Resilient applications using MPI-level constructs

2014 SC'14 Fault Tolerant MPI Tutorial

ICL UT
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

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

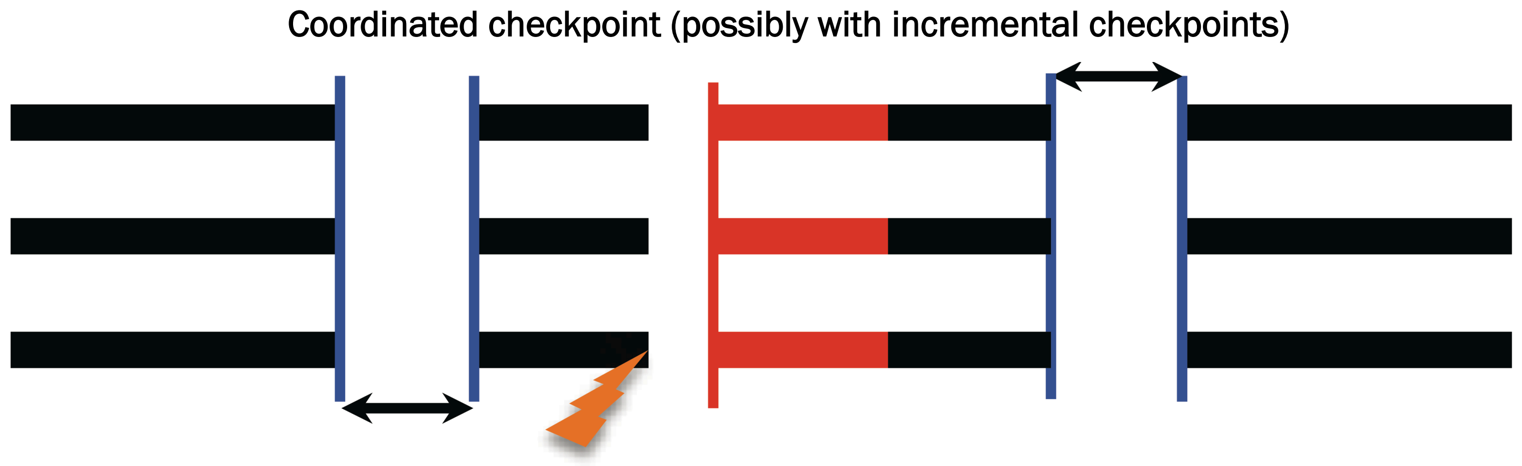
- Can be attached to all objects allowing data transfers: communicators, windows and files
- Allow for minimalistic error recovery: the standard suggests only non-MPI related actions
- All newly created MPI objects inherit the error handler from their parent
  - A global error handler can be specified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

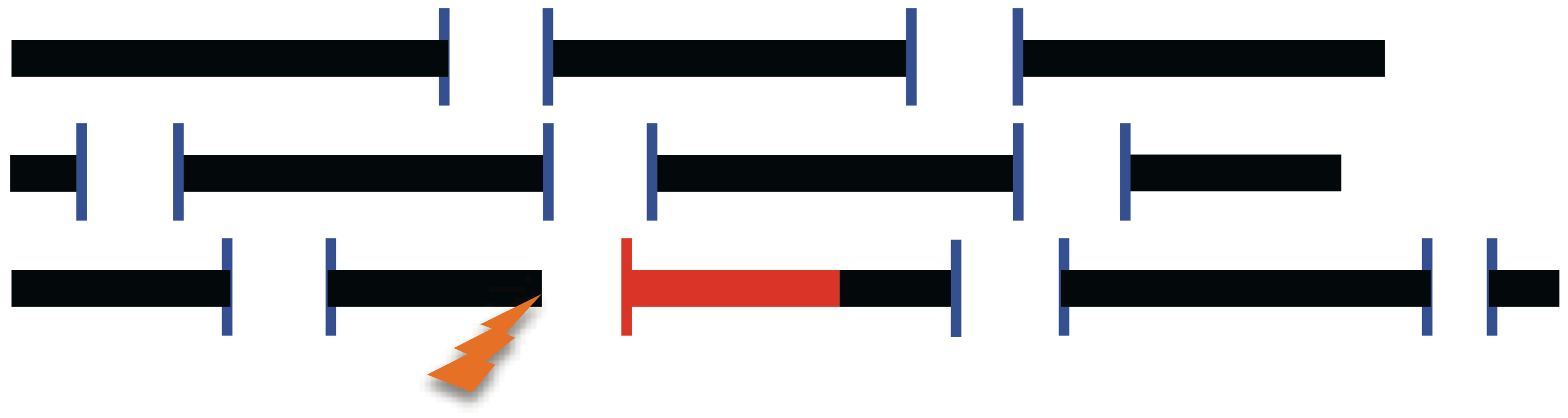
- MPI_Comm_create_errhandler(errh, errhandler_fct)
  - Declare an error handler with the MPI library
- MPI_Comm_set_errhandler(comm, errh)
  - Attach a declared error handler to a communicator
  - Newly created communicators inherits the error handler that is associated with their parent
  - Predefined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

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 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

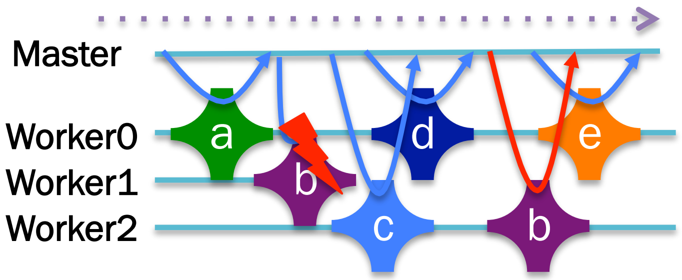
Uncoordinated C/R

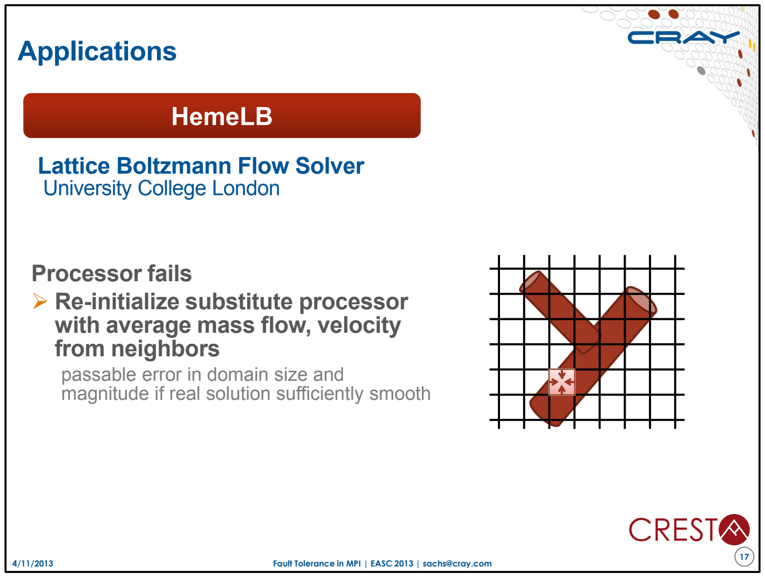


- Checkpoints taken independently
- Based on variants of Message Logging
- 1 fails, 1 rollback
- Can be implemented w/o a standardized user API
- Benefit from ULFM: implementation becomes portable across multiple MPI libraries

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) ☹





Applications

HemeLB

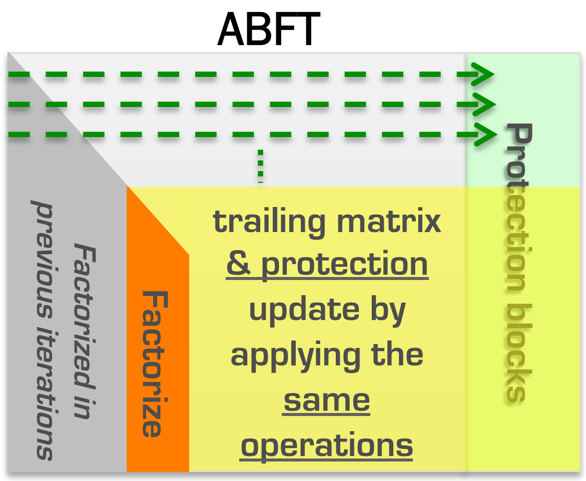
Lattice Boltzmann Flow Solver
University College London

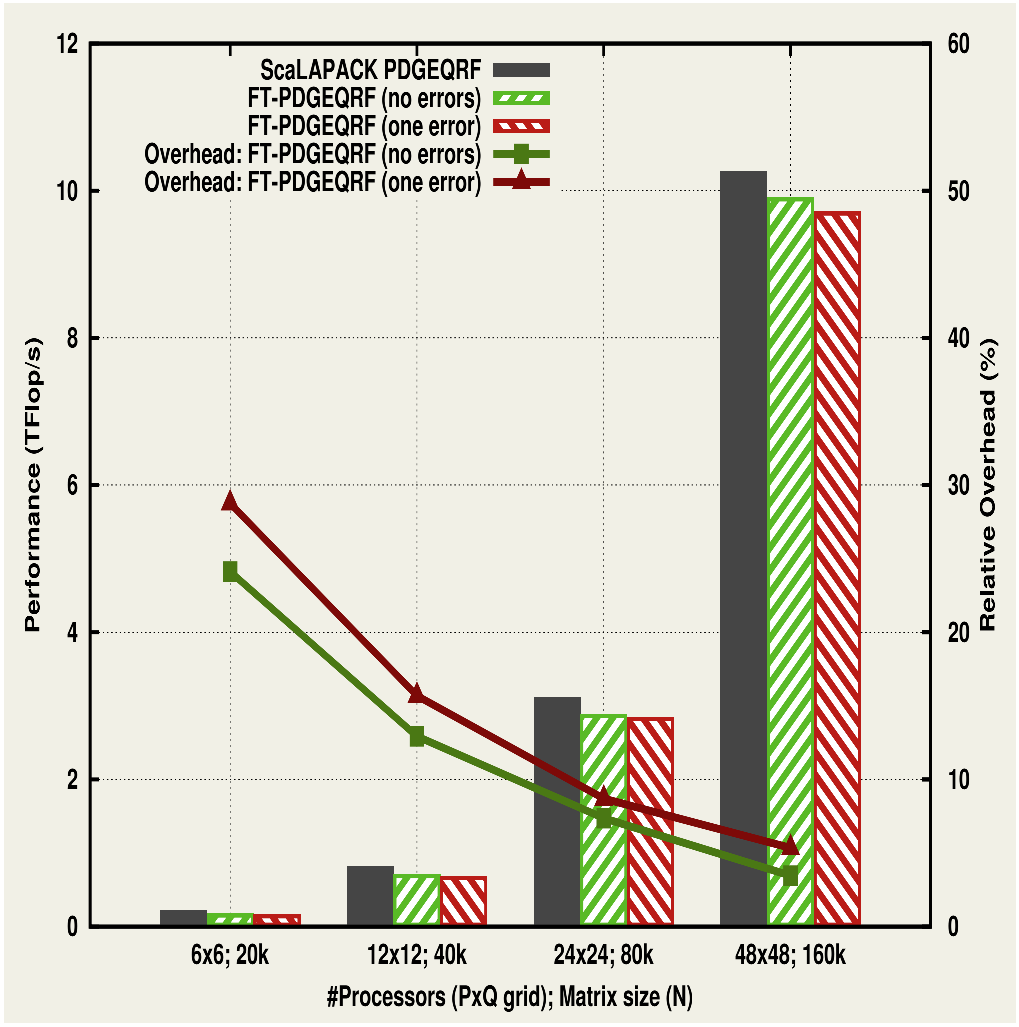
Processor fails
- **Re-initialize substitute processor with average mass flow, velocity from neighbors**
  passable error in domain size and magnitude if real solution sufficiently smooth

4/11/2013 Fault Tolerance in MPI | EASC 2013 | sachs@cray.com 17

Application specific forward recovery

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

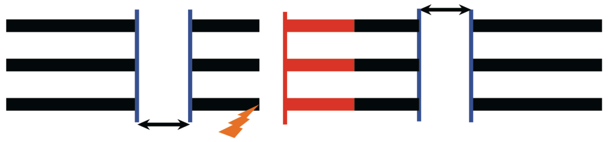

ABFT
Protection blocks
Factorized in previous iterations
Factorize
trailing matrix & protection update by applying the same operations



An API for diverse FT approaches

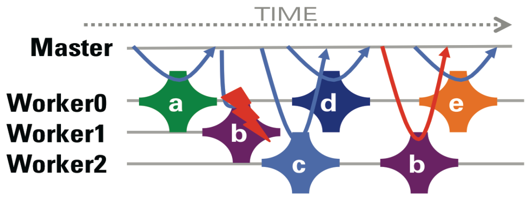
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

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.

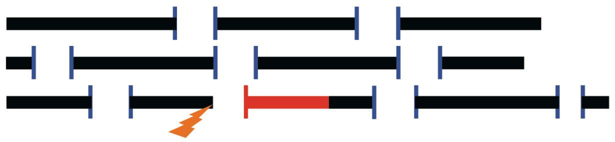
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

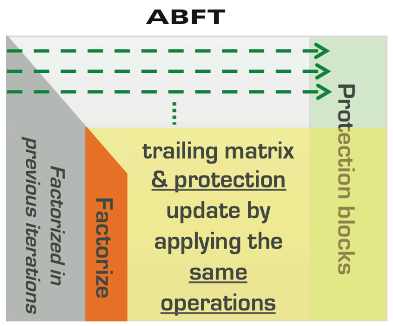
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

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

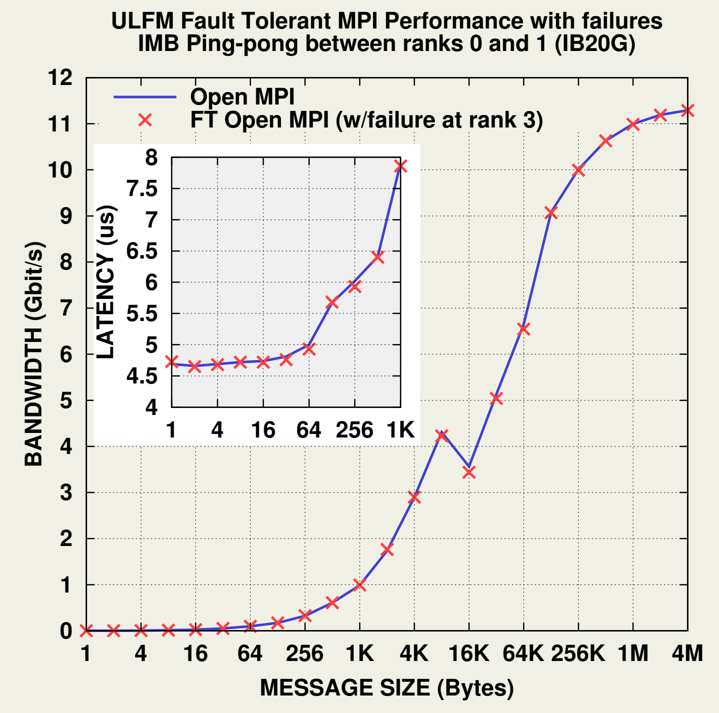

ABFT
Factorized in previous iterations
Factorize
trailing matrix & protection update by applying the same operations
Protection blocks

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

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.

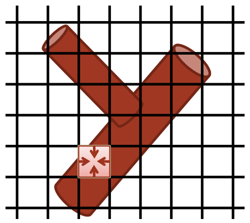
Applications

HemeLB

Lattice Boltzmann Flow Solver
University College London

Processor fails
- **Re-initialize substitute processor with average mass flow, velocity from neighbors**
  passable error in domain size and magnitude if real solution sufficiently smooth

Long running computations
- **Small errors can be eliminated by numerical procedure**



Credits: ETH Zurich

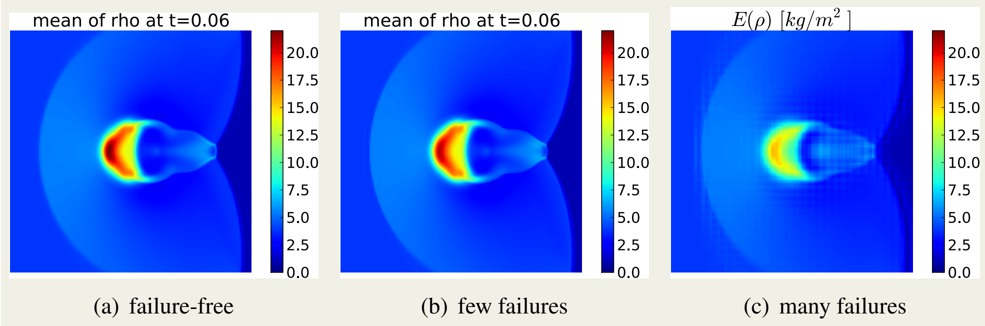
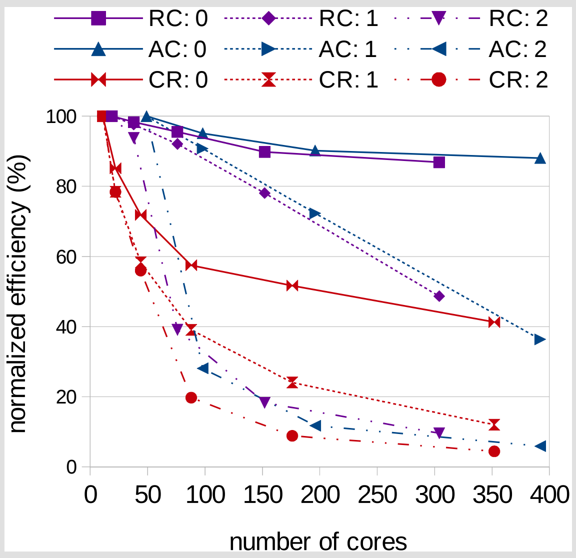


Figure 5. Results of the FT-MLMC implementation for three different failure scenarios.

- **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
- …

Tens of papers about ULFM last year alone.

12 Results: Scalability



- results on OPL cluster, max. resolution of 2^{13}
- in terms of absolute time, CR is always more longer (however, uses fewer processes)
- RC and AC also show best scalability
- plots for 2 failures erratic due to high overheads in β version of ULFM MPI

OPL cluster node: 2x6 cores Xeon5670, QDR IB

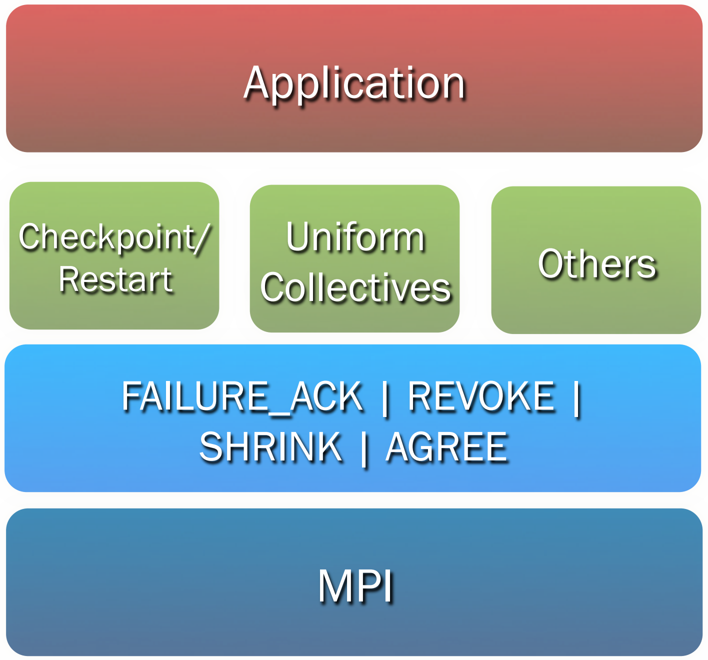
RC=Replication/resampling
AC=Alternate recombination
CR=Checkpoint/Restart

Part rationale, part examples

ULFM MPI API

Minimal Feature Set for FT MPI

- Failure Notification
- Error Propagation
- Error Recovery

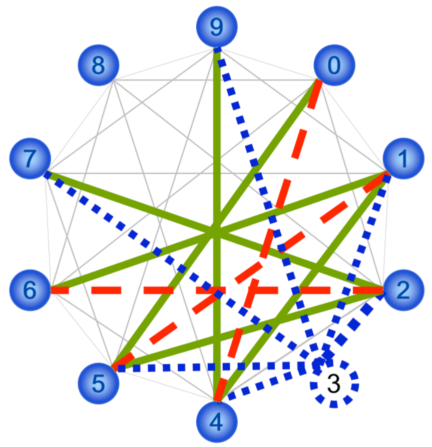


Not all recovery strategies require all of these features, that's why the interface splits notification, propagation and recovery.

ULFM is not a recovery strategy, but a minimalistic set of building blocks for more 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 might 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 couple application ?

Error Recovery

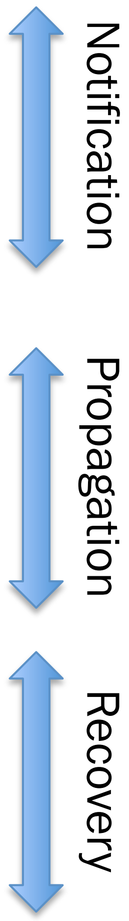
- What is the right recovery strategy?
- Keep going with the remaining processes?
- Shrink the living processes to form a new consistent communicator?
- Spawn new processes to take the place of the failed ones?
- Who should be in charge of defining this survival strategy? What would be the application 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, 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)
  - Agree on the AND value on binary mask, ignoring failed processes (reliable AllReduce)



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

MPI_Comm_revoke

- Communicator level failure propagation
- The revocation of a communicator complete 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_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

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 raise MPI_ERR_PROC_FAILED

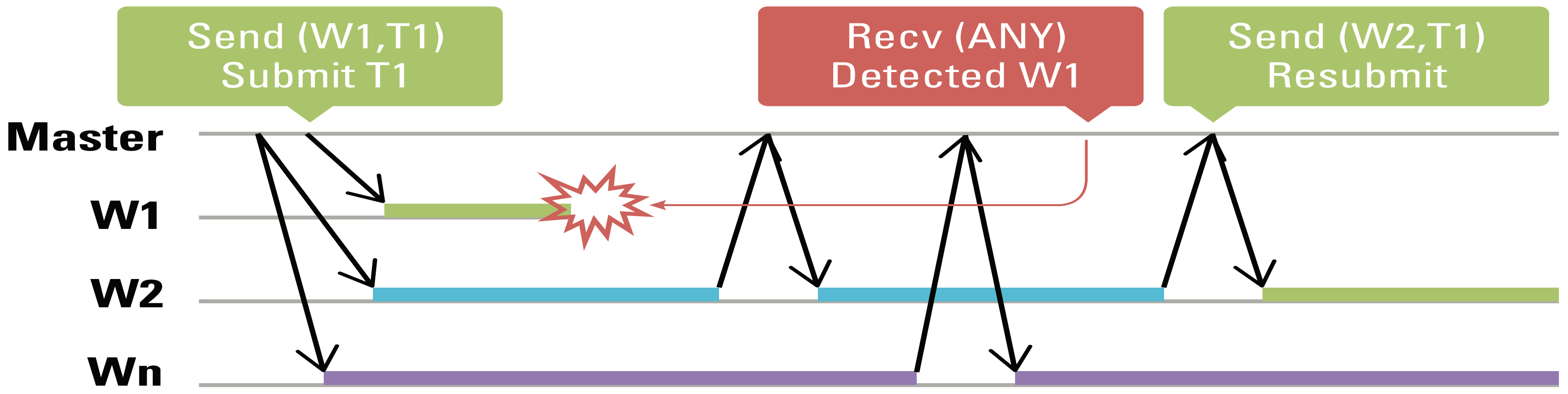
Other mechanisms

- 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.

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)

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

Regaining Control

```
legacy_code(void) {
  /* in legacy non FT code, this Recv may deadlock.
   * the runtime is expected to abort the job
   * and do resource cleanup. No opportunity for
   * recovering gracefully. */
  MPI_Recv(buff, count, datatype,
           src, tag, comm,
           MPI_STATUS_IGNORE);
}

ft_code(void) {
  /* MPI_Recv garanteed to return control to the
   * App if src is dead. */
  rc = MPI_Recv(buff, count, datatype,
                src, tag, comm,
                &status);
  /* restartless recovery becomes possible */
  if( MPI_ERR_PROC_FAILED == rc ) recover();
}
```

- If 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

Regaining Control: ANY_SOURCE

```
ft_code_any(void) {
  int NBR, nfailed=0;
  MPI_Group_size( sendergrp, &NBR );
  for(nbrecv = 0; (nbrecv+nfailed)<NBR; nbrecv++) {
    rc = MPI_Recv(buff, count, datatype,
                MPI_ANY_SOURCE, tag, comm,
                &statusany);
    if( MPI_ERR_PROC_FAILED == rc )
      nfailed = nbsendersfailed( sendergrp );
  }
}
nbsendersfailed(MPI_Group sendergrp) {
/* Count how many of the ANY_SOURCE recv we
 * should repost */
  int nfailed;
  MPI_Group failedgrp, igrp;
  MPI_Comm_failure_ack(comm);
  MPI_Comm_failure_get_acked(comm, &failedgrp);
  MPI_Group_intersection( failedgrp, sendergrp,
                         &igrp );
  MPI_Group_size( igrp, &nfailed );
  MPI_Group_free( &igrp );
  MPI_Group_free( &failedgrp );
  return nfailed;
}
```

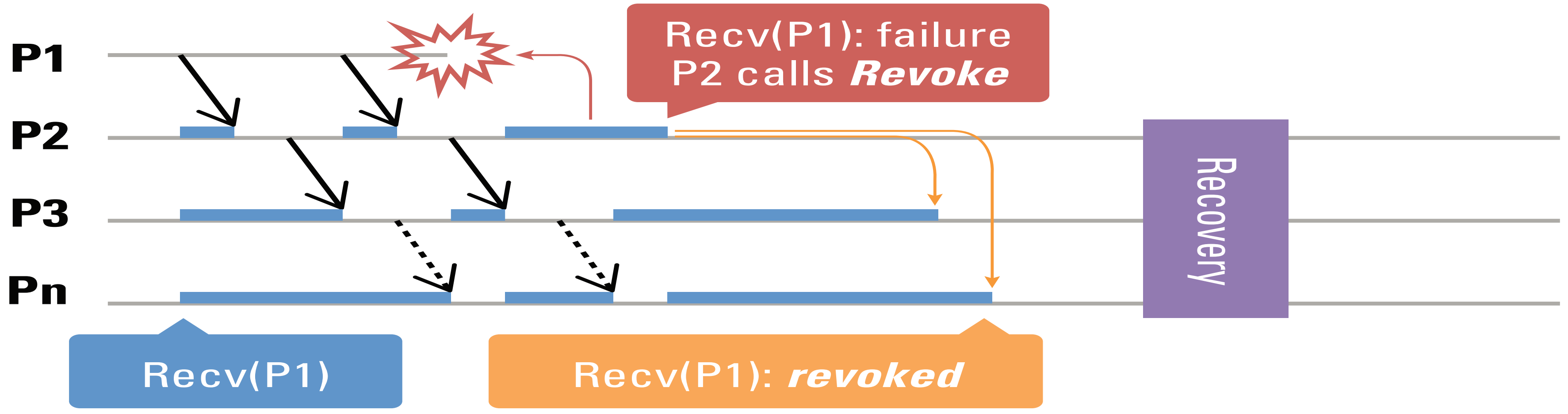
- If the recv uses ANY_SOURCE:
  - Any failure in the comm is potentially a failure of the matching sender!
  - To avoid deadlocking, the recv must be interrupted in any case
  - Application uses new interfaces to inspect the list of failed processes, determine if the ANY_SOURCE receive needs to be reissued

ANY_SOURCE and matching

```
ft_code_any(void) {
  for(i=0; i<nbrecv; i++) {
    MPI_Irecv(buff, count, datatype,
              MPI_ANY_SOURCE, tag, comm, &reqs[i]);
  }
  MPI_Irecv(buff, count, datatype,
            1, tag, comm, &req);
  do {
    rc = MPI_Waitall(nbrecv, reqs, statuses);
    if( MPI_SUCCESS != rc ) {
       int nfailed = nbsendersfailed(sendergrp);
           i=nbrecv;
       while(nfailed) {
         i--;
         if( statuses[i].MPI_ERROR ==
               MPI_ERR_PROC_FAILED ) {
           nfailed--;
         }
         if( statuses[i].MPI_ERROR ==
               MPI_ERR_PROC_FAILED_PENDING ) {
           MPI_Cancel(reqs[i]);
           MPI_Request_free(reqs[i]);
           nfailed--;
         }
                  }
  } while( MPI_SUCCESS != rc )
```

- Non-blocking operations
  - Interrupting non-blocking ANY_SOURCE could change matching order, uh oh…
  - New error code: the operation is interrupted by a process failure, but is still *pending*
  - Can be completed again, if the application knows its safe, matching order respected

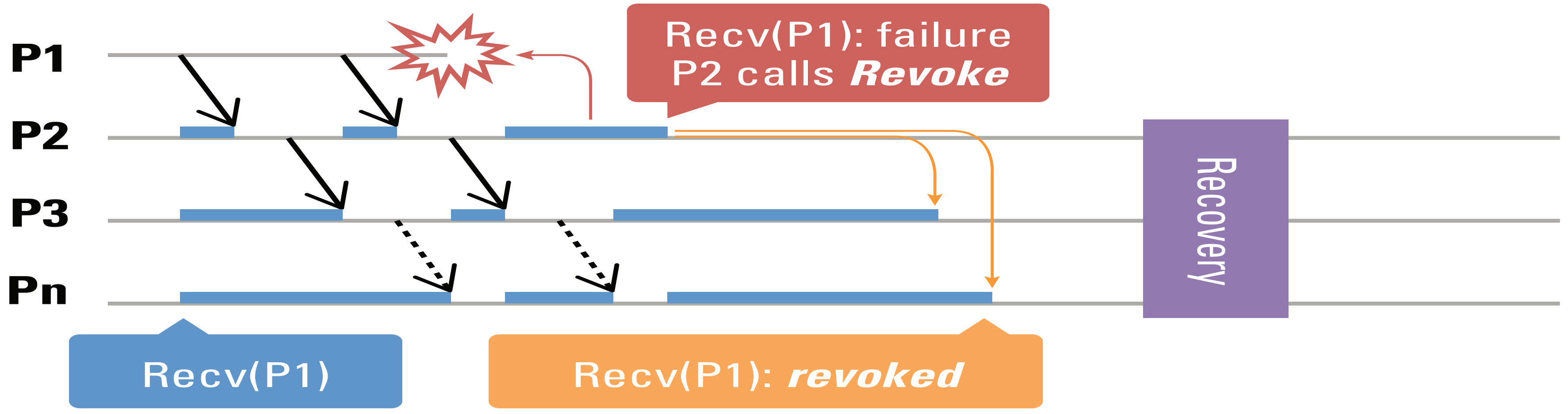
Resolving transitive dependencies



```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED) recovery(comm);
}
deadlocking_transitive_deps(void) {
  for(i=0; i<nbrecv; i++) {
    if(myrank>0) MPI_Irecv(buff, count, datatype,
                        myrank-1, tag, comm, &req);
    if(myrank<n) MPI_Send(buff2, count, datatype,
                        myrank+1, tag, comm, &req);
  }
}
```

- P1 fails
- P2 raises an error and wants to change comm pattern to do application recovery
- but P3..Pn are stuck in their posted recv
- P2 can unlock them with Revoke ☺
- P3..Pn join P2 in the recovery

Resolving transitive dependencies



```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED ||
     err == MPI_ERR_REVOKED ) {
    MPI_Comm_revoke(comm);
    recovery(comm);
  }
}
ft_transitive_deps(void) {
  for(i=0; i<nbrecv; i++) {
    if(myrank>0) MPI_Irecv(buff, count, datatype,
                         myrank-1, tag, comm, &req);
    if(myrank<n) MPI_Send(buff2, count, datatype,
                         myrank+1, tag, comm, &req);
  }
}
```

- P1 fails
- P2 raises an error and wants to change comm pattern to do application recovery
- but P3..Pn are stuck in their posted recv
- P2 can unlock them with **Revoke** ☺
- P3..Pn join P2 in the recovery

Errors and Collective Operations

```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED) recovery(comm);
}

deadlocking_collectives(void) {
  for(i=0; i<nbrecv; i++) {
     MPI_Bcast(buff, count, datatype, 0, comm);
  }
}
```

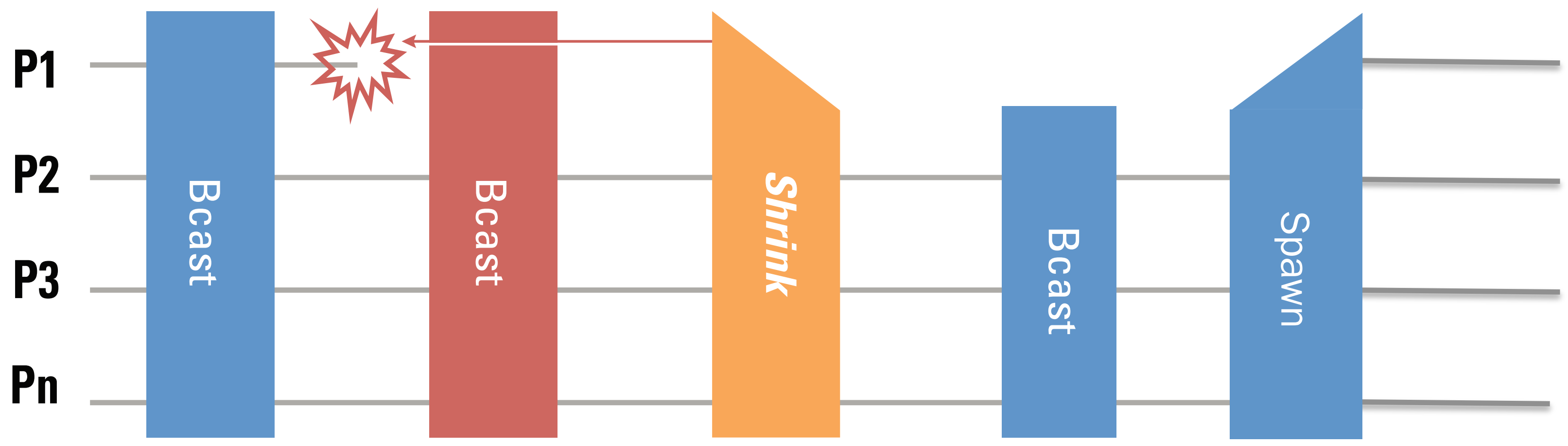
- Exceptions are raised only at ranks where the Bcast couldn't succeed (lax consistency)
  - In a tree-based Bcast, only the subtree under the failed process sees the failure
  - Other ranks succeed and proceed to the next Bcast
  - Ranks that couldn't complete enter "recovery", do not match the Bcast posted at other ranks => deadlock ☹

Errors and Collective Operations

```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED ||
     err == MPI_ERR_REVOKED ) recovery(comm);
}

deadlocking_collectives(void) {
  for(i=0; i<nbrecv; i++) {
     MPI_Bcast(buff, count, datatype, 0, comm);
  }
}
```

- Exceptions are raised only at ranks where the Bcast couldn't succeed (lax consistency)
  - In a tree-based Bcast, only the subtree under the failed process sees the failure
  - Other ranks succeed and proceed to the next Bcast
  - Ranks that couldn't complete enter "recovery", do not match the Bcast posted at other ranks => MPI_Comm_revoke(comm) interrupts unmatched Bcast and forces an exception (and triggers recovery) at all ranks

Full Recovery



- Restores full communication capability (all collective ops, etc).
- MPI_COMM_SHRINK(comm, newcomm)
  - Creates a new communicator excluding failed processes
  - New failures are absorbed during the operation
  - The communicator can be restored to full size with MPI_COMM_SPAWN

A cookbook of the most useful techniques

HANDS ON

- mpicc x.c –o x
  mpirun -np 8 -am ft-enable-mpi ./x

Your first resilient application

```
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();
}
```

- What do we obtain upon failure of the single process?
- What are we missing in order to get the expected output?

Slightly more complex

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

   MPI_Init(NULL, NULL);
   MPI_Comm_set_errhandler(MPI_COMM_WORLD,
               MPI_ERRORS_RETURN);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if( rank == (size-1) ) raise(SIGKILL);

   rc = MPI_Barrier(MPI_COMM_WORLD);
   MPI_Error_string( rc, errstr, &len );
   printf("Rank %d / %d (error %s)\n",
          rank, size, errstr);

   MPI_Finalize();
}
```

- Will this code deadlock?
  - It is guaranteed by the standard that the fail process error will eventually propagate
  - Some processes will detect the failure themselves (if the barrier algorithm create communications between them and the dead process)
  - Others will be informed either by the runtime (OOB) or by revoking the internal communicator used for the collectives.

Who is the dead process?

```
/* usual initialization */
if( rank == (size-1) ) raise(SIGKILL);

rc = MPI_Barrier(MPI_COMM_WORLD);
MPI_Error_string( rc, errstr, &len );
if( MPI_ERR_PROC_FAILED == rc ) {
   OMPI_Comm_failure_ack(MPI_COMM_WORLD);
   OMPI_Comm_failure_get_acked(MPI_COMM_WORLD,
                              &group);
   MPI_Comm_group(MPI_COMM_WORLD, &cgroup);
   MPI_Group_size(group, &g_size);
   ranks1 = (int*)malloc(g_size * sizeof(int));
   ranks2 = (int*)malloc(g_size * sizeof(int));
   for(i = 0; i < g_size; ranks1[i] = i, i++ );
   MPI_Group_translate_ranks(group, g_size, ranks1,
                              cgroup, ranks2);
   printf("Rank %d / %d (error %s) [%d dead: ",
          rank, size, errstr, g_size);
   for(i = 0; i < g_size; ranks1[i] = i, i++ )
       printf("%d ", ranks2[i]);
   printf("]\n");
} else
    printf("Rank %d / %d (error NONE)\n", rank, size);
```

- Upon failure one can use OMPI_Comm_failure_ack to acknowledge the known dead processes
- The group of dead processes is then retrieved using OMPI_Comm_failure_get_acked
- A lot of code is needed to print the failed rank
- Can the same code handle multiple failures?

Who are the dead processes?

```
/* usual initialization */
if( rank > (size/2) ) raise(SIGKILL);

rc = MPI_Barrier(MPI_COMM_WORLD);
MPI_Error_string( rc, errstr, &len );
if( MPI_ERR_PROC_FAILED == rc ) {
   OMPI_Comm_failure_ack(MPI_COMM_WORLD);
   OMPI_Comm_failure_get_acked(MPI_COMM_WORLD,
                              &group);
   MPI_Comm_group(MPI_COMM_WORLD, &cgroup);
   MPI_Group_size(group, &g_size);
   ranks1 = (int*)malloc(g_size * sizeof(int));
   ranks2 = (int*)malloc(g_size * sizeof(int));
   for(i = 0; i < g_size; ranks1[i] = i, i++ );
   MPI_Group_translate_ranks(group, g_size, ranks1,
                                cgroup, ranks2);
   printf("Rank %d / %d (error %s) [%d dead: ",
            rank, size, errstr, g_size);
   for(i = 0; i < g_size; ranks1[i] = i, i++ )
       printf("%d ", ranks2[i]);
   printf("]\n");
} else
    printf("Rank %d / %d (error NONE)\n", rank, size);
```

- It is a distributed system!
  - A single dead process is enough to force a process out of the barrier
  - Thus it is possible that different processes return from the barrier for different reasons
- The group of failed processes returned by OMPI_Comm_failure_ack is not consistent!

MORE COMPLICATED EXAMPLES

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

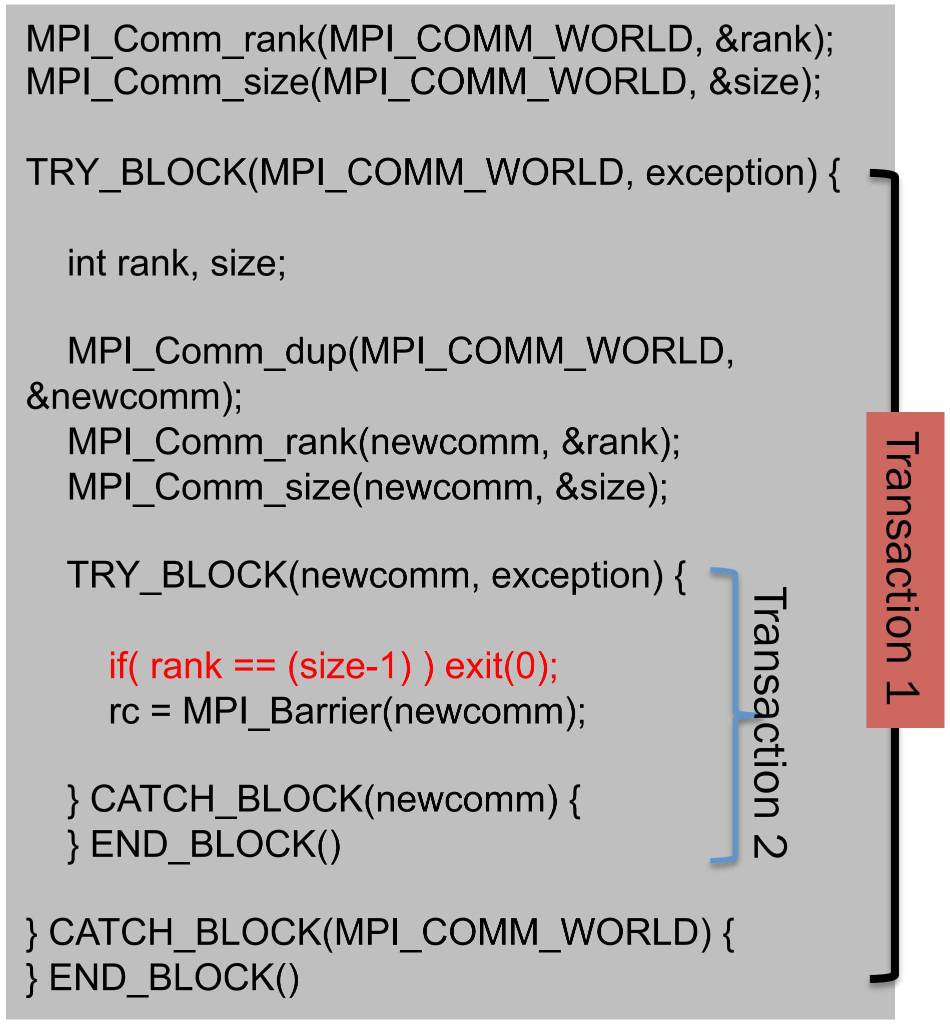
```
/* save data1 to be used in the code below */
transaction1:
TRY_BLOCK(MPI_COMM_WORLD, exception) {

    /* 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()
```

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

CONCLUSION

Thank you

More info, examples and resources available

http://fault-tolerance.org

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE