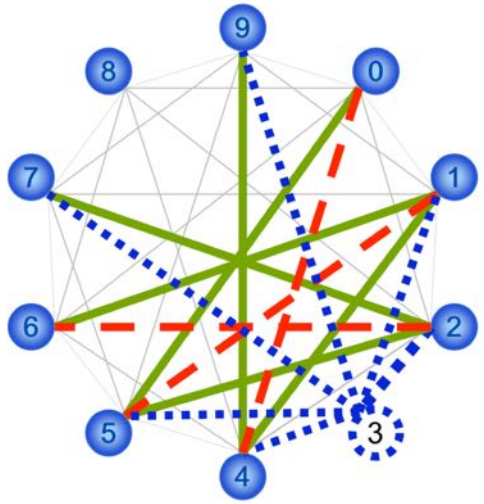
Not all recovery strategies require all of these features, that's why the interface splits notification, propagation and recovery.

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



Failure Notification

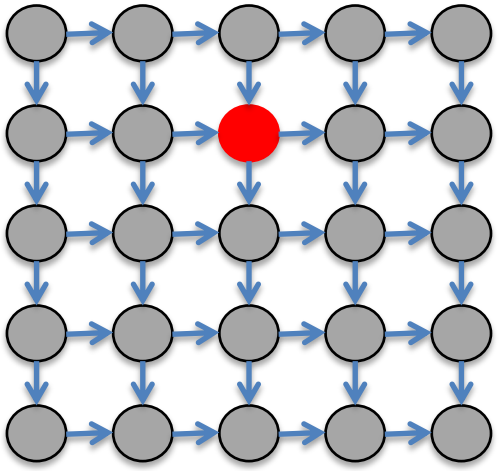
- MPI stands for scalable parallel applications it would be unreasonable to expect full connectivity between all peers
- The failure detection and notification should have a neighboring scope: only processes involved in a communication with the failed process might detect the failure



- 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: offers 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?

Part rationale, part examples

ULFM MPI API, CONTINUING THROUGH ERRORS

Bye bye, world

See 00.noft.c

```
19 int main(int argc, char *argv[])
20 {
21     int rank, size;
22
23     MPI_Init(NULL, NULL);
24     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25     MPI_Comm_size(MPI_COMM_WORLD, &size);
26
27     if( rank == (size-1) ) raise(SIGKILL);
28     MPI_Barrier(MPI_COMM_WORLD);
29     printf("Rank %d / %d\n", rank, size);
30
31     MPI_Finalize();
32 }
```

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?

See q01.err_returns.c

Bye bye, world, but orderly

See 01.err_returns.c

```
19 int main(int argc, char *argv[])
20 {
21     int rank, size, rc, len;
22     char errstr[MPI_MAX_ERROR_STRING];
23
24     MPI_Init(NULL, NULL);
25     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26     MPI_Comm_size(MPI_COMM_WORLD, &size);
27
28     MPI_Comm_set_errhandler(MPI_COMM_WORLD,
29                             MPI_ERRORS_RETURN);
30
31     if( rank == (size-1) ) raise(SIGKILL);
32     rc = MPI_Barrier(MPI_COMM_WORLD);
33     MPI_Error_string(rc, errstr, &len);
34     printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n",
35             rank, size, errstr);
36
37     MPI_Finalize();
38 }
```

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

```
 19 static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
…
 21     char errstr[MPI_MAX_ERROR_STRING];
…
 26     MPI_Error_string( *err, errstr, &len );
 27     printf("Rank %d / %d: Notified of error %s\n",
 28            rank, size, errstr);
 29 }
 30
 31 int main(int argc, char *argv[]) {
…
 33     MPI_Errhandler errh;
…
 39     MPI_Comm_create_errhandler(verbose_errhandler,
 40                                &errh);
 41     MPI_Comm_set_errhandler(MPI_COMM_WORLD,
 42                             errh);
…
 45     MPI_Barrier(MPI_COMM_WORLD);
 46     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

No need to collect rc anymore ☺

- Still using only MPI-2

What caused the error?

See 02.err_hander.c

```
13 #include <mpi.h>
14 #include <mpi-ext.h>
…
19 static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
20     MPI_Comm comm = *pcomm;
21     int err = *perr;
…
23     int …, eclass;
…
27     MPI_Error_class(err, &eclass);
28     if( MPIX_ERR_PROC_FAILED != eclass ) {
29         MPI_Abort(comm, err);
30     }
…
```

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 new error classes:
  - 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

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

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
- Beware: All calls to MPI_Comm_failure_get_acked between a set of MPI_Comm_failure_ack return the same set of failed processes

Who caused the error

Still in 02.err_handler.c

```
…
 19 static void verbose_errhandler(MPI_Comm* pcomm, int* perr,
...) {
 20     MPI_Comm comm = *pcomm;
…
 35     MPIX_Comm_failure_ack(comm);
 36     MPIX_Comm_failure_get_acked(comm, &group_f);
 37     MPI_Group_size(group_f, &nf);
 38     MPI_Error_string(err, errstr, &len);
 39     printf("Rank %d / %d: Notified of error %s. %d found
dead: { ",
 40            rank, size, errstr, nf);
 41

 52 }
```

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

```
…
 19 static void verbose_errhandler(MPI_Comm* pcomm, int* perr,
...) {
 20     MPI_Comm comm = *pcomm;
…
 35     MPIX_Comm_failure_ack(comm);
 36     MPIX_Comm_failure_get_acked(comm, &group_f);
 37     MPI_Group_size(group_f, &nf);
 38     MPI_Error_string(err, errstr, &len);
 39     printf("Rank %d / %d: Notified of error %s. %d found
dead: { ",
 40            rank, size, errstr, nf);
 41
 42     ranks_gf = (int*)malloc(nf * sizeof(int));
 43     ranks_gc = (int*)malloc(nf * sizeof(int));
 44     MPI_Comm_group(comm, &group_c);
 45     for(i = 0; i < nf; i++)
 46         ranks_gf[i] = i;
 47     MPI_Group_translate_ranks(group_f, nf, ranks_gf,
 48                               group_c, ranks_gc);
 49     for(i = 0; i < nf; i++)
 50         printf("%d ", ranks_gc[i]);
 51     printf("}\n");
 52 }
```

Move the “mark” in the known failures list

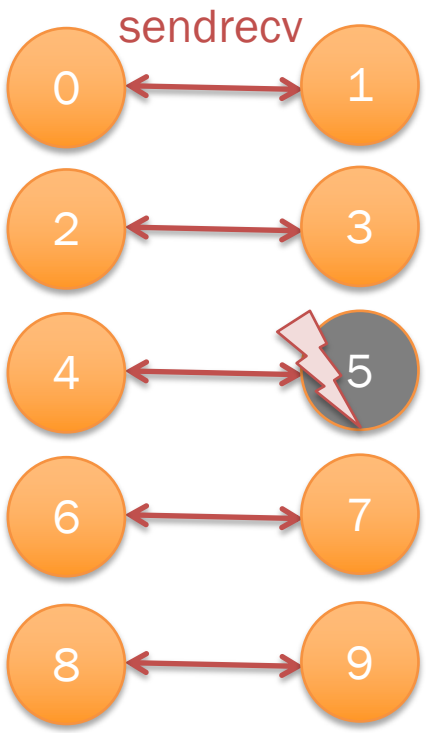
Get the group of marked failed processes

Translate the failed group member’s ranks, in comm

Insulation from irrelevant failures

See 03.undisturbed.c

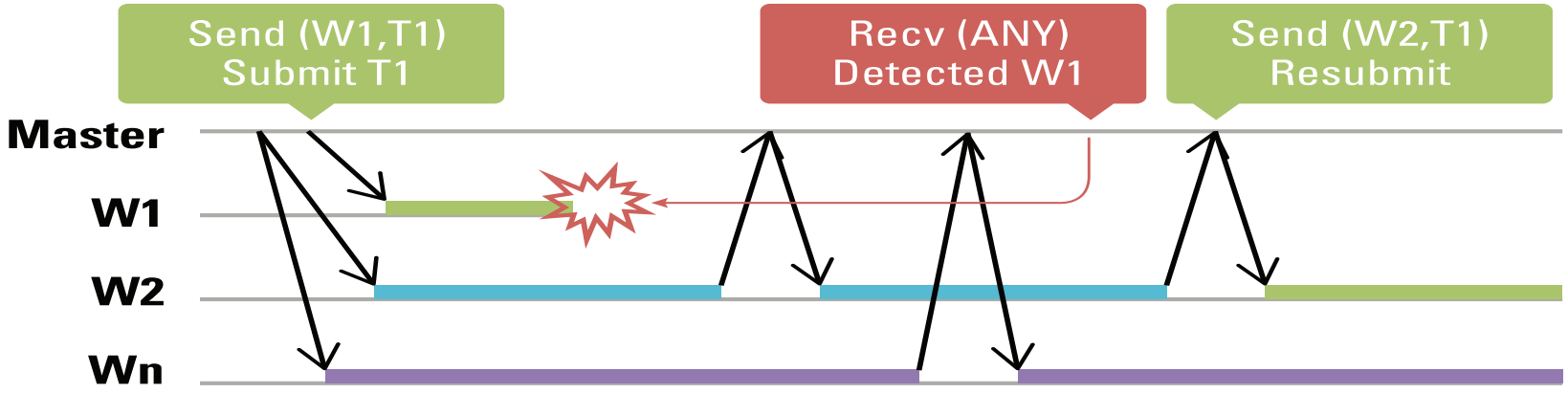
```
25      double myvalue, hisvalue=NAN;
…
36      myvalue = rank/(double)size;
37      if( 0 == rank%2 )
38          peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;
39      else
40          peer = rank-1;
41
42      if( rank == (size/2) ) raise(SIGKILL);
43      /* exchange a value between a pair of two consecutive
44       * odd and even ranks; not communicating with anybody
45       * else. */
46      MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
47                   &hisvalue, 1, MPI_DOUBLE, peer, 1,
48                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
49
50      if( peer != MPI_PROC_NULL)
51          printf("Rank %d / %d: value from %d is %g\n",
52                 rank, size, peer, hisvalue);
```

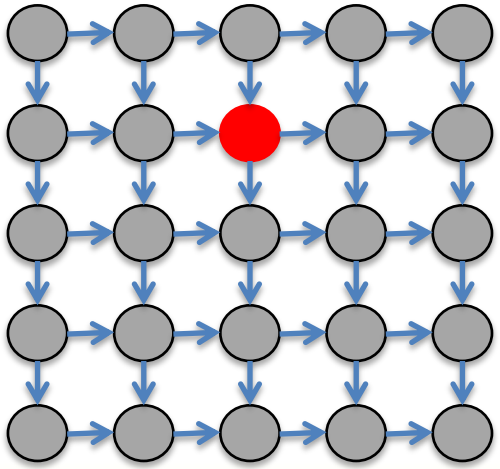


What happens?

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 a Master-Worker example, we can continue w/o recovery!
  - Master sees failed worker
  - Resubmit the lost work unit onto another worker
  - Quietly continues
- Same story with Stencil pattern!
  - Exchange with next neighbor in the same direction instead

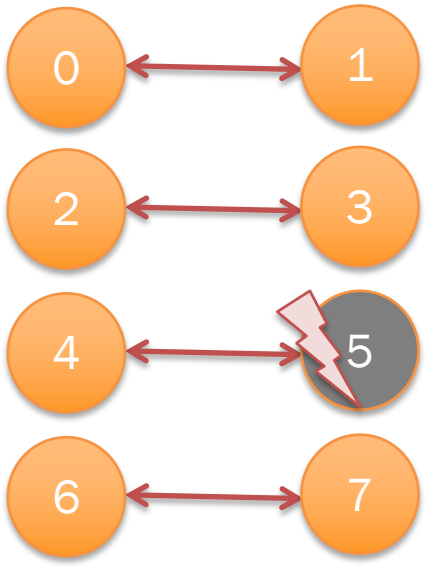




Insulation from irrelevant failures

See 03.undisturbed.c

```
 25     double myvalue, hisvalue=NAN;
…
 36     myvalue = rank/(double)size;
 37     if( 0 == rank%2 )
 38         peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;
 39     else
 40         peer = rank-1;
 41
 42     if( rank == (size/2) ) raise(SIGKILL);
 43     /* exchange a value between a pair of two consecutive
 44      * odd and even ranks; not communicating with anybody
 45      * else. */
 46     MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
 47                  &hisvalue, 1, MPI_DOUBLE, peer, 1,
 48                  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!

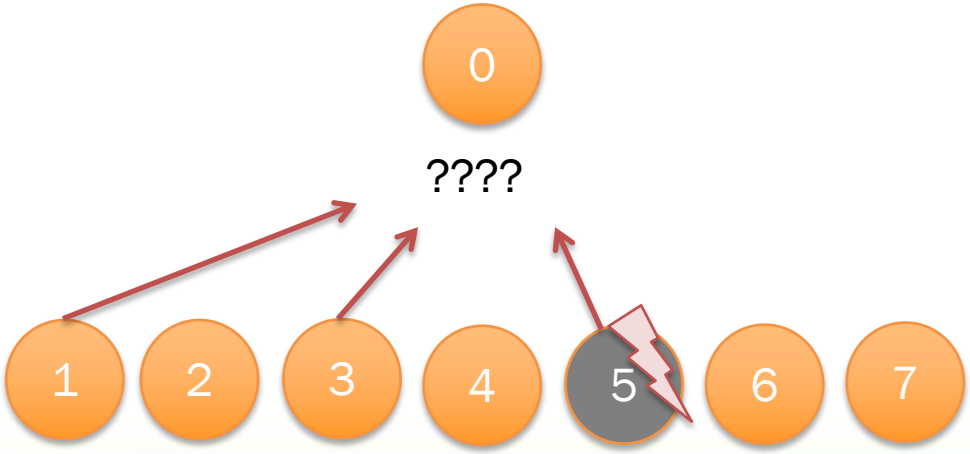
Dealing with MPI_ANY_SOURCE

See 08.err_any_source.c

```
36     if( 0 != rank ) {
37         MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
38     }
39     else {
40         printf("Recv(ANY) test\n");
41         for(i = 1; i < size-nf; ) {
42             rc = MPI_Recv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
43             if( MPI_SUCCESS == rc ) {
44                 printf("Received from %d during recv %d\n", unused, i);
45                 i++;
46             }
47             else {
```

Assume a process dies before sending the message

No specified source



Dealing with MPI_ANY_SOURCE

See 08.err_any_source.c

```
36     if( 0 != rank ) {
37         MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
38     }
39     else {
40         printf("Recv(ANY) test\n");
41         for(i = 1; i < size-nf; ) {
42             rc = MPI_Recv(&unused, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
43             if( MPI_SUCCESS == rc ) {
44                 printf("Received from %d during recv %d\n", unused, i);
45                 i++;
46             }
47             else {
```

Assume a process dies before sending the message

No specified source, the failure detection is homogeneous

MPIX_ERR_PROC_FAILED_PENDING 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)

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

Lets keep it neat and tidy

STABILIZING AFTER AN ERROR

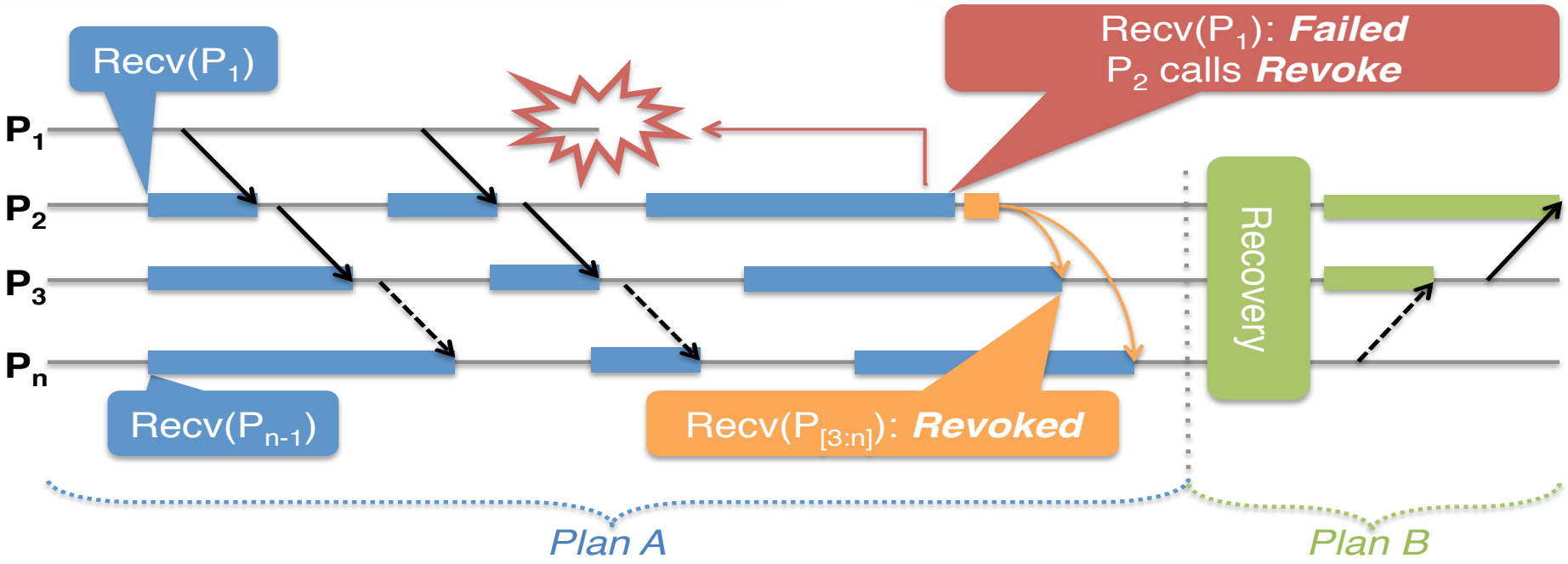
Regrouping after error

See q04.if_error.c

```
56      /* Assign left and right neighbors to be rank-1 and rank+1
57       * in a ring modulo np */
58      left   = (np+rank-1)%np;
59      right  = (np+rank+1)%np;
60
61      for( i = 0; i < 10; i++ ) {
…
70          /* At every iteration, a process receives from it's 'left' neighbor
71           * and sends to 'right' neighbor (ring fashion, modulo np)
72           * ... -> 0 -> 1 -> 2 -> ... -> np-1 -> 0 ... */
73          rc = MPI_Sendrecv( sarray, COUNT, MPI_DOUBLE, right, 0,
74                             rarray, COUNT, MPI_DOUBLE, left , 0,
75                             fcomm, MPI_STATUS_IGNORE );
…
80          if( rc != MPI_SUCCESS ) {
81              /* ???>>> Hu ho, this program has a problem here */
82              goto cleanup;
83          }
```

- Run q04.if_error with 5 processes. What happens?
- How can it be fixed ?

Regrouping after error

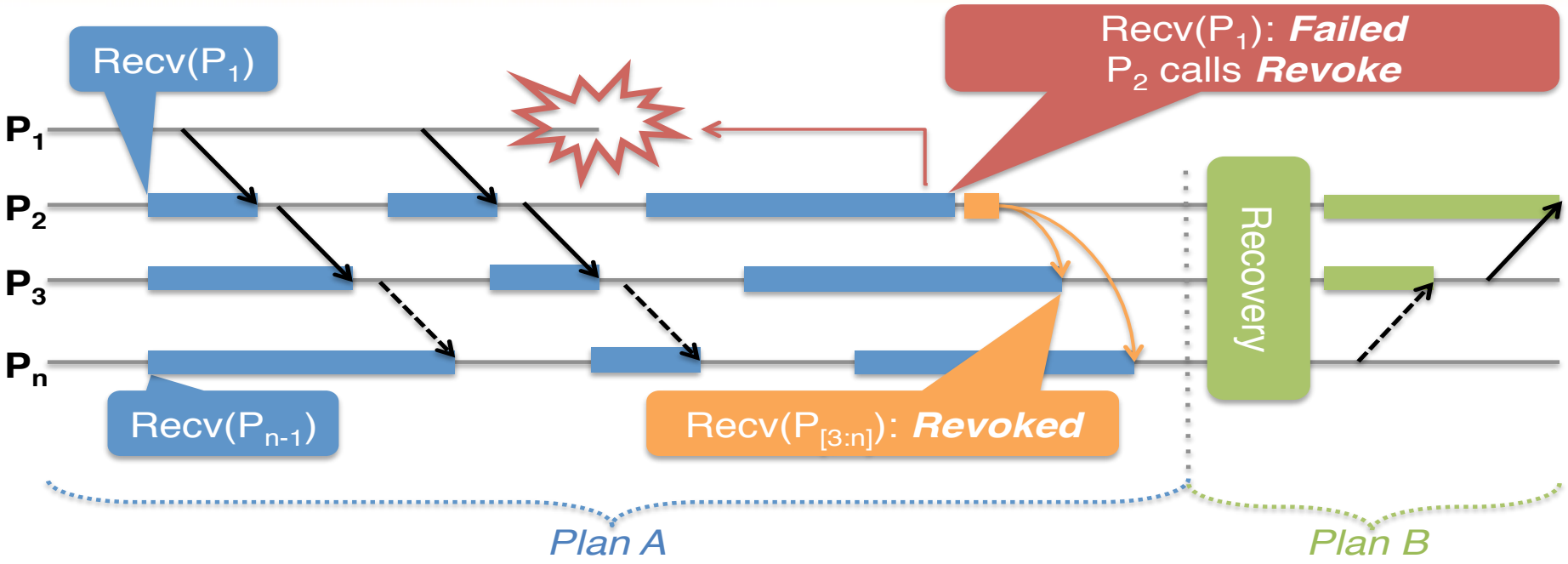


- P1 fails
- P2 raises an error and stop *Plan A* to enter application recovery *Plan B*
- but P3..Pn are stuck in their posted recv
- We need a way to "unstuck" them. Enter Revoke ☺
- P3..Pn join P2 in the recovery

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




```
77          if( rc != MPI_SUCCESS ) {
78              /* Ok, some error occurred, force other processes to exit the loop
79               * because when I am leaving, I will not match the sendrecv, and
80               * that would cause them to deadlock */
81              MPIX_Comm_revoke( fcomm );
82              goto cleanup;
83          }
```

See 04.if_error.c

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

- 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 ?

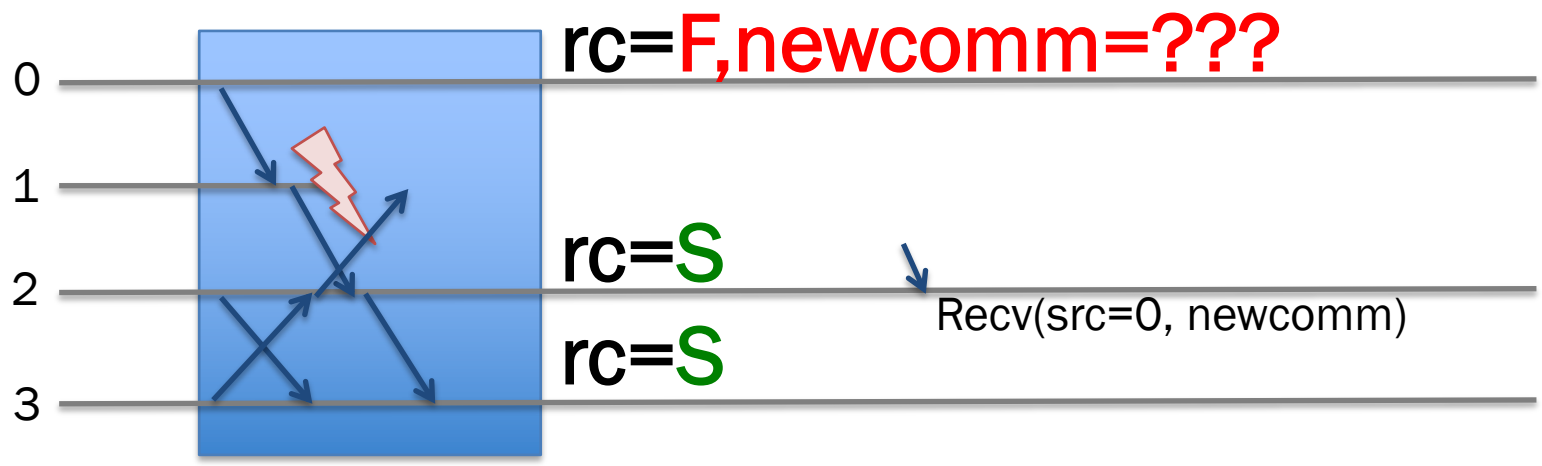
```
bash$ $ULFM_PREFIX/bin/mpirun -np 5 05.err_coll -v
Rank 3 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead:
{ 1 }
Rank 3 / 5: value from 0 is wrong: 0.6
```

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

```
/* 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;
}
```

We need the flag to have a global meaning

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

```
/* 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

Benefits of safety separation

See 07.err_comm_grid2d

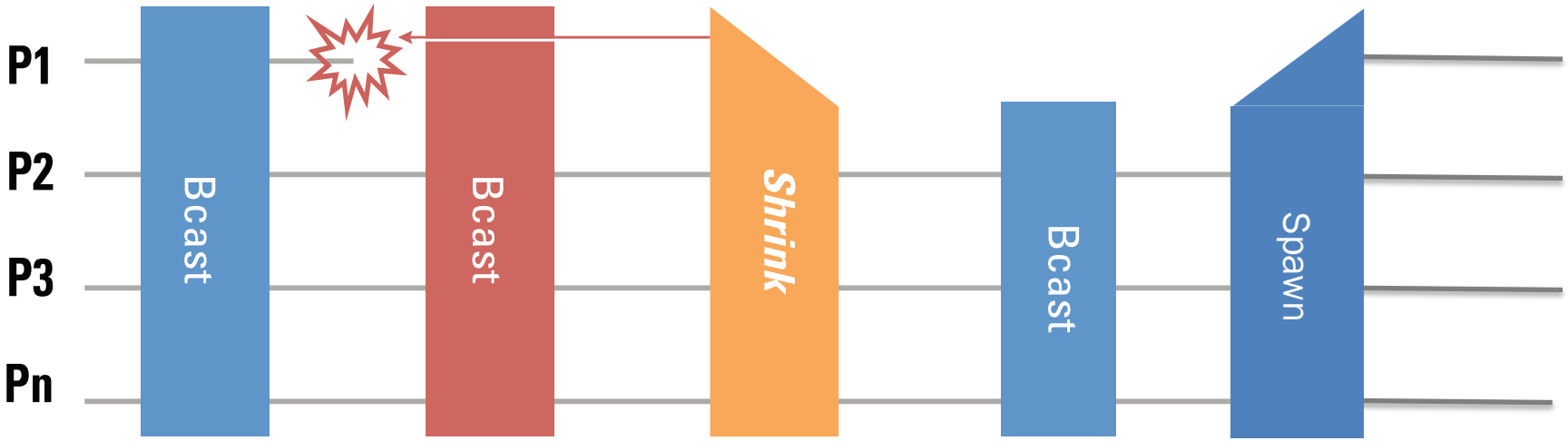
```
/* Create two communicators, representing a PxP 2D grid of
 * the processes. Either return MPIX_ERR_PROC_FAILED at all ranks,
 * then no communicator has been created, or MPI_SUCCESS and all
 * communicators have been created, at all ranks. */
int ft_comm_grid2d(MPI_Comm comm, int p, MPI_Comm *rowcomm, 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

Can we fix it? Yes we can!

FIXING THE WORLD

Full capacity recovery



- After a Revoke, our original comm is unusable. Can we just create a new one ?
- 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

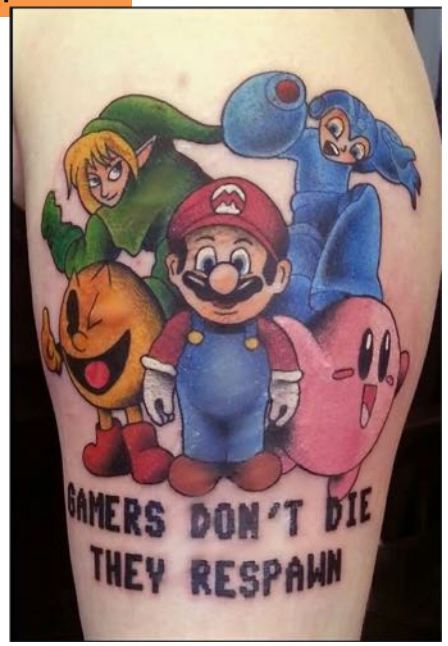
MPI_Comm_shrink

- 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

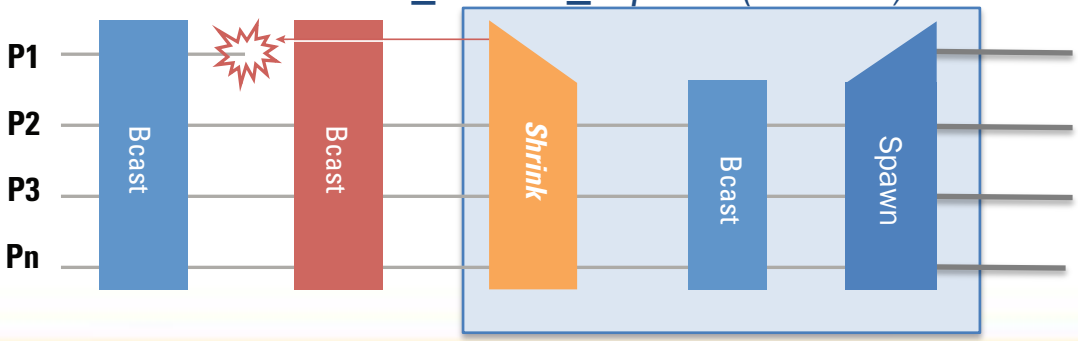
Respawning the deads

See 10.respawn

```
143 int main( int argc, char* argv[] ) {
…
157   /* Am I a spare ? */
158   MPI_Comm_get_parent( &world );
159   if( MPI_COMM_NULL == world ) {
160     /* First run: Let's create an initial world,
161      * a copy of MPI_COMM_WORLD */
162     MPI_Comm_dup( MPI_COMM_WORLD, &world );
…
167   } else {
168     /* I am a spare, lets get the repaired world */
169     MPIX_Comm_replace( MPI_COMM_NULL, &world );
…
174     goto joinwork;
175   }
```



*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”

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

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 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 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

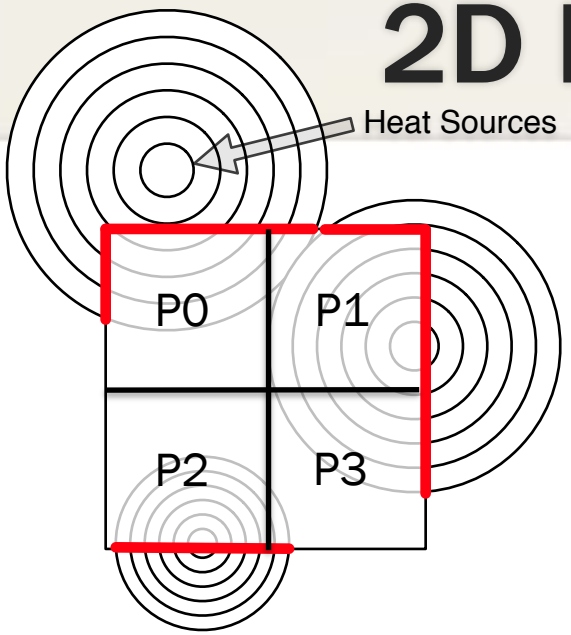
```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

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 1

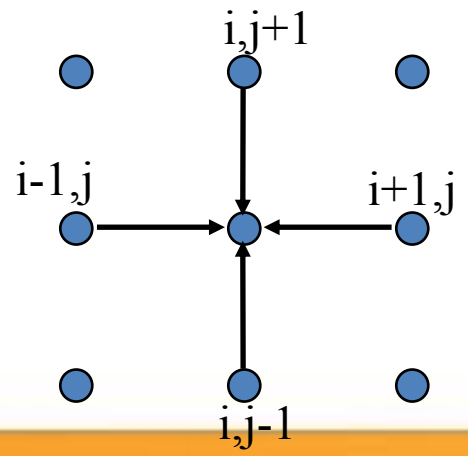
Transaction 2

- 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.)

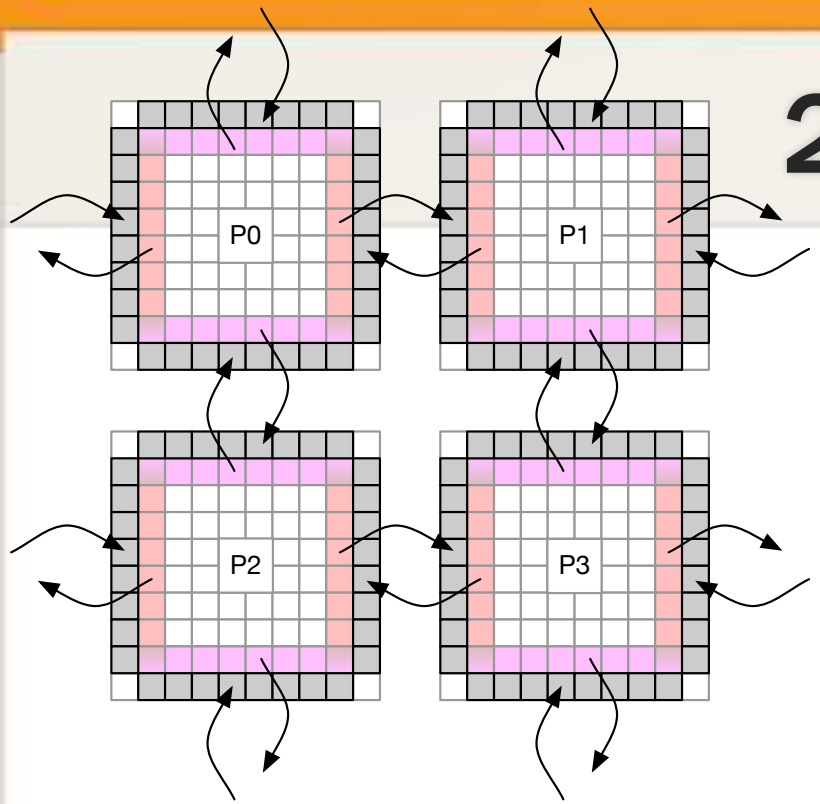


$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$



- 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.)

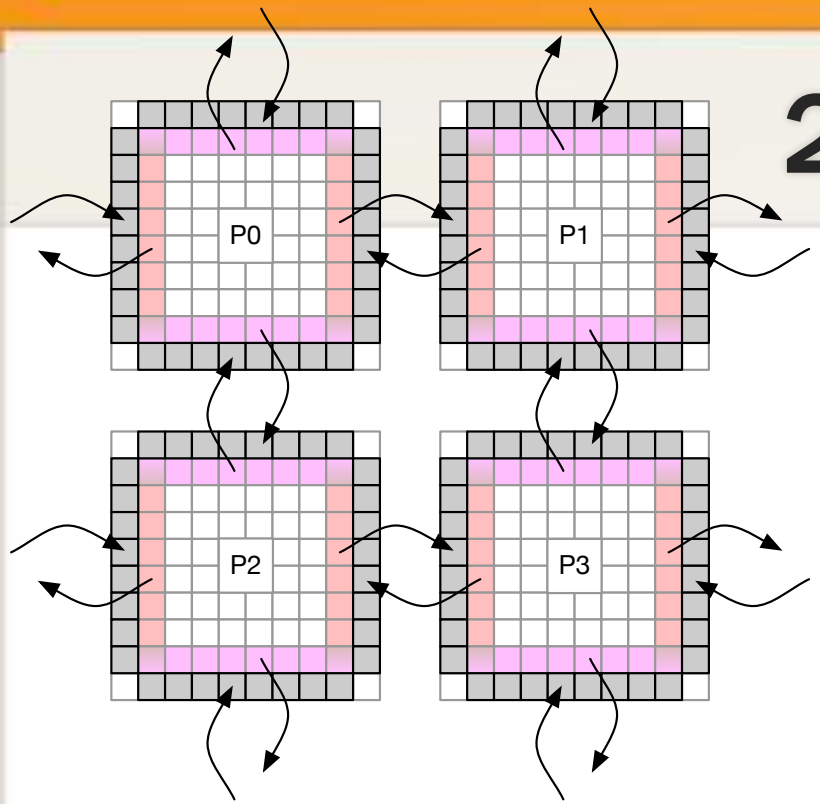


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to checkpoint the local at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

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.)

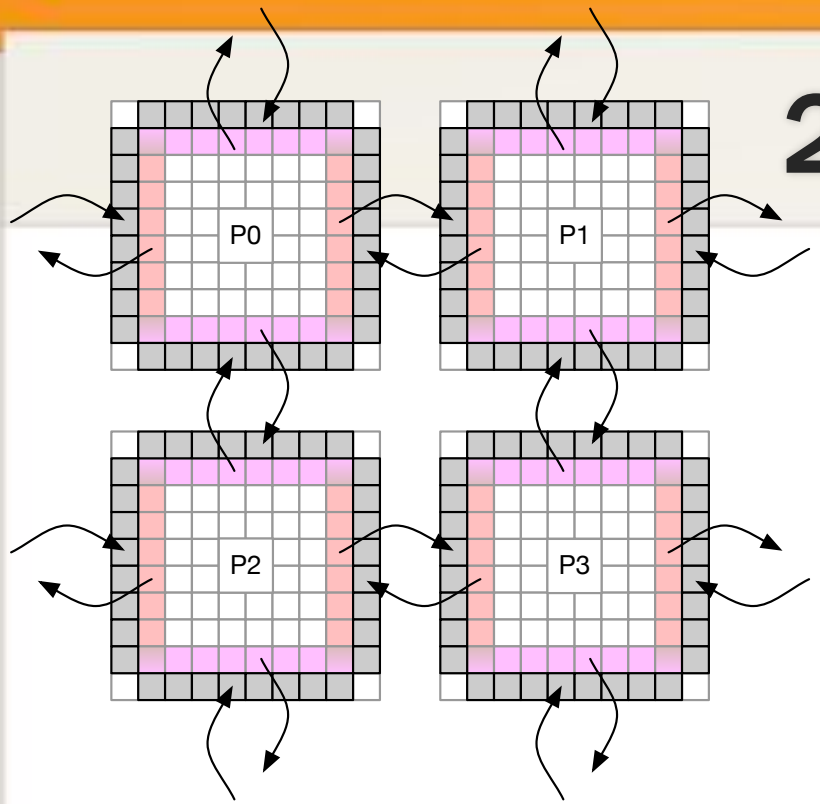


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to checkpoint the local at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/ jacobi_cpu_ckpt_buddy.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.)

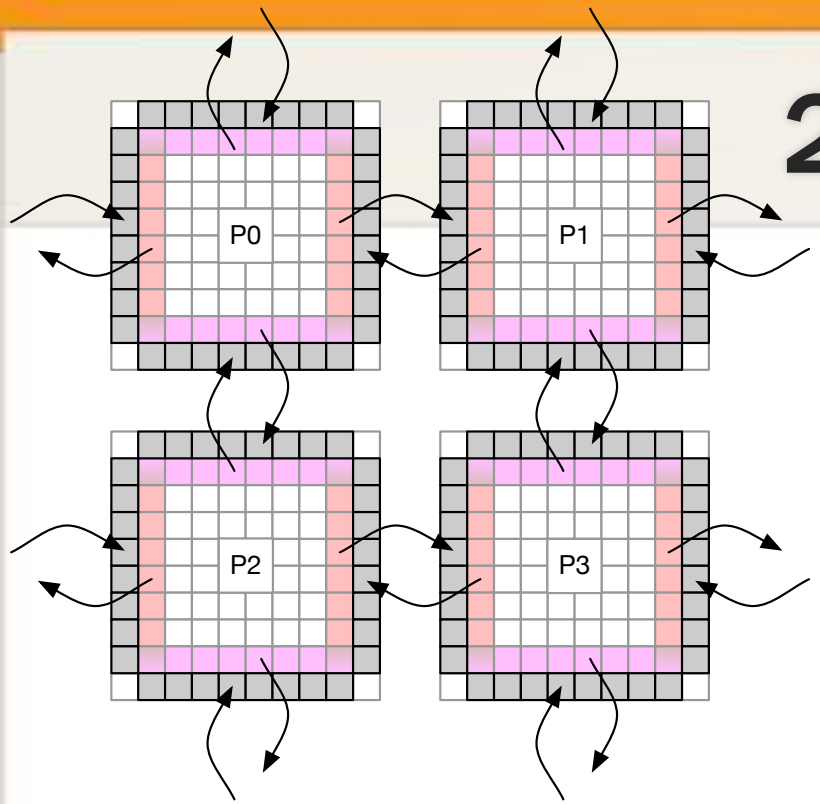


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to checkpoint the local at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/ jacobi_cpu_ckpt_buddy.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.)

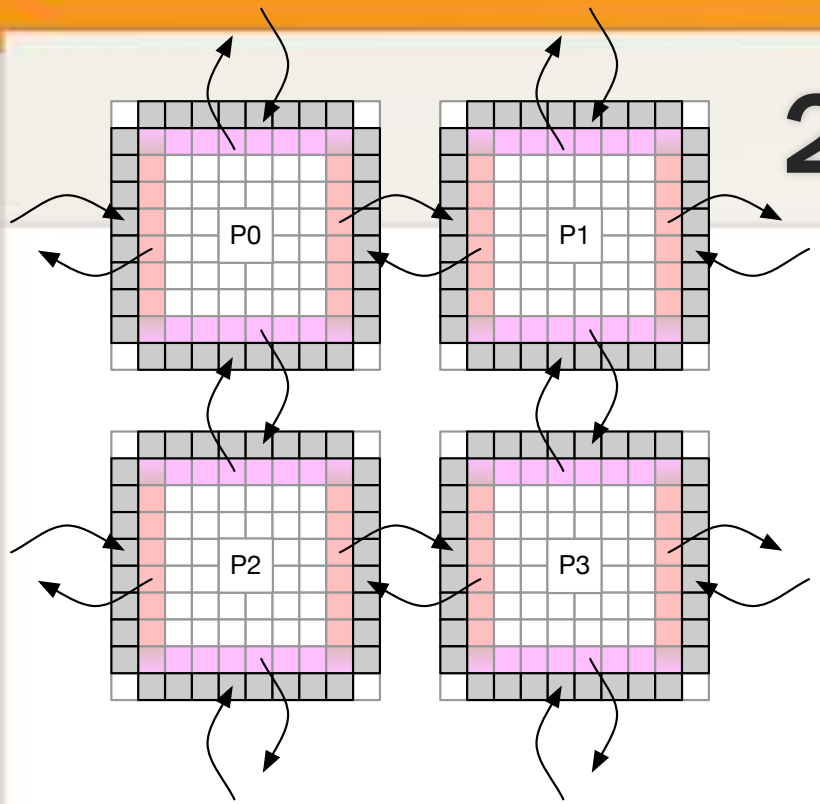


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to checkpoint the local at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/ jacobi_cpu_ckpt_buddy.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.)

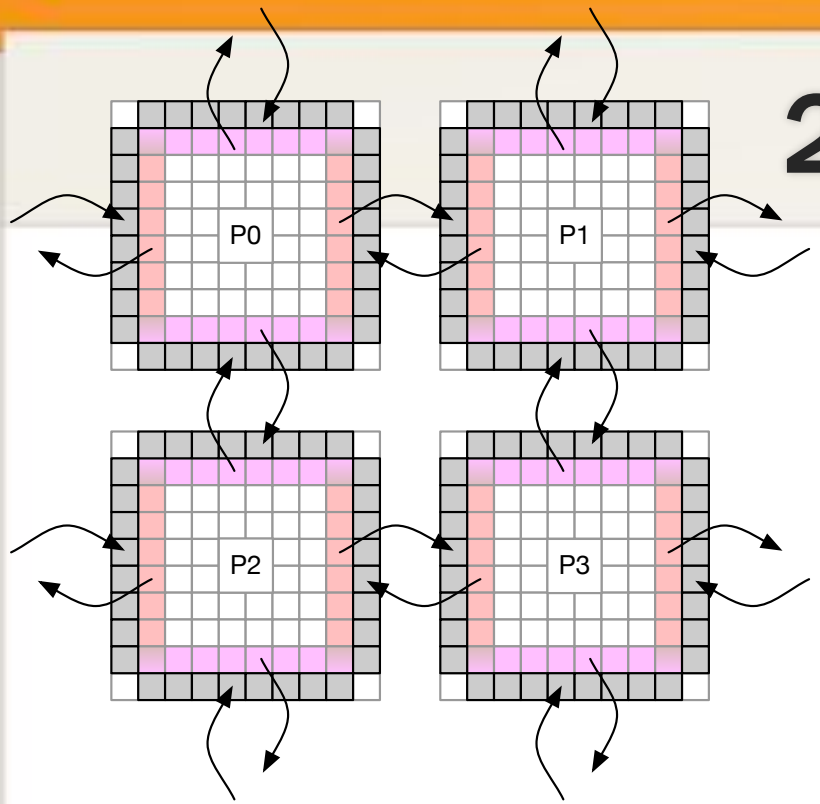


1. We need to be able to break the iterations and jump out of the loop
2. We need to be able to checkpoint the local at regular intervals
3. We need to retrieve the data from the neighbors, coordinate about the iteration and restart the computation

See jacobi/ jacobi_cpu_ckpt_buddy.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 checkpoint the local 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_ckpt_veloc.c

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

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

  do {
    exchange data with neighbors

    if time for chkpt: call VELOC to save local

  local_computation:
    compute local updates and residual

    allreduce the residual with all processes

  } until convergence (iterations or residual)
```

Beyond examples, what people are doing with it

USER'S RECOVERY STORIES

User Level Failure Mitigation: User Adoption

Fenix Framework/S3D

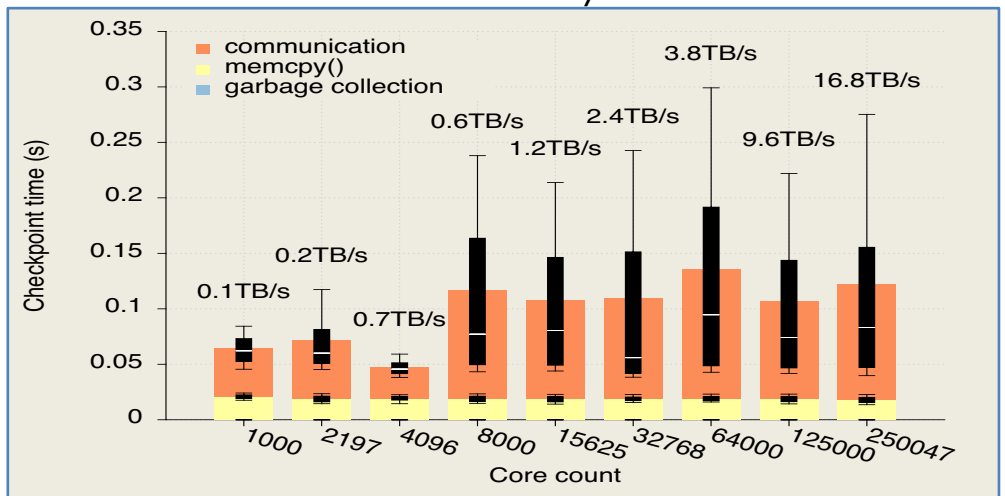


Fig. 3. Checkpoint time for different core counts (8.6 MB/core). The numbers above each test show the aggregated bandwidth (the total checkpoint size over the average checkpoint time).

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

MapReduce

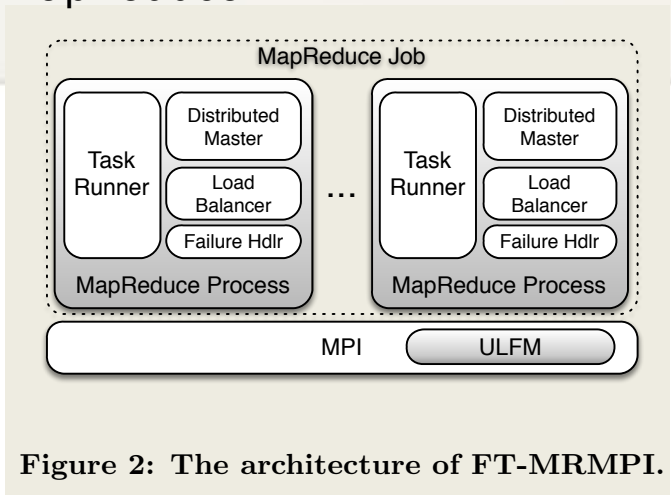
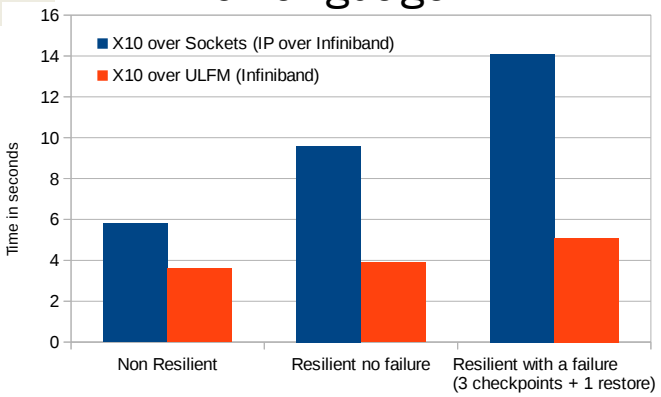


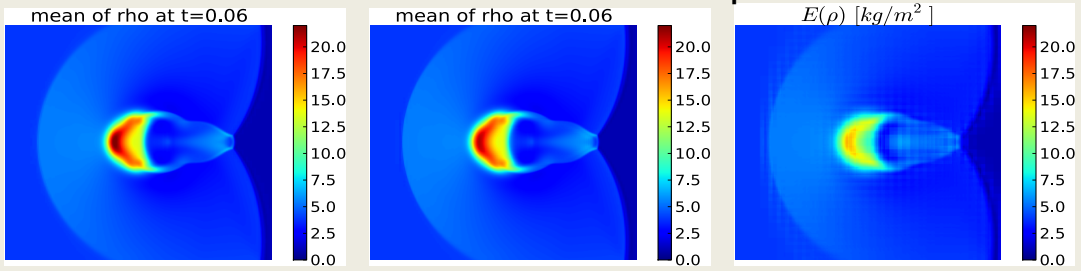
Figure 2: The architecture of FT-MRMPI.

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



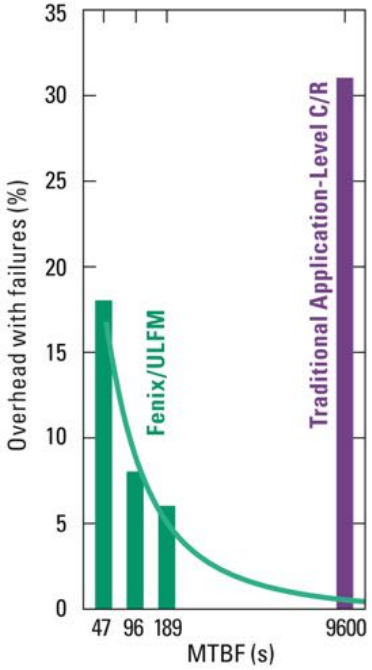
(a) failure-free (b) few failures (c) many failures

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



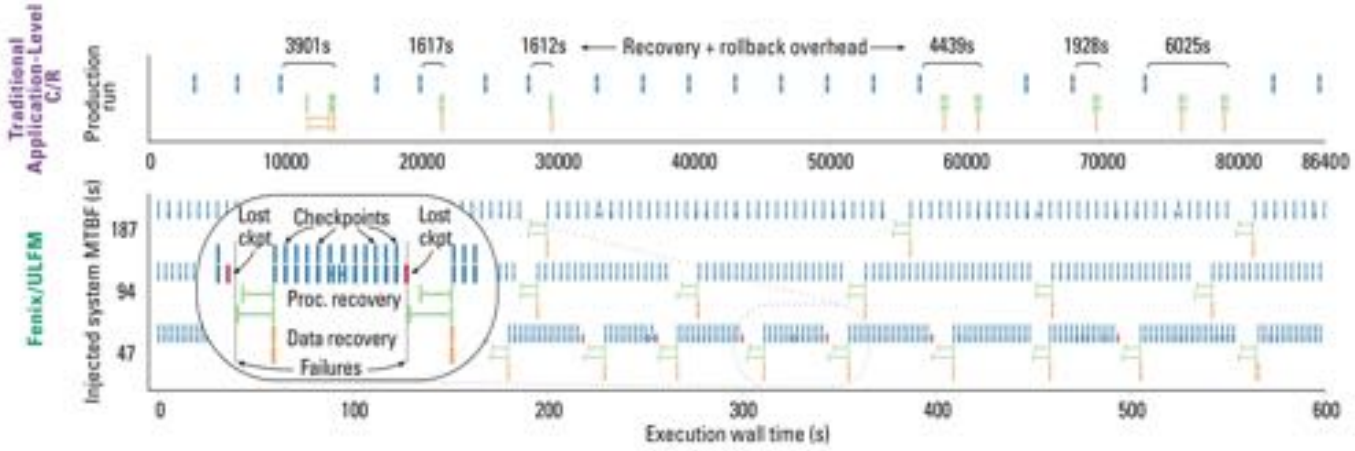


Image courtesy of the authors, M.Gamell, D.Katz, H.Kolla, J.Chen, S.Klasky, and M.Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In Proceedings of SC '14

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

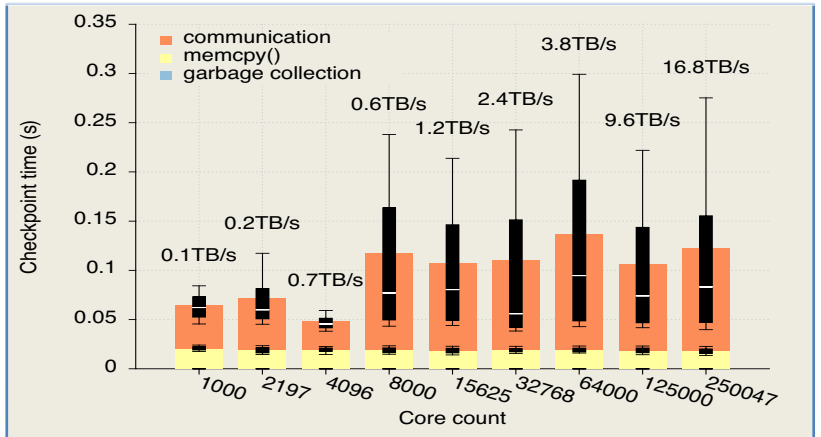


Fig. 3. Checkpoint time for different core counts (8.6 MB/core). The numbers above each test show the aggregated bandwidth (the total checkpoint size over the average checkpoint time).

FRAMEWORKS USING ULFM

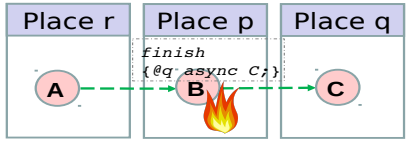
LFLR, FENIX, FTLA, Falanx

Use cases: Languages Resilient X10

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

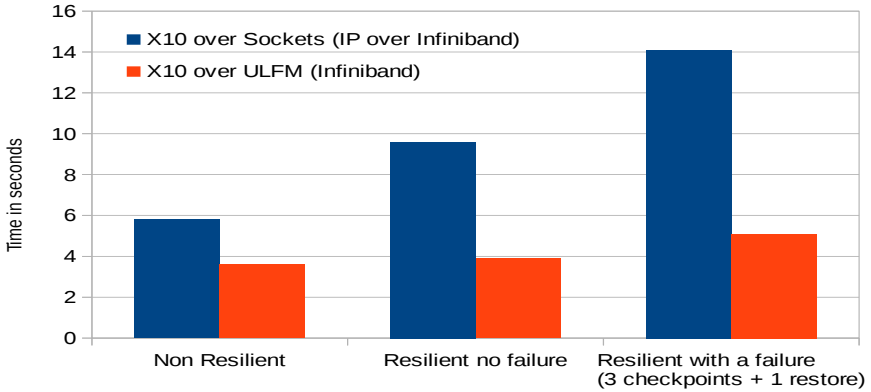
Happens Before Invariance Principle (HBI):
Failure of a place should not alter the happens before relationship between statements at the remaining places.

```
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)

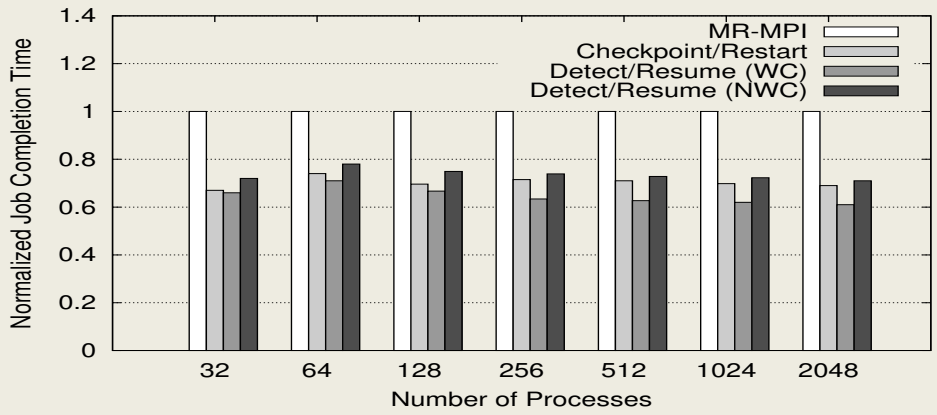


Figure 8: Normalized job completion time of failed and recovery run.

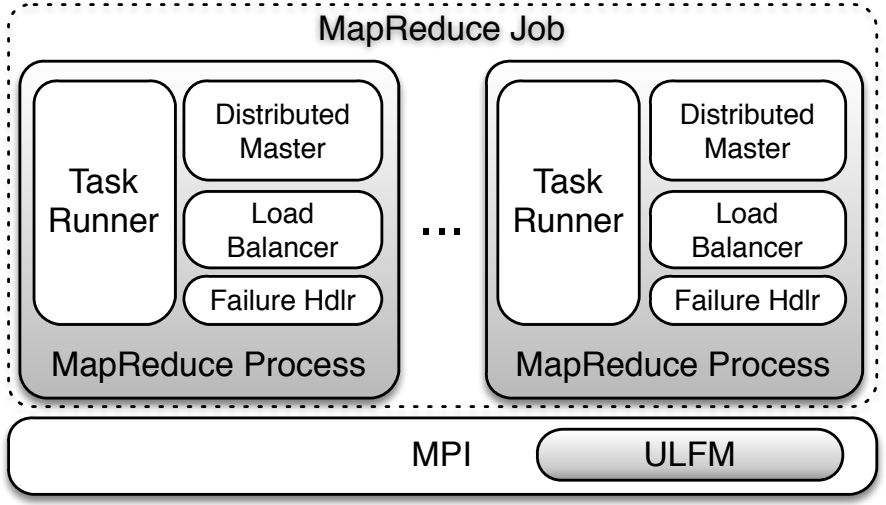


Figure 2: The architecture of FT-MRMPI.

Use cases: Non traditional HPC

SAP Databases

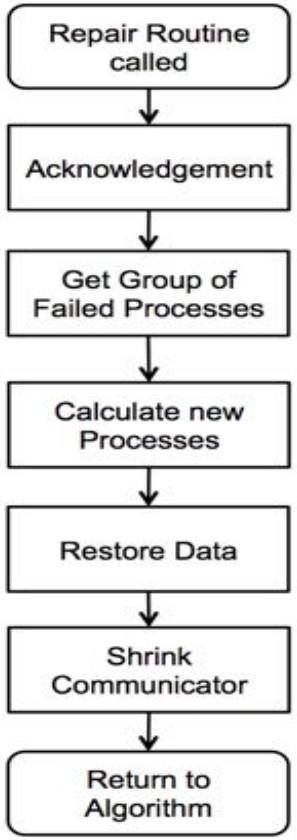


Figure 3.2: Repair Routine

- 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

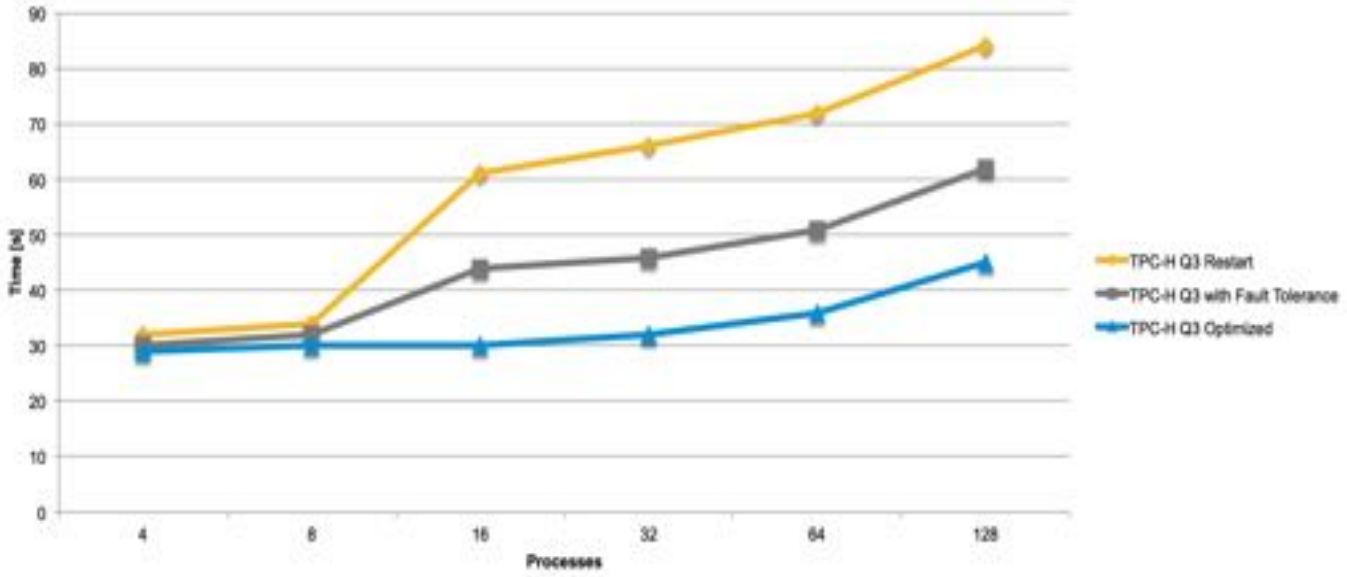
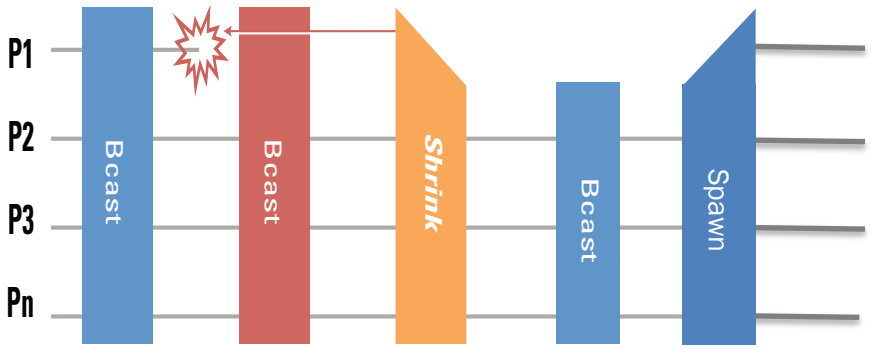


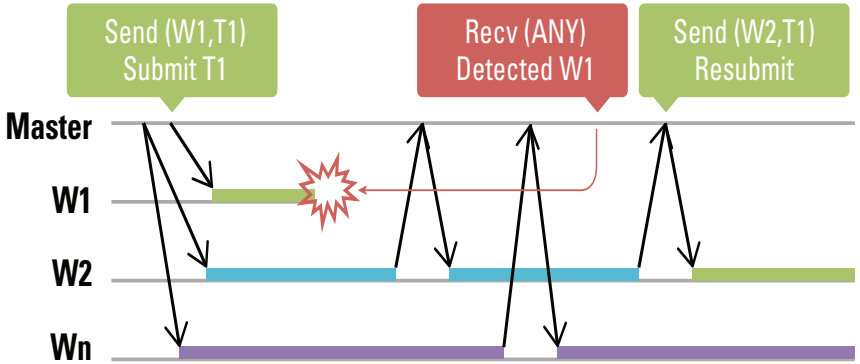
Figure 5.24: Optimization: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 4 (1GB Data per Process)

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.

What is the right approach?

- Bad/good news: there might not be A right approach
- An efficient, scalable and portable approach is certainly a mix of multiple approaches
  - Algorithm specific approaches seems the most efficient, but they have additional requirements from the programming paradigms
  - The development cost should be put in balance with the ownership cost
- 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?
  - 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

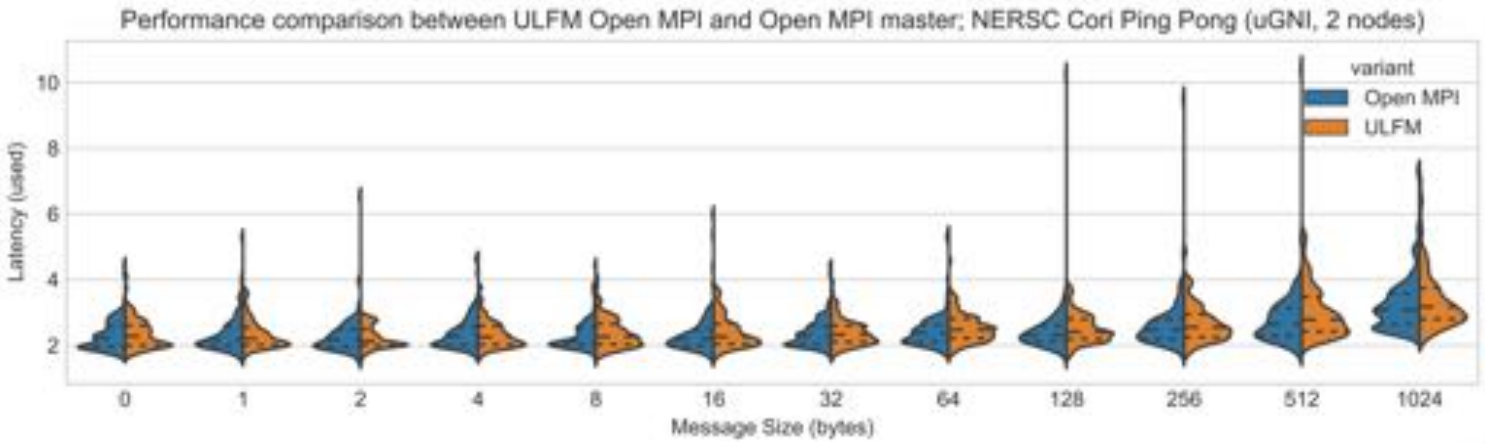
More info, examples and resources available

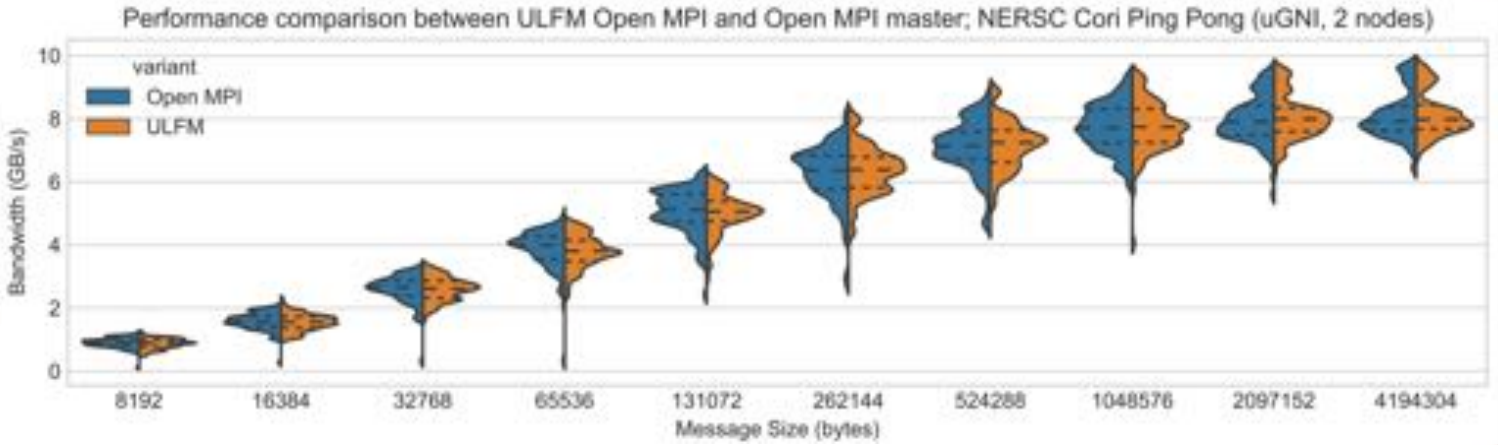
http://fault-tolerance.org

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

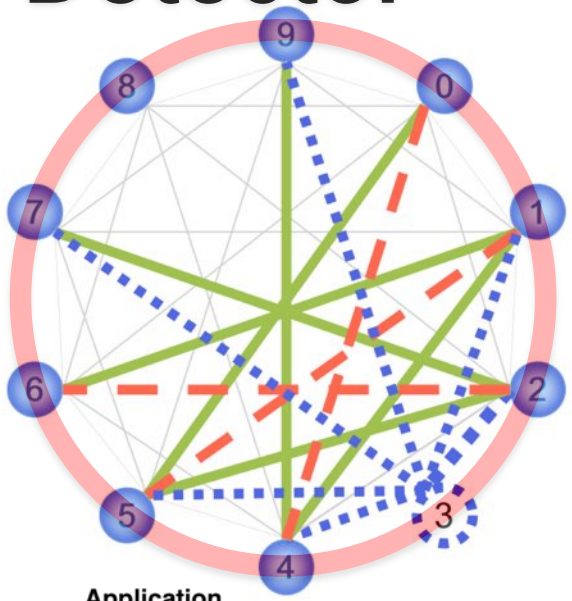
ULFM MPI: Software Infrastructure

- Implementation in Open MPI, MPICH available
- No performance impact
- Open MPI ULFM 2.0 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 mem
  - 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!





Scalable Failure Detector



f = supported number of overlapping failures

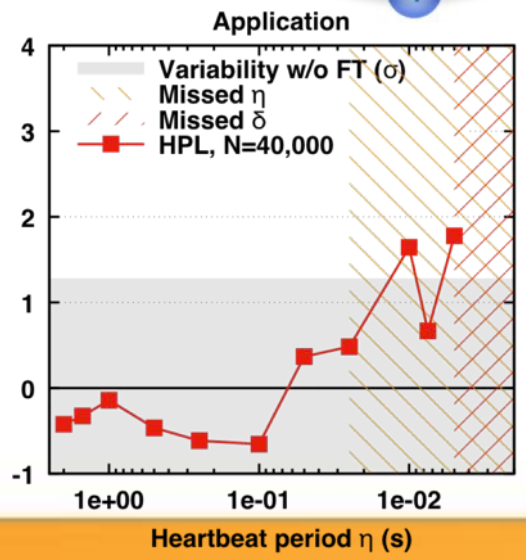
Stabilization Time T(f) = duration of the longest sequence of non stable configurations assuming at most f overlapping faults

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

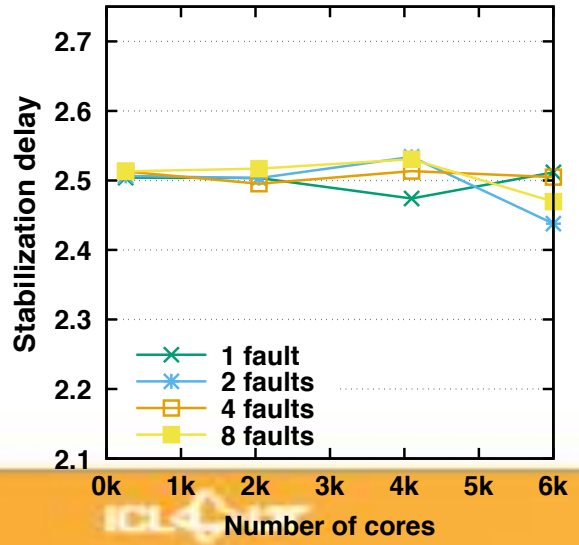
$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{\text{reconnect}} + \underbrace{\frac{f(f+1)}{2}B(n)}_{\text{propagate}}$$

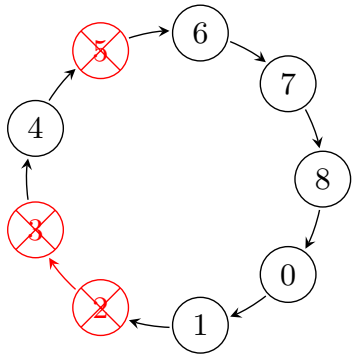
The broadcast algorithm can tolerate up to $\lfloor log(n) \rfloor$ overlapping failures, thus

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



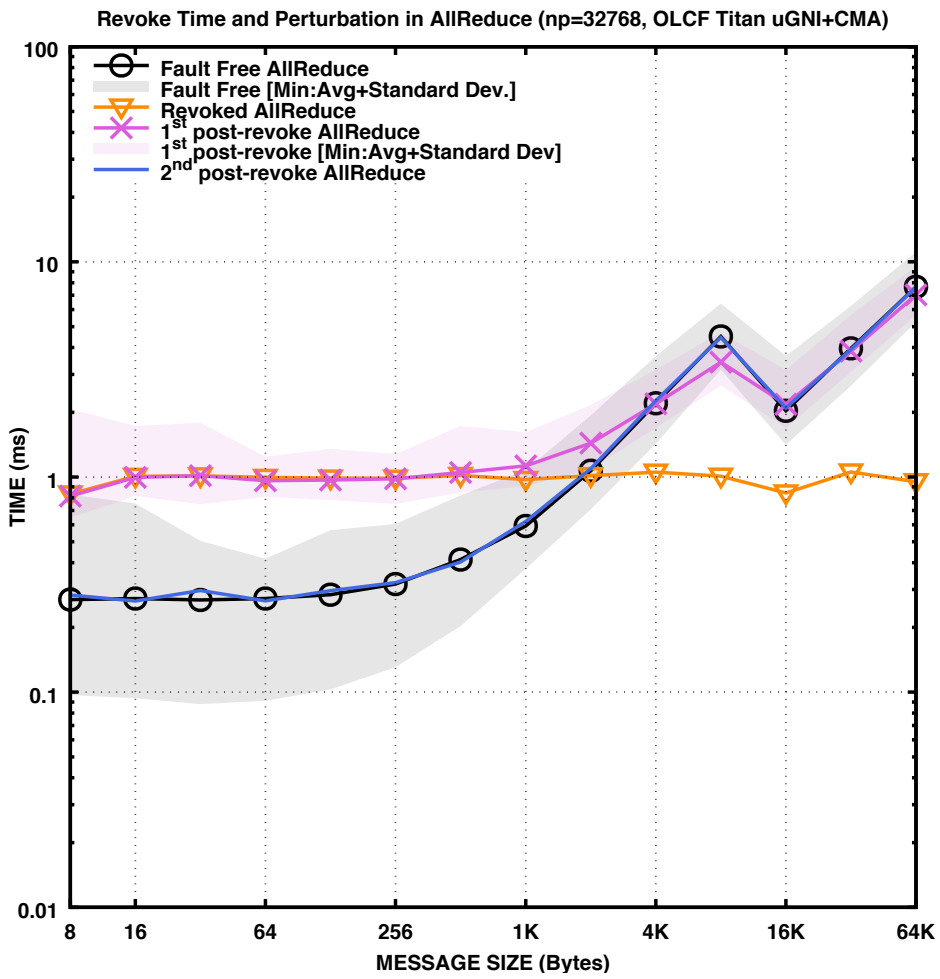
Timeout for suspecting a failure 2.5s





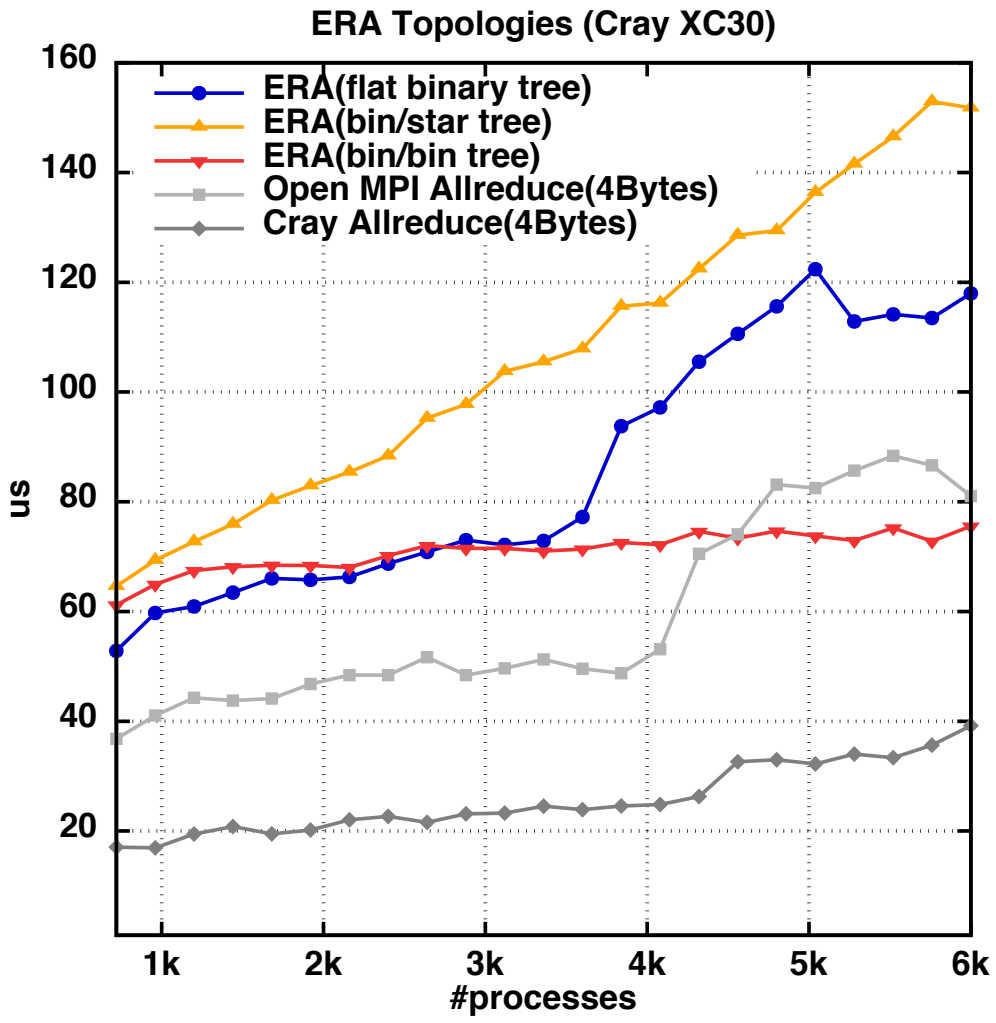
Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J. "Failure Detection and Propagation in HPC systems," SuperComputing, Salt Lake City, UT, November, 2016

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 2nd Allreduce (approximately 1ms on 32k processes), 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)

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

ADVANCED CONTENT

Inside MPIX_COMM_REPLACE

See 10.respawn

```
30  if( comm == MPI_COMM_NULL ) { /* am I a new process? */
31      /* I am a new spawnee, waiting for my new rank assignment
32       * it will be sent by rank 0 in the old world */
33      MPI_Comm_get_parent(&icomm);
35      MPI_Recv(&crank, 1, MPI_INT, 0, 1, icomm, MPI_STATUS_IGNORE);
…
40   }
41   else {
42      /* I am a survivor: Spawn the appropriate number
43       * of replacement
45      /* First: remove dead processes */
46      MPIX_Comm_shrink(comm, &scomm);
47      MPI_Comm_size(scomm, &ns);
48      MPI_Comm_size(comm, &nc);
49      nd = nc-ns; /* number of deads */
50      if( 0 == nd ) {
51         /* Nobody was dead to start with. We are done here */
…
54         return MPI_SUCCESS;
55      }
56      /* We handle failures during this function ourselves... */
57      MPI_Comm_set_errhandler( scomm, MPI_ERRORS_RETURN );
59      rc = MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
60                          0, scomm, &icomm, MPI_ERRCODES_IGNORE);
```

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

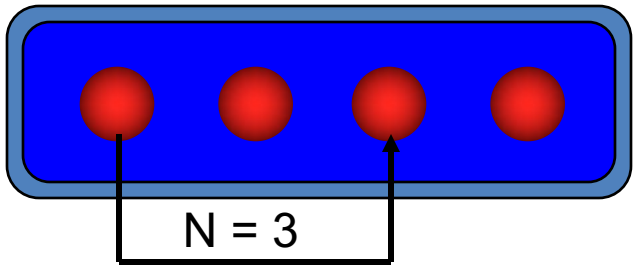
Spawn *nd* new processes

Intercommunicators – P2P

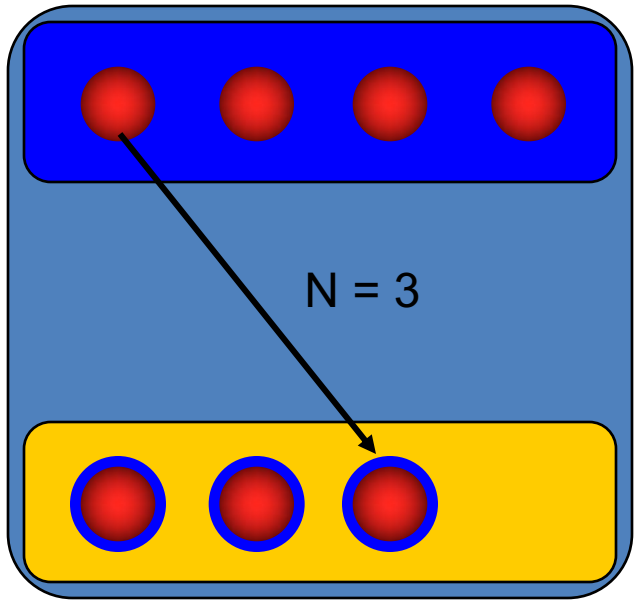
On process 0:

```
MPI_Send( buf, MPI_INT, 1, n, tag, intercomm )
```

- Intracommunicator

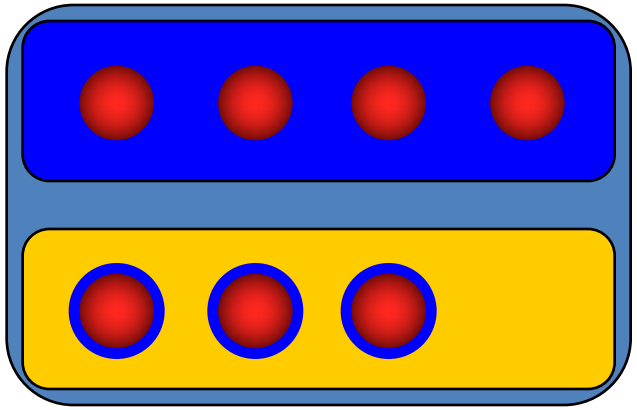


- Intercommunicator



Intercommunicators

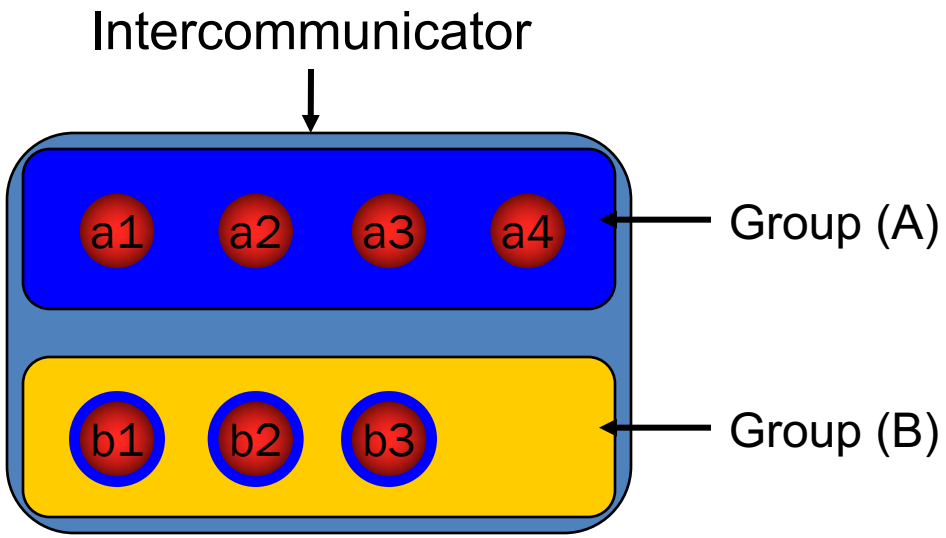
- And what's a intercommunicator ?



- 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

```
59      rc = MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
60                          0, scomm, &icomm, MPI_ERRCODES_IGNORE);
61      flag = (MPI_SUCCESS == rc);
62      MPIX_Comm_agree(scomm, &flag);
63      if( !flag ) {
64         if( MPI_SUCCESS == rc ) {
65            MPIX_Comm_revoke(icomm);
66            MPI_Comm_free(&icomm);
67         }
68         MPI_Comm_free(&scomm);
…
70         goto redo;
71      }
```

Check if spawn worked (using the shrink comm)

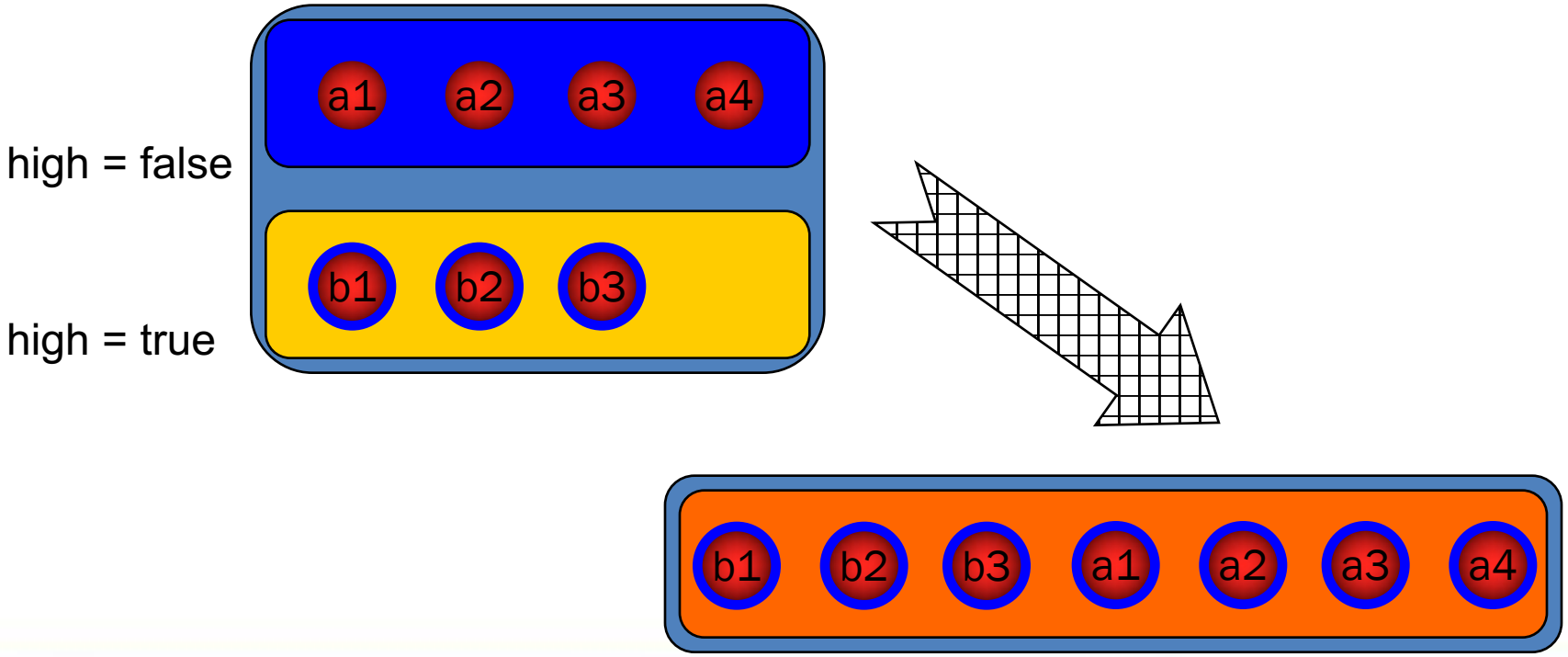
If not, make the spawnees abort with MPI_ERR_REVOKE

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

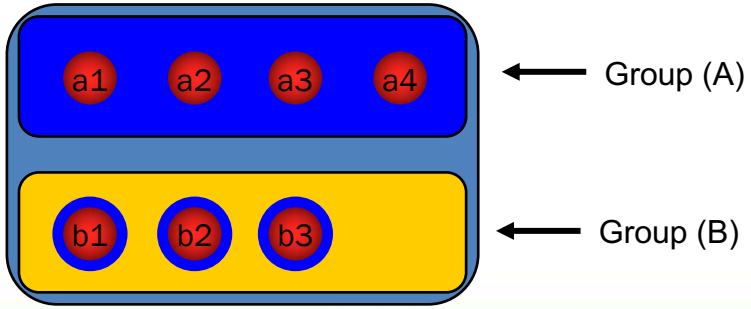
```
95      /* Merge the intercomm, to reconstruct an intracomm (we check
96       * that this operation worked before we proceed further) */
97      rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
98      rflag = flag = (MPI_SUCCESS==rc);
99      MPIX_Comm_agree(scomm, &flag);
100     if( MPI_COMM_WORLD != scomm ) MPI_Comm_free(&scomm);
101     MPIX_Comm_agree(icomm, &rflag);
102     MPI_Comm_free(&icomm);
103     if( !(flag && rflag) ) {
…
108         goto redo;
109     }
```

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

```
130     /* restore the error handler */
131     if( MPI_COMM_NULL != comm ) {
132         MPI_Errhandler errh;
133         MPI_Comm_get_errhandler( comm, &errh );
134         MPI_Comm_set_errhandler( *newcomm, errh );
135     }
```

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

```
 74          /* remembering the former rank: we will reassign the same
 75           * ranks in the new world. */
 76        MPI_Comm_rank(comm, &crank);
 77        MPI_Comm_rank(scomm, &srank);
 78          /* the rank 0 in the scomm comm is going to determine the
 79           * ranks at which the spares need to be inserted. */
 80        if(0 == srank) {
 81              /* getting the group of dead processes:
 82               *   those in comm, but not in scomm are the deads */
 83           MPI_Comm_group(comm, &cgrp);
 84           MPI_Comm_group(scomm, &sgrp);
 85              MPI_Group_difference(cgrp, sgrp, &dgrp);
 86              /* Computing the rank assignment for the newly inserted spares
*/
 87           for(i=0; i<nd; i++) {
 88               MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
 89                   /* sending their new assignment to all new procs */
 90               MPI_Send(&drank, 1, MPI_INT, i, 1, icomm);
 91           }
```

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

```
    /* 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

```
 19 int MPIX_Comm_replace(MPI_Comm worldwspares, MPI_Comm comm, MPI_Comm
*newcomm) {
…
 25     /* First: remove dead processes */
 26     MPIX_Comm_shrink(worldwspares, &shrinked);
 27     /* We do not want to crash if new failures come... */
 28     MPI_Comm_set_errhandler( shrinked, MPI_ERRORS_RETURN );
 29     MPI_Comm_size(shrinked, &ns); MPI_Comm_rank(shrinked, &srank);
 30
 31     if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
 32         /* not enough processes to continue, aborting. */
 33         MPI_Comm_size(comm, &nc);
 34         if( nc > ns ) MPI_Abort(worldwspares, MPI_ERR_PROC_FAILED);
 35
 36         /* remembering the former rank: we will reassign the same
 37          * ranks in the new world. */
 38         MPI_Comm_rank(comm, &crank);
 40         /* >>??? is crank the same as srank ???<<< */
 42     } else { /* I was a spare, waiting for my new assignment */
 44     }
 45     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

```
31  if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
…
36    /* remembering the former rank: we will reassign the same
37     * ranks in the new world. */
38    MPI_Comm_rank(comm, &crank);
39
40    /* the rank 0 in the shrinked comm is going to determine the
41     * ranks at which the spares need to be inserted. */
42    if(0 == srank) {
43      /* getting the group of dead processes:
44       *   those in comm, but not in shrinked are the deads */
45      MPI_Comm_group(comm, &cgrp); MPI_Comm_group(shrinked, &sgrp);
46      MPI_Group_difference(cgrp, sgrp, &dgrp); MPI_Group_size(dgrp, &nd);
47      /* Computing the rank assignment for the newly inserted spares */
48      for(i=0; i<ns-(nc-nd); i++) {
49        if( i < nd ) MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
50        else drank=-1; /* still a spare */
51        /* sending their new assignment to all spares */
52        MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
53      }
…
55    }
56  } else { /* I was a spare, waiting for my new assignment */
57    MPI_Recv(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
58  }
```

Inserting the spares in world

```
31  if(MPI_COMM_NULL != comm) { /* I was not a spare before... */
…
36    /* remembering the former rank: we will reassign the same
37     * ranks in the new world. */
38    MPI_Comm_rank(comm, &crank);
…
51        /* sending their new assignment to all spares */
52        MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
…
56  } else { /* I was a spare, waiting for my new assignment */
57    MPI_Recv(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
58  }
60  /* Split does the magic: removing spare processes and reordering ranks
61   * so that all surviving processes remain at their former place */
62  rc = MPI_Comm_split(shrinked, crank<0?MPI_UNDEFINED:1, crank, newcomm);
…
67  flag = MPIX_Comm_agree(shrinked, &flag);
68  MPI_Comm_free(&shrinked);
69  if( MPI_SUCCESS != flag ) {
70    if( MPI_SUCCESS == rc ) MPI_Comm_free( newcomm );
71    goto redo;
72  }
73  return MPI_SUCCESS;
```

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

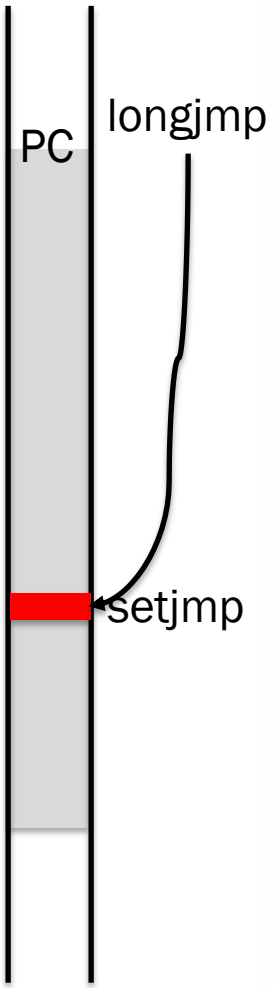
See ex3.1.shrinkspares_reorder.c

Respawn in action: buddy C/R

See 12.buddycr.c

```
109     MPI_Comm_get_parent( &parent );
110     if( MPI_COMM_NULL == parent ) {
111         /* First run: Let's create an initial world,
112          * a copy of MPI_COMM_WORLD */
113         MPI_Comm_dup( MPI_COMM_WORLD, &world );
…
116     } else {
117         /* I am a spare, lets get the repaired world */
118         app_needs_repair(MPI_COMM_NULL);
119     }
…
184    setjmp(jmpenv);
185    while(iteration < max_iterations) {
186        /* take a checkpoint */
187        if(0 == iteration%2) app_buddy_ckpt(world);
188        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

```
121 static int app_needs_repair(void) {
122     MPI_Comm tmp;
123     MPIX_Comm_replace(world, &tmp);
124     if( tmp == world ) return false;
125     if( MPI_COMM_NULL != world) MPI_Comm_free(&world);
126     world = tmp;
127     app_reload_ckpt(world);
128     /* Report that world has changed and we need to re-execute */
129     return true;
130 }
131
132 /* Do all the magic in the error handler */
133 static void errhandler_respawn(MPI_Comm* pcomm, int* errcode, ...) {
 …
142     if( MPIX_ERR_PROC_FAILED != eclass &&
143         MPIX_ERR_REVOKED != eclass ) {
144         MPI_Abort(MPI_COMM_WORLD, *errcode);
145    }
146     MPIX_Comm_revoke(*pcomm);
147     if(app_needs_repair()) longjmp(jmpenv, 0);
148 }
```

- 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

```
 49 static int app_buddy_ckpt(MPI_Comm comm) {
 50     if(0 == rank || verbose) fprintf(stderr, "Rank %04d: checkpointing to %04d after iteration
%d\n", rank, rbuddy(rank), iteration);
 51     /* Store my checkpoint on my "right" neighbor */
 52     MPI_Sendrecv(mydata_array, count, MPI_DOUBLE, rbuddy(rank), ckpt_tag,
 53                  buddy_ckpt,   count, MPI_DOUBLE, lbuddy(rank), ckpt_tag,
 54                  comm, MPI_STATUS_IGNORE);
 55     /* Commit the local changes to the checkpoints only if successful. */
 56     if(app_needs_repair()) {
 57         fprintf(stderr, "Rank %04d: checkpoint commit was not successful, rollback instead\n",
rank);
 58         longjmp(jmpenv, 0);
 59     }
 60     ckpt_iteration = iteration;
 61     /* Memcopy my own memory in my local checkpoint (with datatypes) */
 62     MPI_Sendrecv(mydata_array, count, MPI_DOUBLE, 0, ckpt_tag,
 63                  my_ckpt, count, MPI_DOUBLE, 0, ckpt_tag,
 64                  MPI_COMM_SELF, MPI_STATUS_IGNORE);
 65     return MPI_SUCCESS;
 66 }
```

See 12.buddycr.c