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 quickly 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 global 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 as much as possible the recovery time when failure occurs and making the recovery as automatic as possible.

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

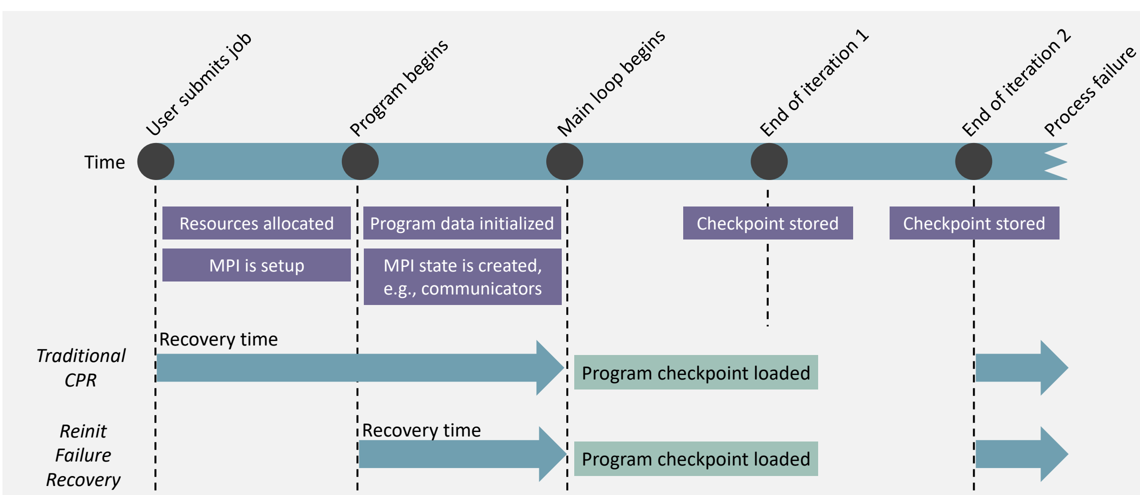

Figure 12.1: The coarse-grained fault tolerance model (Reinit) provides a mechanism to reduce the recovery time for bulk synchronous applications that use periodic synchronous checkpoint/restart.

12.2 Fault Model

The Reinit model provides a predefined fault-tolerance mechanism to survive MPI process failures. 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*.

The Reinit model assumes that the application's data will be recovered after a failure. 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).

After resilient_fn is re-executed due to failure recovery, the only valid communication objects are the communicators MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_NULL. If the Session Model is in use, the only valid process set names after resilient_fn is re-executed are "mpi://WORLD" and "mpi://SELF".

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 be 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.

12.3.1 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 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.

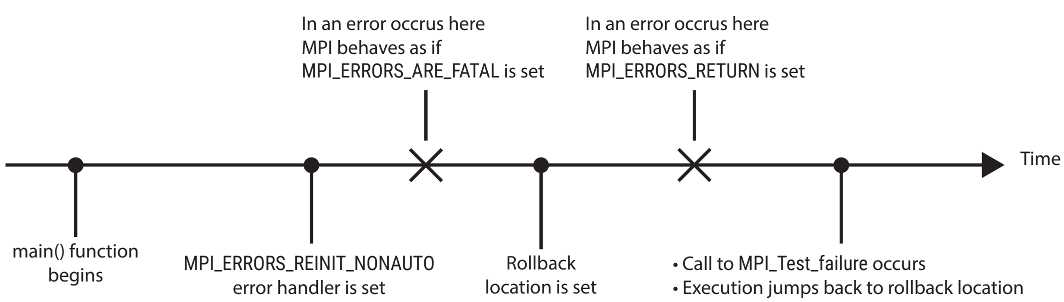

Figure 12.2: Different error scenarios for the MPI_ERRORS_REINIT_NONAUTO error handler.

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.

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 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.

12.6 Failures During Device Code Execution

MPI applications may execute code in hardware devices, such as GPUs, which can suffer from failures. In general, it may not be possible to stop the execution of device code. When MPI causes the program execution to resume at a rollback location when a device code region is being executed, this device code region may not be terminated automatically. The MPI_ERRORS_REINIT_NONAUTO handler along with the MPI_TEST_FAILURE function can be used to enable the program execution to be resumed only when device code is not being executed.

12.7 Examples

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

Example 12.1 Using Reinit with automatic error handling to recover from process 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 Reinit with non-automatic error handling to recover from process 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();
}
}
```