Designing a Reinitializable and Fault Tolerant MPI Library

Nawrin Sultan\textsuperscript{1}  Shane Farmer\textsuperscript{1} Anthony Skjellum\textsuperscript{2}  Ignacio Laguna\textsuperscript{3}  Kathryn Mohror\textsuperscript{3}  Murali Emani\textsuperscript{3}

\textsuperscript{1}Auburn University, Auburn, AL
\textsuperscript{2}University of Tennessee at Chattanooga, TN
\textsuperscript{3}Lawrence Livermore National Laboratory, Livermore, CA

Motivation

- MTBF reduces as HPC systems grow larger with 1000’s to 1M+ processors/cores\textsuperscript{1}
- Checkpoint/Restart is the standard way of surviving failures
- We want Checkpoint/Restart to be faster
- We need to be able to reinitialize MPI rapidly

Objectives

-Checkpoint the state of the MPI library
  - Remove application state dependency
  - Allow applications to tell MPI to checkpoint itself
  - Reinitialize the state of the MPI library from a previously taken checkpoint
  - Running MPI job will continue without being suspended from the job queue
  - Define portable checkpoint

Implementation/Architecture

-\#include <mpi.h>

```c
int main(int argc, char **argv) {
    // variable initialization
    while (!abort && !done) {
        switch(code) {
            case MPI_SUCCESS:
                if (just_check) {
                    // code indicating reinitialization
                    // Reinitialize MPI state or stores fresh as needed */
                    code = MPIX_New_reinitialize( . . . );
                    MPIX_get_fault( . . . , &fault_epoch);
                }
                break;
            case MPIX_SUCCESS_RECOVERY:
                if (just_check) {
                    // Reloading MPI checkpoint
                    code = MPIX_Reinitialize( . . . );
                    MPIX_checkpoint_write( . . . , &checkpoint);
                }
                break;
            case MPIX_SUCCESS_RESTART:
                if (just_check) {
                    // Process restarted from scratch
                    code = App_Checkpoint_Send( . . . , &restart_iteration);
                    MPIX_checkpoint_write( . . . , &checkpoint);
                }
                break;
            default:
                if (just_check) {
                    // Process restarted from scratch
                    code = App_Checkpoint_Read( . . . , &restart_iteration);
                    MPIX_checkpoint_write( . . . , &checkpoint);
                }
                break;
        }
    }
    MPI_Finalize();
    return 0;
}
```

Recovering From Failure

- Step1: Detect the failure of process K;
- Step2: Notify all other processes through asynchronous notification;
- Step3: Spawn a new process on the node where rank K ran on;
- Step4: Spawning entity loads CP0 from checkpoint;
- Step5: Notify that the newly spawned process of the modified table made from CP0; it is now in a consistent state;
- Step6: We notify all the other ranks of the modified table by sending just the new process tuple;
- Step7: A new checkpoint of the modified process table is made; Call it CP1;
- Step8: The MPI_COMM_WORLD and group underneath is restored to a consistent state; computation can continue

References


Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC (LLNL-PRST-741063) and partial support from NSF under grants 1360199 and 1323932 acknowledged. The ideas expressed are those of the authors and not necessarily LLNL or NSF.