Chapter 12

Coarse-Grained Fault Tolerance (Reinit)

12.1 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, the failure propagates to other processes so restarting the application from a previously-saved checkpoint is a simple and effective solution to recover from failures.

A large number of MPI applications already use some form of synchronous CPR. The goal of *coarse-grained fault tolerance* is to provide an easy-to-use interface to improve the efficiency of CPR in bulk synchronous applications by reducing the recovery time when failure occurs and making recovery in MPI more automatic.

In this chapter, we refer to the coarse-grained fault tolerance model and interface as the *Reinit* (i.e., reinitialization) model and interface, respectively.

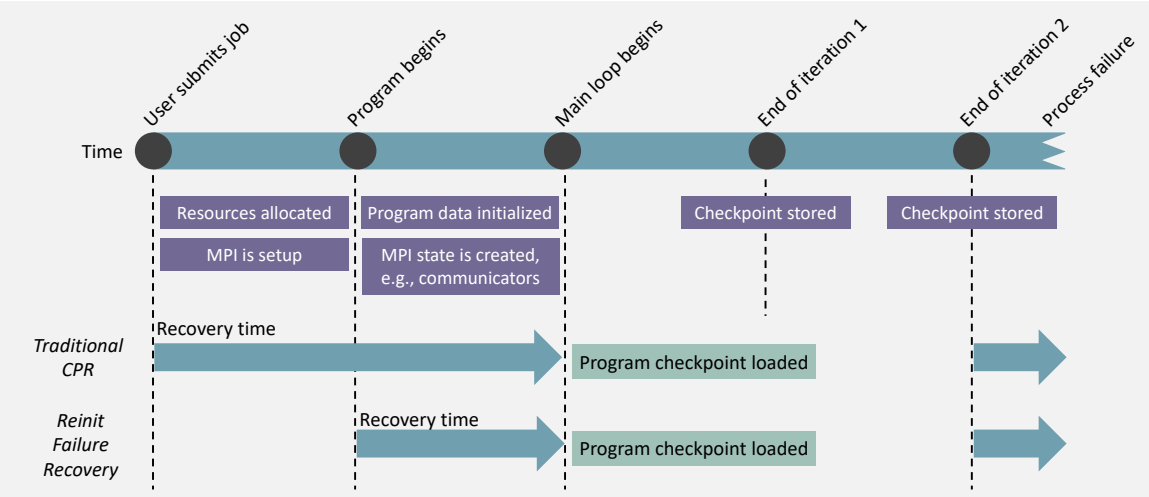

Figure 12.1: The coarse-grained fault tolerance model, Reinit, provides a mechanism to reduce the recovery time for bulk synchronous applications that use periodic synchronous checkpoint/restart.

12.2 Fault Model

The Reinit model provides a predefined fault-tolerance mechanism to survive MPI process failures. The definition of a process failure is in Section X.Y. The Reinit model assumes that

the application is responsible for recovering data that may be lost due to process failures. The application can use different mechanisms to recover its data, for example, reloading a checkpoint that was saved before the failure occurred or regenerating the data.

12.3 Reinit MPI Interface

The Reinit interface is composed of two MPI functions: MPI_REINIT and MPI_TEST_FAILURE.

MPI_REINIT(resilient_fn, data)

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

C binding

```
int MPI_Reinit(MPI_Reinit_fn resilient_fn, void *data)
```

The user-defined function resilient_fn should be in C and type `MPI_Reinit_fn` which is defined as: `typedef MPI_Reinit_fn void (*)(void *data));`

The first argument is a user defined function, `resilient_fn`, which is called by MPI_REINIT to recover from failures. The second argument is a pointer to user-defined data. This pointer is passed as an argument to the user-defined function, resilient_fn, when the function is called. A valid MPI program must contain at most one call to MPI_REINIT. Calling MPI_REINIT more than one time results in undefined behavior. MPI_REINIT should be called only after MPI has been initialized with the World Model. It is valid to use the Session Model as long as MPI_REINIT is called after the World Model is used for initialization.

The purpose of resilient_fn 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 function automatically or nonautomatically (see the Error Handling section for more details).

12.3.1 MPI State Cleanup

In the Reinit model, MPI automatically cleans up its state during failure recovery. The goal of the Reinit failure recovery is to bring back to MPI a state as a similar as possible to the state after MPI_INIT / MPI_INIT_THREAD returns.

After resilient_fn is reexecuted due to failure recovery, all MPI state associated with the World Model and Session Model is cleaned up. This means that MPI objects and handles created in the World and Session models become invalid; the only valid communication objects are the communicators MPI_COMM_WORLD, MPI_COMM_SELF, and the predefined constant MPI_COMM_NULL. Also, all objects created by MPI calls are automatically freed.

With respect to tools and dynamic processes, after resilient_fn is reexecuted, sessions created from the MPI tool information interface are invalid, and processes started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE become invalid (dangling). A high quality implementation should destroy dangling processes after MPI recovered from the failure.

Advice to users. MPI objects that are created before MPI_REINIT is called will not be valid after the resilient_fn function is reexecuted due to a failure. (*End of advice to users.*)

Calling MPI_REINIT sets the resilient_fn function to be a rollback location and makes this rollback location active. After activating the rollback location, MPI_REINIT calls the resilient_fn procedure. After MPI_REINIT 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 survive most of the process failures that can occur during the execution of the program, users are advised to execute most calls to MPI and computation **before** MPI_REINIT returns. (*End of advice to users.*)

An MPI process must invoke MPI_FINALIZE only after MPI_REINIT returns.

Advice to implementors. Implementors can freely choose how to implement the specification of a rollback location and resuming the execution of the program at that location when a failure occurs. A practical way to implement such functionality in C and C++ is by using the setjmp and longjmp interface. The execution context to specify a rollback location can be saved using setjmp. When a failure occurs, longjmp can transfer control to the call site of setjmp, effectively resuming execution at that location. Another way to implement such functionality is using C++ try and catch exception handling. (*End of advice to implementors.*)

12.3.2 Checking for Failures

MPI_TEST_FAILURE()

IN	void

C binding

```
int MPI_Test_failure(void)
```

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_NONAUTO 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.

12.4 Error Handling

MPI provides two predefined error handlers that can be used to handle failures using the Reinit model. While these error handlers are intended to be used primarily to handle failures

when the World Model is used to initialize MPI, it is allowed to use the Session Model and the World Model concurrently to handle failures with 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 must be associated only to the predefined MPI_COMM_WORLD communicator in the World Model. Associating the Reinit error handlers to window, file, session objects, or communicators other than MPI_COMM_WORLD is undefined.

> *Rationale.* Associating a Reinit error handler to MPI_COMM_SELF would have no effect—MPI_COMM_SELF includes only the process itself and the goal of the Reinit model is that all processes participate in failure recovery. 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, associating a Reinit error handler to MPI_COMM_WORLD will eventually allow handling failures that affect other MPI objects. (*End of rationale.*)

The following Reinit error handlers are available in MPI:

- **MPI_ERRORS_REINIT_AUTO**: 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_NONAUTO**: 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 when MPI_TEST_FAILURE is called. The second effect is that it returns the error code to the user.

Using the MPI_ERRORS_REINIT_AUTO 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_NONAUTO handler causes MPI to resume execution only after the MPI_TEST_FAILURE function is called if an error was detected.

12.4.1 Association of Error Handlers

The Reinit error handlers must be associated to MPI_COMM_WORLD before the MPI_REINIT procedure is called. Calling MPI_REINIT before associating any of the Reinit error handlers produces undefined behavior.

After a Reinit error handler has been associated to MPI_COMM_WORLD, it is invalid to associate a different Reinit error handler to MPI_COMM_WORLD.

12.4.2 Behavior for Specific Error Conditions

If an error occurs and one of the Reinit error handlers has been set but there is no active Reinit rollback location, MPI will behave as if the MPI_ERRORS_ARE_FATAL error handler is set (see Figure 12.2).

Errors can occur between the moment the MPI_ERRORS_REINIT_NONAUTO handler is set and the MPI_TEST_FAILURE function is called—if an error occurs in such period of time, MPI behaves as if the MPI_ERRORS_RETURN handler is set.

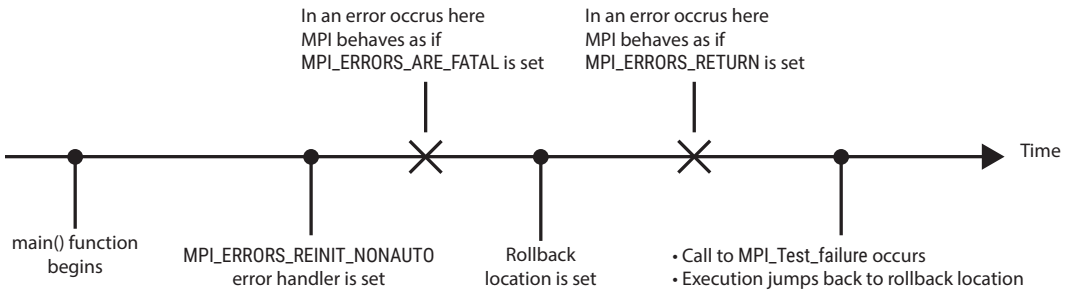

Figure 12.2: Different error scenarios for the MPI_ERRORS_REINIT_NONAUTO error handler.

12.5 Tools

The Reinit interface supports the use of MPI tools. The following must be taken into consideration when writing MPI tools.

The Reinit interface assumes that, when a process failure occurs, data may be lost. If a tool requires data that can be lost due to failures, the tool must implement a mechanism to recover such data, for example, reloading a checkpoint.

An MPI implementation should provide a performance variable of type MPI_T_PVAR_CLASS_COUNTER named MPI_T_FAILURES that reflects the number of times the MPI process has been reinitialized due to failures. The variable has a value of zero initially and it is incremented every time the program resumes execution at the rollback location.

The performance variables that are provided by an MPI implementation are not reset when execution resumes at the rollback location. Tools are responsible for presenting information about performance variables to users after taking into account failures.

12.5.1 Reinit callbacks

A tool may need to clean up its state when a failure occurs. To do so, the MPI tool Information Interface can export events associated to the change of control when the program is resumed from the rollback location. The index of the events can be obtained by the tool by the function MPI_T_EVENT_GET_INDEX. There are two event types:

- **Before cleanup:** This event occurs before MPI resumes execution at the rollback location, before the state of MPI is cleanup, and before `resilient_fn` function is called. The name of the event type must be the string "BEFORE_REINIT".
- **After cleanup:** This event occurs after MPI resumes execution at the rollback location, after the state of MPI is cleanup, but before `resilient_fn` function is called. The name of the event type must be the string "AFTER_REINIT".

The callback routines associated to these event types must have the highest safety requirement level, i.e., MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE, since the MPI implementation may invoke the callbacks from a signal handler. The event types may be bound to the MPI_T_BIND_NO_OBJECT.

> *Advice to users.* If a tool desires to get notifications about the Reinit events, the tool should register the callbacks as soon as possible, preferably in the initialization

of MPI. If a failure occurs and the execution is being resumed before callbacks have been associated, the tool may not be notified of the events. (*End of advice to users.*)

12.6 Examples

We present a few examples of how to use the Reinit interface with synchronous and asynchronous error handlers.

Example 12.1. Using automatic error handling to recover from failures.

```
#include "mpi.h"

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_COMM_WORLD, MPI_ERRORS_REINIT_AUTO);
  // MPI_Reinit sets the rollback location
  // to resilient_function and calls it.
  // In automatic 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 12.2. Using non-automatic error handling to recover from failures.

```
#include "mpi.h"

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();
    MPI_Test_failure();
    store_checkpoint();

    // MPI + computation
    compute();
    // Calling MPI_Test_failure will resume execution at the
    // rollback location, that is the resilient_function,
    // in case of a failure.
    MPI_Test_failure();

    // MPI + computation
    compute();
    MPI_Test_failure();
  }
}
```