AUBURN
UNIVERSITY

MPI Stages

Nawrin Sultana
Auburn University

Anthony Skjellum
University of Tennessee at Chattanooga

Ignacio Laguna Kathryn Mohror Murali Emani
Lawrence Livermore National Laboratory

THE UNIVERSITY OF TENNESSEE CHATTANOOGA

Lawrence Livermore National Laboratory

Key Concepts

- Checkpoint the state of MPI library and application
- Synchronous checkpoints
- Minimize recovery time
  - Eliminate job restart time - Reinit
  - Removes the requirement of reinitialization all processes after failure
- Portability across implementations
- Incorporate FT support with low programming complexity

[1] Designing a Reinitializable and Fault Tolerant MPI Library [Poster-EuroMPI/USA'17]
[2] Checkpointable MPI: A Transparent, Fault-Tolerance Approach for MPI [Short paper- ExaMPI'17]

Overview

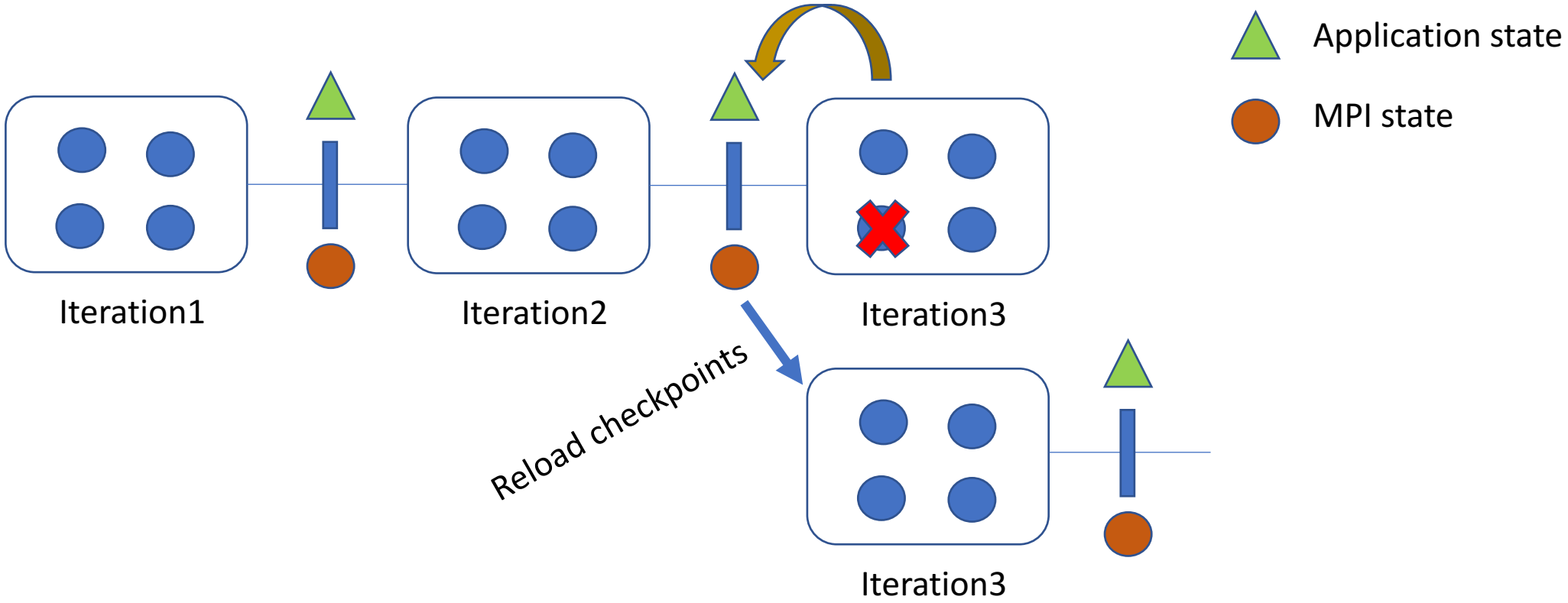

Figure 1: Both MPI state and application state are checkpointed after each iteration. After a failure, both states are restored by reloading the most recent checkpoints.

Handling Failure

- Using ‘epoch’ to distinguish between a new and restarted process
- Live processes will be notified after a failure
  - Discard the current work and return error code (MPIX_TRY_RELOAD)
  - Application check the return code and call MPIX_Load_checkpoint
  - Wait on implicit barrier
- Failed process (Relaunch with epoch > 0)
  - Load MPI checkpoint from MPI_Init
  - Wait on implicit barrier
- After barrier completion all processes load the application checkpoint and continue the execution
- All processes load the last synchronous checkpoint

Pseudocode

```
#include <mpi.h>
int main(int argc, char **argv) {
  // variable initialization
  while (!abort && !done) {
       switch(code) {
        case MPI_SUCCESS:  // new and restarted processes
          code = MPI_Init(&argc, &argv);
           MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
          break;

        case MPIX_TRY_RELOAD: // Live processes to load MPI checkpoint
          code = MPIX_Load_checkpoint(. . .);
          break;
        . . .
       }

       MPIX_Get_fault_epoch(&fault_epoch);
       code = main_loop(. . .);
   }
  MPI_Finalize();
  return 0;
}
```

```
int main_loop(int fault_epoch, int *done) {
     if (fault_epoch > 0)
          Application_Checkpoint_read();
     else
         // Initialize application state

     for (int i = iteration; i < MAX_ITERATIONS; ++i) {
           . . .
           int code = MPI_Reduce(. . .);
           if (code == MPIX_TRY_RELOAD)
               return code;
           . . .
           Application_Checkpoint_Write(. . .);
           MPIX_Checkpoint_write(. . .);
     }

     if (code == MPI_SUCCESS)
          *done = 1;
     return MPI_SUCCESS;
}
```

Challenges

- Defining the critical MPI state to checkpoint
- Minimize Checkpoint/restart overhead
  - Identify least number of MPI object to checkpoint

Current Status

- Implementing a prototype using own mini MPI as a proof-of-concept
- Currently only supports MPI_COMM_WORLD
- Add collective calls and multiple communicators
- Test on LULESH

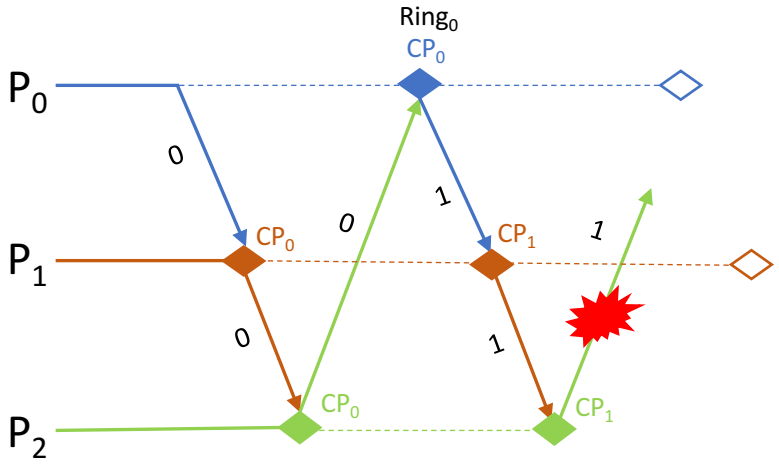

Figure 2: P_2 fails after CP_1; CP0 will be loaded and the ring will start by sending 1

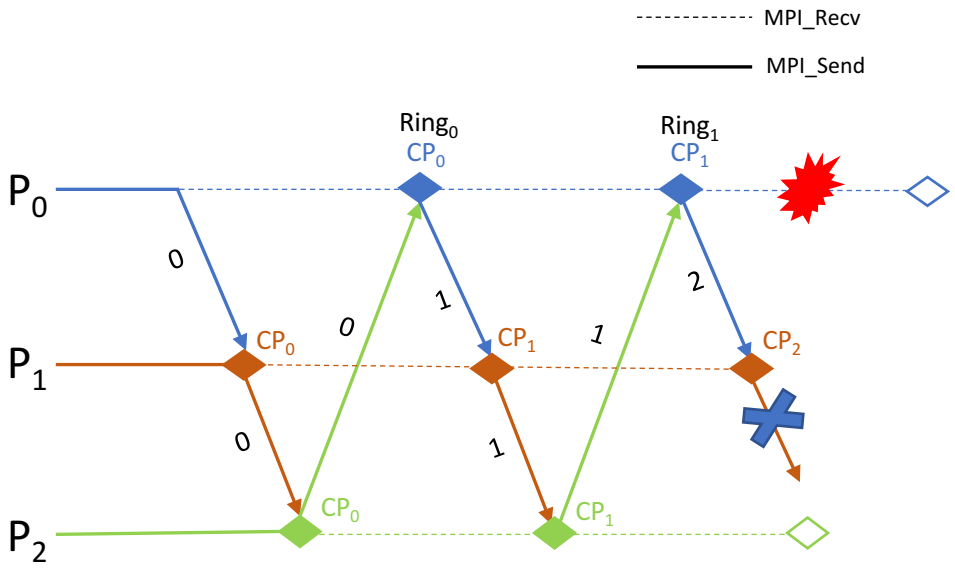

Figure 3: Process P_0 fails after completion of second Ring