MPI Global-Restart Fault Tolerance Specification

Version 0.1.0

Unofficial, for comment only

Ignacio Laguna and Giorgis Georgakoudis
ilaguna@llnl.gov, georgakoudis1@llnl.gov

Lawrence Livermore National Laboratory

November 9, 2020

Chapter 1

Global-Restart Fault Tolerance

1.1 Introduction

The traditional method to handle process failures in large-scale scientific applications is periodic, global synchronous checkpoint/restart (CPR). When a process failure occurs in a bulk synchronous MPI program, it quickly propagates to other processes so re-starting the application from a previously-saved checkpoint is a simple solution to recover from failures.

A large number of MPI applications already use some form of global synchronous CPR. The goal of global-restart fault tolerance is to provide an easy-to-use interface to improve the efficiency of CPR in bulk synchronous applications by reducing as much as possible the recovery time when failure occurs.

In this chapter, we refer to the global-restart fault tolerance model and interface as the **Reinit** (i.e., re-initialization) model and interface, respectively.

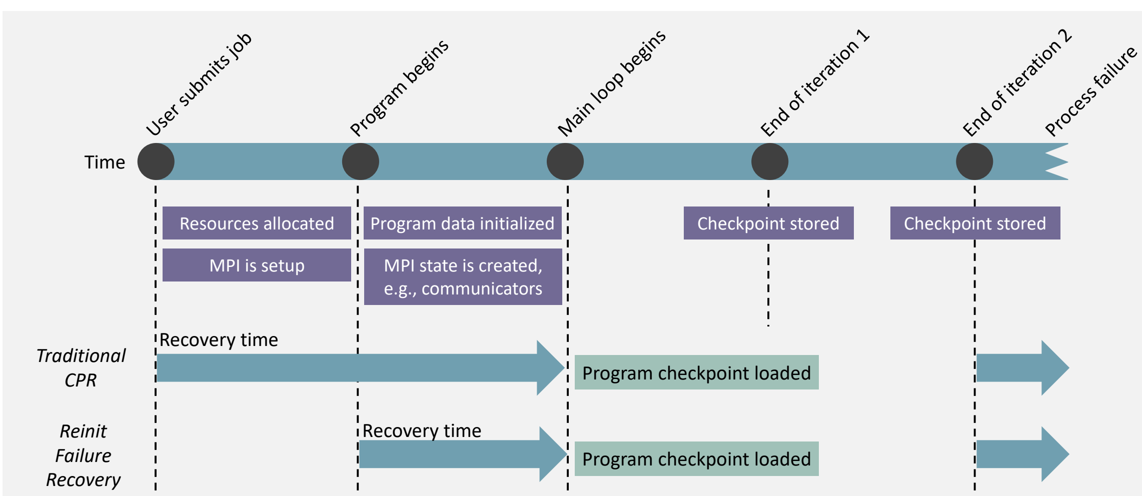

Figure 1.1: The global-restart fault tolerance model (Reinit) provides a mechanisms to reduce the recovery time for bulk synchronous applications that use periodic synchronous checkpoint/restart.

1.2 Fault Model

The Reinit model provides a pre-defined fault-tolerance mechanism to survive **MPI process failures**. We use the definition of process failures used in Section 2.8, i.e., a process failure occurs when an MPI process unexpectedly and permanently stops communicating (e.g., a software or hardware crash results in an MPI process terminating unexpectedly). In the rest of the chapter, when we refer to *failures* we mean *MPI process failures*.

1.3 Reinit MPI Interface

The Reinit interface for global-restart fault tolerance is composed of two MPI functions: `MPI_REINIT` and `MPI_TEST_FAILURE`. This section describes the syntax of these MPI functions.

MPI_Reinit

`int MPI_Reinit(resilient_fn, void *data)`

IN	resilient_fn	user defined procedure (function)
IN	data	pointer to user defined data

The user-defined procedure should be in C, a function of type MPI_Reinit_function which is defined as:

```
typedef MPI_Reinit_fn void (*)(void *data));
```

The first argument is a user defined procedure, `resilient_fn`, which is called by the MPI_Reinit procedure. The second argument is a pointer to user defined data. This pointer is passed as an argument to the user defined procedure, `resilient_fn`, when the procedure is called. A valid MPI program must contain at most one call to the MPI_Reinit procedure. Calling MPI_Reinit more than one time results in undefined behavior.

The purpose of the user defined `resilient_fn` procedure is to specify a *rollback location*, i.e., a program location to resume execution after a process failure occurs. Depending on the error handler being used, upon the detection of a process failure, MPI will cause the execution of the program to resume at the `resilient_fn` procedure synchronously or asynchronously (see the Error Handling section for more details).

After the `resilient_fn` procedure is re-executed due to failure recovery, the only valid communication objects are the communicators MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_NULL.

> *Advice to users.* MPI objects that are created before MPI_Reinit is called will not be valid when the `resilient_fn` procedure is re-executed due to a failure. (*End of advice to users.*)

Calling the MPI_Reinit procedure sets the `resilient_fn` procedure to be a rollback location and makes this rollback location active. After activating the rollback location, MPI_Reinit calls the `resilient_fn` procedure. After the MPI_Reinit procedure returns, the rollback location becomes inactive. If a failure occurs during an inactive rollback location, MPI cannot resume execution at the rollback location, and as a result cannot recover from failures using the Reinit model.

> *Advice to users.* To able to survive most of the process failures that can occur during the execution of the program, most calls to MPI and computation should be executed before MPI_Reinit returns. (*End of advice to users.*)

An MPI process must invoke MPI_FINALIZE only after MPI_Reinit returns.

MPI_Test_failure

```
int MPI_Test_failure()
```

The MPI_Test_failure procedure causes the program to resume execution at the rollback point that was activated by MPI_Reinit when two conditions occur: (1) the MPI_ERRORS_REINIT_SYNC handler is associated with MPI_COMM_WORLD, and (2) a failure has been detected before MPI_Test_failure is called.

If no failures were detected before MPI_Test_failure is called, the return code value is MPI_SUCCESS and the procedure performs no operations. If on the other hand failures are detected before the procedure is called, the procedure does not return and it immediately resumes execution at the rollback point.

1.4 Error Handling

MPI provides two predefined error handlers that can be used to handle failures using the Reinit model. Unlike other predefined error handlers, such as MPI_ERRORS_ARE_FATAL, that can be associated to communicator, window, file, and session objects, the Reinit error handlers are by default associated to the predefined MPI_COMM_WORLD communicator. Associating the Reinit error handlers to window, file, session objects, or communicators other than MPI_COMM_WORLD is undefined.

> *Rationale.* Associating the Reinit error handler to MPI_COMM_SELF would have no effect if a failure occurs because the process that contains MPI_COMM_SELF failed and the error handler cannot be called. Since a process failure during the handling of MPI objects, such as windows, files and sessions eventually manifest itself as a process failure in MPI_COMM_WORLD, it makes sense to associate a Reinit error handler to MPI_COMM_WORLD only. (*End of rationale.*)

The following Reinit error handlers are available in MPI:

- **MPI_ERRORS_REINIT_ASYNC**: The handler is called by MPI immediately after a process failure is detected. The handler, when called, causes the execution of the program to resume at (or jump back to) the active rollback location that was activated by MPI_Reinit.
- **MPI_ERRORS_REINIT_SYNC**: The handler has two effects. The first effect is that it enables the MPI_Test_failure function to cause the execution of the program to resume at (or jump back to) the active rollback location. The second effect is that it returns the error code to the user.

Using the MPI_ERRORS_REINIT_ASYNC handler causes MPI to resume execution of the program when an error is detected whether or not the error is detected during a call to MPI. On the other hand, using the MPI_ERRORS_REINIT_SYNC handler causes MPI to resume execution only after MPI_Test_failure function is called if an error was detected.

1.5 Examples

Example 1.1 Using Reinit with asynchronous error handling to recover from process failures

```
typedef struct {
    int argc;
    char **argv;
} data_t;

void resilient_function(void *arg)
{
    data_t *data = (data_t *)arg;
    // Cleanup library, if needed
    cleanup_library_state();
    // Resume computation from checkpoint
    // or initialize application data
    if( load_checkpoint() )
        printf("Resume from checkpoint\n");
    else
        init_app_data(data->argc, data->argv);
    bool done = false;
    while(!done) {
        done = compute();
        store_checkpoint();
    }
}

int main(int argc, char *argv[])
{
    // Initialize user defined data type
    data_t data = { argc, argv };

    MPI_Init(argc, argv);
    MPI_Comm_set_errhandler(MPI_WORLD_COMM, MPI_ERRORS_REINIT_ASYNC);
    // MPI_Reinit sets the rollback location
    // to resilient_function and calls it.
    // In asynchronous error handling, the program
    // will go to the rollback location as soon a
    // failure is detected
    MPI_Reinit(&data, resilient_function);
    MPI_Finalize();
```

```
    return 0;
}
```

Example 1.2 Using Reinit with synchronous error handling to recover from process failures

```
void resilient_function(void *arg)
{
    data_t *data = (data_t *)arg;
    // Cleanup library, if needed
    cleanup_library_state();
    // Resume computation from checkpoint
    // or initialize application data
    if( load_checkpoint() )
        printf("Resume from checkpoint\n");
    else
        init_app_data(data->argc, data->argv);
    bool done = false;
    while(!done) {
        done = compute();
        store_checkpoint();
        // Calling MPI_Test_failure will go to the
        // rollback location, that is resilient_function,
        // in case of a failure
        MPI_Test_failure();
        // MPI + computation
        MPI_Test_failure();
        // MPI + computation
        MPI_Test_failure();
    }
}
```

1.6 To-Do List

1. Define FORTRAN bindings
2. Define what happens with MPI state in tools (e.g., PMPI tools).