A quick glance at performance

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.
Continuing through errors

• Full master worker example:
  • Look into the bagoftask directory: errh_blank.f
  • We simulate “blank mode”: communications continue with surviving processes w/o communicator repair
  • (code adapted from “FTMPI” tests, ported to ULFM)
• 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)
Some fun with collective operations

- Look at “ulfm/ex1.ftmpi_ulfm_err_returns.c”
- This program does several Barriers, at some point, a rank commits suicide
- This program doesn’t survive failures, fix it 😊
- Why don’t we need to fix scomm the same as we need to fix fcomm (in that example)?
- Line 70: the program never reaches this abort, why?
More fun with Collectives

• Look now at “ulfm/ex1.ftmpi_ulfm_err_returns-nonuniform.c”

• This program is almost identical to the previous one, but employs Bcast.

• Look at line 78, there are more cases, can you explain why?
Detecting errors (consistently)

• Can you devise a quick way to obtain a globally consistent group of failed processes?

```c
void MPIX_Comm_failures_allget(MPI_Comm comm, MPI_Group * grp) {
??
}
```
Detecting errors (consistently)

- Can you devise a quick way to obtain a globally consistent group of failed processes?

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

- 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

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

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

• Communicator creation functions are collective
• Like all other collective, they may succeed or raise ERR_PROC_FAILED differently at different ranks
• Therefore, caution is needed before using the new communicator: is the context valid at the peer?
• How can you create a wrapper that looks like normal MPI (except for communication cost!), and ensures a safe communicator creation?
• Hint: we need to agree on the success of the split here

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

    rc = MPI_Comm_split(comm, color, key, newcomm);
    flag = (MPI_SUCCESS==rc);

    return rc;
}
```
Creating Communicators, safely

Communicator creation functions are collective
Like all other collective, they may succeed or raise ERR_PROC_FAILED differently at different ranks
Therefore, caution is needed before using the new communicator: is the context valid at the peer?
Can be embedded into wrapper routines that look like normal MPI (except for communication cost!)

Full example in “ex1.ftmpi_ulfm_safecomm_creation.c”

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

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

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

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

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

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

- 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

```c
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);
    }
}
```
Resolving transitive dependencies

- 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

```c
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);
    }
}
```
Avoiding deadlocks

• See example “ex2.ftmpi_ulfm_revoke.c”
  • What do you observe about this program?
  • Why?

• How can we fix this problem?
Iterative Algorithm – with shrink

while( gnorm > epsilon ) {
    iterate();
    compute_norm(&lnorm);

    rc = MPI_Allreduce( &lnorm, &gnorm, 1,
                        MPI_DOUBLE, MPI_MAX, comm);

    if( (MPI_ERR_PROC_FAILED == rc) ||
        (MPI_ERR_COMM_REVOKED == rc) ||
        (gnorm <= epsilon) ) {
        if( MPI_ERR_PROC_FAILED == rc )
            MPI_Comm_revoke(comm);
        allsuceeded = (rc == MPI_SUCCESS);
        MPI_Comm_agree(comm, &allsucceeded);
        if( !allsucceeded ) {
            MPI_Comm_revoke(comm);
            MPI_Comm_shrink(comm, &comm2);
            MPI_Comm_free(comm);
            comm = comm2;
            gnorm = epsilon + 1.0;
        }
    }
}
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
Inserting Spares, at the right place

• See “ex3.ftmpi_ulfm_spares.c”

• We start with extra processes (spares)

• When a failure happens, we will “shrink out” the dead and continue with the same number of processes

• Problem: rank ordering is not preserved
  • But we can fix this! 😊
After Shrink, reordering

• After Shrink, any old (non spare) process knows the list of dead processes

• The new spares have no idea

• Quick solution: rank 0 in shrinked comm assigns the spares to their positions
  • Using “translate rank” to convert the failed group ranks into the original ranks of the failed processes
  • Using Spit to reorder the shrink
MPI_Comm_split

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

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>color</td>
<td>0</td>
<td>⊥</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>⊥</td>
</tr>
<tr>
<td>key</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

3 different colors => 3 communicators
1. {A, D, F, G} with ranks {3, 5, 1, 1} => {F, G, A, D}
2. {C, E, I} with ranks {2, 1, 3} => {E, I, C}
3. {H} with ranks {1} => {H}

B and J get MPI_COMM_NULL as they provide an undefined color (MPI_UNDEFINED)
Inserting replacements (at the right place)

• See “ex4.ftmpi_ulfm_respawn.c”

• We start with the right number of processes. When a failure happens, we will “shrink out” the dead and respawn the missing ranks.

• Problem: rank ordering is not preserved.

• Problem: “spawn” creates an intercomm (not an intracomm).
  • But we can fix this! 😊
Intercommunicators – P2P

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

- Intracommunicator
  - N = 3

- Intercommunicator
  - N = 3
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

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

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

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

![Diagram showing the process of merging intracommunicators with and without the high argument.](image)
int checkpoint_restart(MPI_Comm *comm) {
    int rc, flag;
    checkpoint_in_memory(); // store a local copy of my checkpoint
    rc = checkpoint_to(comm, (myrank+1)%np); // store a copy on myrank+1
    flag = (MPI_SUCCESS==rc); MPI_Comm_agree(comm, &flag);
    if( !flag ) { // if checkpoint fails, we need restart!
        MPI_Comm newcomm; int f_rank; int nf;
        MPI_Group c_grp, n_grp, f_grp;
        redo:
    }
    MPIX_Comm_replace(comm, &newcomm);
    MPI_Comm_group(comm, &c_grp);
    MPI_Comm_group(newgroup, &n_grp);
    MPI_Group_difference(c_grp, n_grp, &f_grp);
    MPI_Group_size(f_grp, &nf);
    for(int i=0; i<nf; i++) {
        MPI_Group_translate_ranks(f_grp, 1, &i, c_grp, &f_rank);
        if( (myrank+np-1)%np == f_rank ) {
            serve_checkpoint_to(newcomm, f_rank);
        }
    }
    MPI_Group_free(&n_grp); MPI_Group_free(&c_grp); MPI_Group_free(&f_grp);
    rc = MPI_Barrier(newcomm);
    flag=(MPI_SUCCESS==rc); MPI_Comm_agree(comm, &flag);
    if( !flag ) goto redo; // again, all free clutter not shown
    restart_from_memory(); // rollback from local memory
    MPI_Comm_free(comm);
    *comm = newcomm;
}
Thank you

More info, examples and resources available

http://fault-tolerance.org
Recreating the world, no spawn

```c
int MPIX_Comm_replace(MPI_Comm worldwspares, MPI_Comm comm, MPI_Comm *newcomm) {
    MPI_Comm shrinked; MPI_Group cgrp, sgrp, dgrp;
    int rc, flag, i, nc, ns, nd, crank, srank, drank;

    redo:
    MPI_Comm_shrink(worldwspares, &shrinked);
    MPI_Comm_size(shrinked, &ns); MPI_Comm_rank(comm, &srank);
    if(MPI_COMM_NULL != comm) {
        MPI_Comm_size(comm, &nc); if( nc > ns ) MPI_Abort(comm, MPI_ERR_INTERN);
        MPI_Comm_rank(comm, &crank);
        MPI_Comm_group(comm, &cgrp);
        MPI_Comm_group(shrinked, &sgrp);
        MPI_Group_difference(cgrp, sgrp, &dgrp); MPI_Group_size(dgrp, &nd);
        if(0 == srank) for(i=0; i<ns-nc-nd; i++) {
            if( i < nd ) MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
            else drank=-1;
            MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
        } // some group free clutter missing
    } else {
        MPI_Recv(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
    }

    rc = MPI_Comm_split(shrinked, crank<0?MPI_UNDEFINED:1, crank, newcomm);
    flag = (MPI_SUCCESS==rc);
    MPI_Comm_agree(shrinked, &flag);
    MPI_Comm_free(&shrinked);
    if( !flag ) goto redo; // some newcomm free clutter missing
    return MPI_SUCCESS;
}
```
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm) {
    MPI_Comm shrinked, spawned, merged;
    int rc, flag, flagr, nc, ns;

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

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