FA-MPI Discussion [MPI-3 FT Group]

May 23, 2012

Tony Skjellum, tony@cis.uab.edu
University of Alabama at Birmingham

This work is supported in part by NSF #CCF-1239962

Outline

- Overview of FA-MPI proposal as it stands
- Key ideas
  - What
  - Requirements
  - Features
- Motivation to read the 55-page report ☺
- Status of implementation effort starting
- Will present in more detail in Japan next week, and in July meeting

FA-MPI

- Support fault awareness with low fault free overhead and low jitter
- Expand application space for MPI to faulty environments and to transactional environments (e.g., business space)
- Support hierarchical recovery, fail back and forward
- Offer a complementary [not competitive] technology to the existing MPI-3 proposals – you can have this too, not instead of current FT WG efforts
- Targets only non-blocking APIs as first class targets
- Designed to be used by hand or by aspecting engines such as Rose to allow nominal and fault-aspected MPI programs
- Relies on: ABFT, replay, CPR, and other means for user-driven recovery; FA-MPI provides the MPI piece of the technology only… also includes potential ability to “signal” other transactional technologies (file systems, transactional memories etc) in future, advanced implementations

Main ideas

- Timeout is only new MPI type introduced
- Work with non-blocking MPI-3 APIs
- Two-levels of completion : normal, transactional for requests
- Two-way flow of errors : to/from app and MPI
- Two levels of errors: local (normal MPI) and globalized at transaction point
- Not limited to process failure as the fault model (left as implementation and venue specific, although process and data errors are obvious ones)

Main ideas

- Timeout is only new MPI type introduced
- Work with non-blocking MPI-3 APIs
- Two-levels of completion : normal, transactional for requests
- Two-way flow of errors : to/from app and MPI
- Two levels of errors: local (normal MPI) and globalized at transaction point
- Not limited to process failure as the fault model (left as implementation and venue specific, although process and data errors are obvious ones)

“Fear, Uncertainty, Doubt”

- 55-page proposal means it is complex – no, just long and has examples… ideas straightforward
- Transactions mean this is expensive and not very scalable – no, we have to do synchronizations as with any other data parallel program from time to time, our synchronizations are allgathers on error state under user control
- Doesn’t support blocking functions – can be supported with layered library approach, our audience is non-blocking MPI programs
- etc.

Transaction

- Split-phase collective operation, with fault-awareness (non-blocking or blocking)
- Supports entry and exit conditions from sequences of nominal MPI code (running at no to low fault free overhead)
- Fault-aware allreduce and allgather type operation
- Propagates error across surviving members of communications/windows/files.
- Works with bundles of MPI requests
- Slots for communicators, files, windows in transactional scope for error propagation too
- Fail forward or fail backward, depending on outcome and application strategy
- "Signal other systems of rollbacks" and other components that are transactional in nature or could be made transactional or idempotent in nature to "go back"… this creates the middle-out requirements we talk about in tech report… future optimization for scalability in moderate fault environments.

Requirements Addressed (Briefly)

- A transactional programming model for MPI
- Friendly to 2-sided, DPM, 1-sided, and I/O
- Users set granularity of transactions
- Low added jitter
- No/Low fault-free overhead within transactions
- Minimum impact on scalability (allow apps to tradeoff transaction granularity + recover + etc to minimize time to completion)
- Transactions propagate errors to "surviving group of comm, file, win (s)"
- Mechanism for app to tell MPI which messages to cancel (messages in flight issue)
- Fault injection by app to MPI where needed (observability and testability)
- Minimal new opaque types for MPI-3
- Easy to convert apps/aspect/use
- No multiplicative number of new APIs (additive)
- Support all MPI-3 non-blocking operations [the new probe mechanism and message type are a problem since not using a request … shows the design divergence in this MPI-3 feature to non-request for handling/describing an operation]
- Designed to support hierarchical recovery
- Specify no policies for fault recovery
- Identify operations not directly supportable (e.g., blocking)
- No period of fault freeness assumed between "test and do" (race)
- Communicators/groups etc treated as single-assignment objects
- Enable "fault engineer" to handle different fault models, even those un-modeled by MPI
- Use conventional non-blocking APIs within transaction blocks
- Allow MPI implementations to commit more data or other non-control-flow-related errors if this leads to a better overall fault scenario [can add scalability in Shannon Information Theory Sense]
- Define a minimal number of error conditions
- Enable MPI to say when it cannot continue
- Avoid globalizing state beyond the groups actively involved in the fault detection, isolation, mitigation, recovery
- Leverage new features of MPI-3

TryBlock_start…TryBlock_end

- `TryBlock_start()`
- synchronizing collective operation that admits or rejects the transaction
- Can be non-blocking
- Version for comm, win, file, or just use comm representing group of all objects
- Turns on transaction mode for all requests related to comm, win, file
- Variants to allow for multiple and single communicator-type entities activated in a transaction
- Timeout capability

- `TryBlock_waitall()`
- synchronizing collective operation that reports consistent error state to all surviving participants
- Can also have test variants for non-blocking
- Concludes a transaction with zero or more global-to-group errors
- Isolation, mitigation, recovery can follow the sampling of these global-to-group errors
- Timeout capability

Example, 1 pt 1

- int recovery_mode = 0;
-     do /* this is the soft retry loop. An ABFT fault only was raised,
-         simply redo the loop until we lose confidence in that
-         recovery approach */
-     {
- `   int numerrors = 0;
-         int error_injection = 0;
-         MPI_Timeout timeout;
-         MPI_Timeout_set_ticks(&timeout,1000000); /*specify timeout */
-         /* Ex: 1000000 time periods in units of MPI_Wtick() */
-         /* locally move data into common buffers for Try_Block: */
-         /* now try to enter the TryBlock: */
-         sync_error = MPI_TryBlock_start(comm, MPI_UNSIGNED_INT, local_error_injection, group_error_state, timeout, &req); /* simplest form of TryBlock */
-         /* synchronous error outcome;   a timeout of 0 = infinity */
-         if(sync_error == MPI_SUCCESS) {
-           block_entry = 1;
-           local_error |= MPI_Operation1(comm |
-                     window | file, &req[0]);
-           ABFT_error_logic0(&error_injection);
-             /* user defined logic for their application */
-           local_error |= MPI_Operation1(comm |
-                     window | file, &req[1]);
-           ABFT_error_logic1(&error_injection);
-             /* user defined logic for their application */
-           ...
- local_error |= MPI_Operation1(comm |
-                 window | file, &req[N_requests-1]);
-         ABFT_error_logicN_1(&error_injection);
-           /* user defined logic for their application */

        /* if we find an error in any operations, but
-           local_error is not asserted by any operation,
-           then we use fault injection to indicate the
-           error noted locally; this is a user-defined
-           value (0 is no-error): */
-
-         int try_flag = 0;
-         sync_error = MPI_TryBlock_waitall(req, &try_flag,
-               &try_status,
-               MPI_UNSIGNED_INT, &local_error_injection,
-                group_error_state,
-                 timeout, N_requests, reqs, statuses,
-                 &numerrors, Error_indices);
-          MPI_Request_free(req);
-       }
-       else
-         block_entry = 0;
-

Example, 1 pt 2

```
•        /* Recovery Begins Here   (this is exemplary, not mandatory): */
•                switch(sync_error)
•                {
•                  case MPI_SUCCESS: /* no error */
•                     if(block_entry == 1)
•                     {
•                       /* use completed buffers; note that
•                          increment progress counter,
•                          iterators, etc */
•                     }  else {
•                       /* can't happen, unmodeled behavior */
•                     }
•                     break;
•                  case MPI_LOCAL_ERROR_INJECTION;
•                      /* one or more processes
•                         raised a non-zero error_injection */
•                     soft_retry++;
•                     break;
•                  case MPI_TIMEOUT:
•                     /* can adapt timeout, can do soft retry if 0 or 1 */
•
•        if(block_entry == 0)
•                     {
•                     }
•                     else
•                     {
•                     }
•                     /* alternatively can treat timeout as a hard error,
•                        such as if several backoffs have failed */
•
•                     /* may or may not fall through here... */
•                  case MPI_TASK_FAULT: /* CONST NAMES PRELIM! *.
•                  case MPI_COMMUNICATION_FAULT:
•                  case MPI_UNMODELED_FAULT:
•                  case MPI_MULTIPLE_FAULTS:  /*(see statuses)*/
•                  default:
•                     /* recovery requires repair of a communicator,
•                        and possible backtrack to a checkpoint */
•                     recovery_mode = 1;
•                     break;
•               }

•            } while((soft_retry &&
•                   (soft_retry <soft_retry_max))
•                   && !recovery_mode);
```

MPI_Comm_split_sync

- Workhorse function for mitigation/recovery
- Provides the best set of split groups of surviving entities available to MPI
- Code/keys set by application [looking to allow MPI to set some colors and keys too as generalization]
- Functions with _sync are among the handful of hardened FA-MPI operations designed to work through Faults
- Provides synchronized error state to surviving processes
- Blocking/nonblocking options
- Has timeout capability

MPI_Comm_spawn_and_merge_sync()

- Provides a robust means to spawn and merge to form a new intra communicator
- Surviving group members get notified of what went on with synchronized error info
- Offers to grow size of the net group (at least temporarily) on success

Timeout is only new type

```
•  MPI_Timeout timeout;  /* a timeout opaque object,
•                           measured in units of MPI_Wtick() */
•
•  int MPI_Timeout_set_ticks(MPI_Timeout *timeout, MPI_Aint ticks);
•                              /* set in units of MPI_Wtick() */
•  int MPI_Timeout_set_seconds(MPI_Timeout *timeout, double *usec);
•                              /* set in usec */
•
•  int MPI_Timeout_get_ticks(MPI_Timeout timeout, MPI_Aint &ticks);
•                              /* set in units of MPI_Wtick() */
•  int MPI_Timeout_get_time(MPI_Timeout timeout, double *usec);
•                              /* read in usec */
```

Waiting is at two-levels; local provided to support intra-transaction wait only

```
int MPI_Test_local(MPI_Request *request, int *flag, MPI_Timeout timeout,
                   MPI_Status *status);

int MPI_Testall_local(int count, MPI_Request array_of_requests[], int *flag,
                   MPI_Timeout timeout, MPI_Status array_of_statuses[])

int MPI_Testsome_local(MPI_Timeout timeout, int incount,
                    MPI_Request array_of_requests[],
                    int *outcount, int array_of_indices[],
                    MPI_Status array_of_statuses[])

int MPI_Wait_local(MPI_Timeout timeout, MPI_Request *request,
                   MPI_Status *status);

int MPI_Waitall_local(MPI_Timeout timeout, int count,
                   MPI_Request array_of_requests[],
                   MPI_Status array_of_statuses[]);

int MPI_Waitsome_local(MPI_Timeout timeout, int incount,
                  MPI_Request array_of_requests[], int *outcount,
                  int array_of_indices[], MPI_Status array_of_statuses[]);
```

- **KEY IDEAS: Requests not destroyed, can still be sampled for global error later; timeouts supported**

MPI-2-style 1-sided Support

- Win_fence mode is well understood
- MPI_Put and MPI_Get have to timeout (but we add no explicit timeout) in an FA-MPI implementation
- We are still figuring out how to support Win_Start, Win_Put, Win_Complete
- Still reviewing the MPI-3 improvements to 1-sided and how to support

MPI I/O support

- Collective non-blocking I/O
- We understand how to use TryBlocks with these
- The user program will have to have a way to backup changes in a file since last transaction
- We are just providing the error detection, and allowing the MPI implementation to give errors to the application, and the application to give errors to MPI, and MPI to give info on possible rollback to a transactionally aware file system
- You often may need to rollback the file even without a file error, so this is mainly about communication of error state
- If the filesystem within MPI takes an error, then MPI will signal the application of that at the transaction close. Those errors may cause deeper recovery (less desirable).
- It seems like we need an interface to parallel file systems underneath MPI I/O to describe errors, recovery, mitigation to optimize implementations.
- If single parallel files / file systems cannot continue, currently the MPI application cannot continue. But the app could use a replicated storage approach as its mitigation method with the infrastructure we provide.

Implementation Work

- FA-MPI-Proto1
- Implementation starting
  - OpenMPI, NBC libraries being considered as vectors
- Team has been assembled at UAB
- Design/prototyping starting now
- Header files for API will be among first deliverables to allow dry compiles
- Will report on implementation progress at BOF at SC2012 as well as earlier to MPI Forums

Documentation/Write-ups

- 55-page tech report (lightly updated recently)
  - Has been distributed in February/March
  - Ask me for latest if interested
- Developing conference paper (50%)
  - Seeking appropriate venue (ideas ?)
- API reference to be developed once API finalized for proposing to forum/impl. (20%)
- MPI ticket documentation and chapterware to be developed (.5%)

Conclusions

- FA-MPI Introduces Transactions
- Transactional programming model as affordable as other FT programming models
- Regions of no/low fault-free overhead possible
- Minimal new concepts
- Support for parts of 1-sided and MPI I/O (non-blocking)
- Aspect friendly
- More next week in Japan!