Using ULFM Fault Tolerant MPI
The cookbook

Some simple code snippets for most common use cases

Aurelien Bouteiller & WG

MPI Forum Dec. 2013

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

Reminder: ULFM goals

- What is it? This is an MPI recovery API
  - What it is not: an application recovery strategy/API
  - Once they can communicate again, application employ a fault tolerance strategy of their choosing to repair dataset, continue etc.
- Low overhead
  - MPI implementations must remain mostly unchanged
  - Existing operations should not have stronger semantic that makes them more expensive
- Flexible
  - Different communication/Usage pattern have different needs
  - Pushing the strongest use case leads to poor performance
  - However, all use cases must be expressible (not necessarily straightforward for some, but we'll see here how to cope)

Creating Communicators, safely

```
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;
}
```

- 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?
- This code snippet solves this uncertainty and makes it simple to create comms again
- Can be embedded into wrapper routines that look like normal MPI (except for communication cost!)

Creating Communicators, safely

```
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

When one needs Revoke 1/2

```
void multiple_comm_collectives(grid2d, comm,…) {
    APP_Create_grid2d_comms(grid2d, comm, rowcomm, colcomm);
    rc = MPI_Allgather(…, rowcomm);
    if( MPI_SUCCESS != rc ) {
        MPI_Comm_revoke(colcomm);
        return;
    }
    rc = MPI_Allgather(…, colcomm);
    if( MPI_SUCCESS != rc ) return;
    compute();
}
```

- If failure is in rowcomm, chances are nobody is dead in colcomm
- A process that raises an exception on rowcomm may not want to participate to the next collective in colcomm
- yet it is not dead, so it has to match its operations, otherwise it is incorrect user code)
- Therefore, this process needs to call Revoke, so that the Allgather doesn't keep waiting on its participation

When one needs Revoke 2/2

```
void one_comm_p2p_transitive_stencil_1d(MPI_Comm comm,…) {
    int rcl=MPI_SUCCESS, rcr=MPI_SUCCESS;
    int leftrank=myrank-1;
    int rightrank=myrank+1;

    for( i=0; i<50; i++ ) {
        if(-1!=leftrank) rcl= MPI_Sendrecv(…, leftrank, …, leftrank,
… ,comm);
        if(np!=rightrank) rcr= MPI_Sendrecv(…, rightrank, …, rightrank,
… ,comm);
        if( MPI_SUCCESS!=rcl || MPI_SUCCESS!=rcr ) {
            MPI_Comm_revoke(comm);
            return;
        }
        dither(); // computation only
    }
```

- If Sendrecv(left=2) fails at rank 3, nobody but rank 1 knows (through sendrecv(right=2))
- A process that raises an exception may not want to continue the dither loop. yet it is not dead, so it has to match its operations, otherwise it is incorrect user code
- This process needs to call Revoke, so that the MPI_Sendrecv(left=3) doesn't keep waiting on its participation

P2P continues across errors

```
void one_comm_p2p_transitive_stencil_1d_norevoke(MPI_Comm comm,…) {
     int rcl=MPI_SUCCESS, rcr=MPI_SUCCESS;
     int leftrank=myrank-1; limax=-1; MPI_Status lstatus;
     int rightrank=myrank+1; rimax=-1; MPI_Status rstatus;

     for( i=0; i<50; i++ ) {
          if(-1!=leftrank && limax<i) //skip over failed iterations
               rcl= MPI_Sendrecv(…, leftrank, sendtag=i, …,
                                        leftrank, MPI_ANY_TAG,
                                        comm, MPI_STATUS_IGNORE);
          if(np!=rightrank && rimax<i)
               rcr= MPI_Sendrecv(…, rightrank, sendtag=i, …,
                                        rightrank, MPI_ANY_TAG,
                                        comm, MPI_STATUS_IGNORE);
          while( MPI_SUCCESS!=rcl ) {
               leftrank--;
               if(-1!=leftrank) {
                    rcl= MPI_Sendrecv(… , leftrank, sendtag=i, …,
                                             leftrank, MPI_ANY_TAG,
                                             comm, &lstatus);
               }
          }
          limax=lstatus.MPI_TAG;
          // (omitted: same stitching for right neighbor)
          dither();
     }
```

- If process on the left fails, stich the chain with next process on the left (some left iteration skipping may happen)
- When a new left neighbor has been found, the normal sendrecv with right will be matched, Communication pattern with right neighbor is unchanged
- Therefore, no need to revoke, dead processes are ignored, algorithm continues on the same comm w/o propagating error condition further

Detecting errors (consistently)

```
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 );
}
```

- Rationale for not standardizing Failures_allget:
  - agreeing on all failures is as expensive as shrinking the comm (computing a new cid while making the failed group agreement is trivial and negligible)
  - Can be written in 4 lines, with the only mild annoyance of having an intermediate comm object to free.
  - Somewhat, to discourage users to use it when unnecessary (it is expensive, making it easy to spot by having a call to a “recovery” function is good)

Spawning replacement ranks 1/2

```
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);
        flagr = flag = MPI_SUCCESS==rc;
        MPI_Comm_agree(shrinked, &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;
        }
```

Spawning replacement ranks 2/2

```
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)
```

Example: in-memory C/R

```
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_Comm_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;
    }
```

Creating Communicators, variant 2

- Communicator creation functions are collective
- Like all other collective, they may succeed or raise ERR_PROC_FAILED differently at different ranks
- Therefore, users need extra caution when using the resultant communicator: is context valid at the target peer?
- This code snippet solves this uncertainty and makes it simple to create comms again.

```
int MPIX_Comm_split_safe(MPI_Comm comm, int
color, int key, MPI_Comm *newcomm) {
    int rc1, rc2;
    int flag;

    rc1 = MPI_Comm_split(comm, color, key,
newcomm);
    rc2 = MPI_Barrier(comm);
    if( MPI_SUCCESS != rc2 ) {
        if(MPI_SUCCESS == rc1 {
            MPI_Comm_revoke(newcomm);
        }
    }
    /* MPI_SUCCESS == rc2 => MPI_SUCCESS == rc1*/
    /* note: when MPI_SUCCES!=rc2, we may revoke
     * a fully working newcomm, but it is safer */
    return rc2;
}
```