THE UNIVERSITY OF TENNESSEE CHATTANOOGA

SIMCENTER

Center of Excellence in Applied Computational Science and Engineering

Serializing and Deserializing MPI Opaque Objects with MPI Stages

Derek Schafer (UTC), Ignacio Laguna (LLNL), Kathryn Morhor (LLNL)
Anthony Skjellum—presenter

EuroMPI 2020 [Poster]
September 23, 2020

Serializing and Deserializing MPI Opaque Objects with MPI Stages

Derek Schafer, Ignacio Laguna, Anthony Skjellum, Kathryn Mohror

Lawrence Livermore National Laboratory

SimCenter Center of Excellence in Applied Computational Science & Engineering

The University of Tennessee at Chattanooga

Introduction

- MPI Stages is a generalized form of the Reinit model of fault tolerance
- MPI Stages saves internal MPI state in a separate checkpoint from the application state saved by the application. Upon failure, both MPI and application state can be recovered
- This work expands the Stages model to enable serialization of MPI Handles

MPI Stages API

- Previous Stages API Functions
  1. MPIX_Checkpoint_write
  2. MPIX_Checkpoint_read
  3. MPIX_Get_fault_epoch
  4. MPIX_Serialize_handler_register
  5. MPIX_Serialize_handles
  6. MPIX_Deserialize_handler_register
  7. MPIX_Deserialize_handles
- New Stages API Functions
  8. MPIX_Deallocate_stale_resources
  9. MPIX_FT_errno

Ongoing Efforts

- Integrating MPI Stages into a production application, such as the seismic wave propagation modelling library from LLNL, SW4).
- Expand serialization support for more than just MPI Groups, MPI Communicators and MPI Datatypes
- Finer-grained handle de/serialization
- Performance optimizations of the Stages model
- Identifying key atomic fault-tolerance actions

We **invented** MPI opaque object **serialization** for fault tolerance.

Implementation Details

- User provides handles to MPI objects to be saved during checkpoint
- Serializes MPI objects in JSON format
  - Groups save their ID and list of processes
  - Communicators save ID, Group ID, Context ID, and topology info
  - Datatypes save input parameters (and dependent type information)
  - Dependent MPI objects may also be saved even if not explicitly requested
- Not every object created by an MPI application need to be saved
- FIFO ordering of serialization callback functions
- No extra save state required in user checkpoint
- No synchronization needed upon restoring
- Assumes application has no messages in flight or active requests

```
typedef struct
{
    MPI_Comm *comms;
    int comm_size;
    MPI_Group *grps;
    int group_size;
    MPI_Datatype * types;
    int datatype_size;
} MPIX_Handles;
```

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-POST-813591).

Resilient Loop Example

```
#include <mpi.h>
int main(int argc, char **argv){
    int code = MPI_SUCCESS, done = 0;
    /* MPI Initialization */
    MPI_Init(&argc, &argv);
    /* Set error handler */
    MPI_Comm_set_errhandler(...);
    /* Register function for serialization */
    MPIX_Serialize_handler_register(s_function);.
    /* Register funciton for deserialization */
    MPIX_Deserialize_handler_register(d_function);
    while(0 == done){ /* Resilient loop */
        if(MPIX_TRY_RELOAD == code) {
            /* Deallocate old resources */
            MPIX_Deallocate_stale_resource(s_function);
            /* Restore MPI state from checkpoint */
            MPIX_Checkpoint_read();
        }
        else if(MPI_SUCCESS != code)
            MPI_Abort(...);
        /* Figure out current epoch */
        MPIX_Get_fault_epoch(&fault_epoch);
        /* Main simulation loop */
        code = main_loop(argc, argv, fault_epoch, &done);
    }
    /* Free MPI Resources */
    MPI_Comm/Group/Type_free(...);
    MPI_Finalize();
    return 0;
}
```

Application Loop Example

```
int main_loop(int argc, char **argv, int epoch, int *done) {
    /* Initialize or restore application state */
    if (epoch > 0){
        Application_Checkpoint_Read(...);
        /* Populate handles from last checkpoint*/
        MPIX_Deserialize_handles();
    }
    else
        /* Initialize application state */
    /* Main simulation loop */
    while(iteration < MAX_ITERATION){
        /* MPI Communication */
        ...
        code = MPI_Allreduce(...);
        if(MPI_SUCCESS != code)
            return MPIX_TRY_RELOAD;
        ...
        /* Save application state */
        Application_Checkpoint_Write(...);
        /* Save Opaque MPI Handles */
        MPIX_Serialize_handles();
        /* Save MPI State */
        MPIX_Checkpoint_write();
    }
    /* Barrier to ensure no processes call */
    /* finalize until simulation is complete */
    code = MPI_Barrier(...);
    if (code == MPI_SUCCESS)
        *done = 1; /* Successful Completion */
    return code;
}
```

Outline

- Background on MPI Stages
- Why we need to serialize/deserialize MPI opaque objects + example APIs
- What the resilient loop looks like
- What the application loop looks like
- Summary

Why Serialize/Deserialize

- The stages model effects either of these two things to every MPI process after a detected fault
  - process recovery/restart OR
  - process “return to consistent state”
- Opaque objects held by the user must be reassociated with user code and reconnected with MPI!
- Recovery has an aspect of reconnection at deserialization
- This is a cooperative model of serialization/deserialization and CPR between the application, MPI+Stages extensions

Background – MPI Stages

- Generalized form of the Reinit model of fault tolerant MPI
- Key idea: checkpoints MPI and application state at synchronous checkpoints under user control
- To support more general MPI programs, we must be able to serialize/deserialize handles
- This model is working effectively on more and more complex MPI codes over time
- Utilizes our own MPI implementation-ExaMPI- that is designed to support MPI-state
- Fault models is currently process faults (like ULFM and others); network and other faults possible also in future.

MPI Stages API

- Previous Stages API Functions
  1. MPIX_Checkpoint_write
  2. MPIX_Checkpoint_read
  3. MPIX_Get_fault_epoch
  4. MPIX_Serialize_handler_register
  5. MPIX_Serialize_handles
  6. MPIX_Deserialize_handler_register
  7. MPIX_Deserialize_handles
- New Stages API Functions
  8. MPIX_Deallocate_stale_resources
  9. MPIX_FT_errno

Resilient Loop Example

```c
#include <mpi.h>
int main(int argc, char **argv){
    int code = MPI_SUCCESS, done = 0;
    /* MPI Initialization */
    MPI_Init(&argc, &argv);
    /* Set error handler */
    MPI_Comm_set_errhandler(...);
    /* Register function for serialization */
    MPIX_Serialize_handler_register(s_function);.
    /* Register funciton for deserialization */
    MPIX_Deserialize_handler_register(d_function);
    while(0 == done){ /* Resilient loop */
        if(MPIX_TRY_RELOAD == code) {
            /* Deallocate old resources */
            MPIX_Deallocate_stale_resource(s_function);
            /* Restore MPI state from checkpoint */
            MPIX_Checkpoint_read();
        }
        else if(MPI_SUCCESS != code)
            MPI_Abort(...);
        /* Figure out current epoch */
        MPIX_Get_fault_epoch(&fault_epoch);
        /* Main simulation loop */
        code = main_loop(argc, argv, fault_epoch, &done);
    }
    /* Free MPI Resources */
    MPI_Comm/Group/Type_free(...);
    MPI_Finalize();
    return 0;
}
```

Application Loop Example

```c
int main_loop(int argc, char **argv, int epoch, int *done) {
    /* Initialize or restore application state */
    if (epoch > 0){
        Application_Checkpoint_Read(...);
        /* Populate handles from last checkpoint*/
        MPIX_Deserialize_handles();
    }
    else
        /* Initialize application state */
    /* Main simulation loop */
    while(iteration < MAX_ITERATION){
        /* MPI Communication */
        ...
        code = MPI_Allreduce(...);
        if(MPI_SUCCESS != code)
            return MPIX_TRY_RELOAD;
        ...
        /* Save application state */
        Application_Checkpoint_Write(...);
        /* Save Opaque MPI Handles */
        MPIX_Serialize_handles();
        /* Save MPI State */
        MPIX_Checkpoint_write();
    }
    /* Barrier to ensure no processes call */
    /* finalize until simulation is complete */
    code = MPI_Barrier(...);
    if (code == MPI_SUCCESS)
        *done = 1; /* Successful Completion */
    return code;
}
```

Summary/Conclusions

- MPI Stages is an effective model of MPI Fault Tolerance
- The type of serialization and deserialization is useful here and perhaps for other models
- The resilient loop model can also, potentially, be useful as common aspect with other MPI F. T. models too
- This approach works in practice with acceptable overheads
- Future support for I/O, window, and persistent collective handles planned.

Acknowledgements

- Ignacio Laguna, Kathryn Morhor, LLNL
- Derek Schafer, Anthony Skjellum

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract

DE-AC52-07NA27344 (LLNL-POST-813591).

- Contacts:
  - lagunaperalt1@llnl.gov
  - mohror1@llnl.gov
  - Derek-schafer@utc.edu
  - Tony-skjellum@utc.edu