ULFM Process Fault Tolerance reading

Aurelien Bouteiller
FT WG

MPI Forum, March 02
Chicago, IL
Info, resources, participate

• Issue Ticket (w/ links to PRs)
  • https://github.com/mpi-forum/mpi-issues/issues/20

• Implementation available
  • Version 1.1 based on Open MPI 1.6 released early November 2015
    https://bitbucket.org/icldistcomp/ulfm
  • Full communicator-based (point-to-point and all flavors of collectives) support
  • Network support IB, uGNI, TCP, SM
  • Runs with ALPS, PBS, etc...
  • RMA, I/O in progress
  • http://fault-tolerance.org/
1. Failure Notification
2. Error Propagation
3. 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.
Errors are visible only for operations that can’t complete

- 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

- Operations that can’t complete return ERR_PROC_FAILED
  - State of MPI objects unchanged (communicators, etc)
  - Repeating the same operation has the same outcome

- Operations that can be completed return MPI_SUCCESS
  - Pt-2-pt operations between non failed ranks can continue

- Leverage on existing error handler infrastructure
  - MPI_COMM_SET_ERRHANDLER
  - conveniently capture and manage the new survivable error codes

Example: only rank4 should report the failure of rank 5
## 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)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return code


ENGELMANN, Christian et NAUGHTON, Thomas. A NETWORK CONTENTION MODEL FOR THE EXTREME-SCALE SIMULATOR.


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...
Resolving transitive dependencies

proc_failed_err_handler(MPI_Comm comm, int err, ...) {
  if(err == MPI_ERR_PROC_FAILED ||
    err == MPI_ERR_REVOKED) {
    if(err == MPI_ERR_PROC_FAILED) 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_Ssend(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
• Some applications are moldable
  • Shrink creates a new communicator on which collectives work

• Some applications are not moldable
  • Spawn can recreate a “same size” communicator
  • It is easy to reorder the ranks according to the original ordering
Scalable Resilient Constructs: Revoke

- BMG* Revoke propagation in less than 100μs
- First post-Revoke collective operation sustains some performance degradation resulting from the network jitter associated with the circulation of revoke tokens
- After the fifth Barrier (approximately 700μs), the Revoke reliable broadcast has completely terminated, therefore leaving the application free from observable jitter.

Scalable Resilient Agreement

- Novel Early Returning Agreement algorithm*
- Logarithmic topology & logarithmic computation: scalable
- 2x the Cray AllReduce latency at 6k processors!

User projects: Resilient X10

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

```
try{ /*Task A*/
    at (p) { /*Task B*/
        finish { at (q) async { /*Task C*/ } }
    }
} catch(dpe:DeadPlaceException){ /*recovery steps*/}
```

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

- 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

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.
User projects: Fenix+S3D

- Fenix is a framework to provide scoped user level checkpoint/restart
  - Provides some of the same services provided by the “MPI_Reinit” idea floated around by T. Gamblin
  - Recover failed processes with revoke-shrink-spawn-reorder sequence
  - Recovered and surviving processes jump back to the start (longjump in Fenix_init)
  - Fenix has helpers to perform user directed “in-memory” or “buddy” checkpointing (and reload)
  - Injection of FT layer: PMPI based

- **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

```fortran
1  allocate(yspc(nx,ny,nz,nslvs))
2  allocate(other_arrays)
3  call MPI_Init()
4  [...] ! Initialize non-conflicting modules
5  call Fenix_Checkpoint_Allocate(C_LOC(yspc),
6     sizeof(yspc),ckpt_yspc)
7  call Fenix_Init(Fenix_Neighbors,PEER_NODE_SIZE,
8     Fenix_resume_to_init, status, C_LOC(world))
9
10 if(status.eq.Fenix_st_survivor) then
11   [...] ! Finalize conflicting modules
12 endif
13 [...] ! Initialize conflicting modules
14 if(status.eq.Fenix_st_new)
15   call initialize_yspc()
16 endif
17
18 do ! Main loop
19   [...] ! Iterate and update yspc array
20   if(mod(step-1,CHECKPOINT_PERIOD).eq.0) then
21     call Fenix_Checkpoint(ckpt_yspc);
22   endif
23 enddo
24
25 call Fenix_Finalize()
26 call MPI_Finalize()
```

User projects: Fenix+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

S3D Code snippet to declare to Fenix the communicators to recover

```
call MPI_Comm_split(gcomm, py+1000*pz, r, xcomm)
call MPI_Comm_split(gcomm, px+1000*pz, r, ycomm)
call MPI_Comm_split(gcomm, px+1000*py, r, zcomm)
call Fenix_Comm_Add(xcomm);
call Fenix_Comm_Add(ycomm);
call Fenix_Comm_Add(zcomm);
[...]
call MPI_Comm_split(gcomm, xid, r, yz_comm)
call MPI_Comm_split(gcomm, yid, r, xz_comm)
call MPI_Comm_split(gcomm, zid, r, xy_comm)
call Fenix_Comm_Add(yz_comm);
call Fenix_Comm_Add(xz_comm);
call Fenix_Comm_Add(xy_comm);
```