Handling C++ Exceptions in MPI Applications

Jiri Jaros

Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations, Brno, CZ

1 Motivation and Goals

Handling error states in C++ applications is managed by exceptions. In distributed MPI applications, it is often necessary to inform the other processes (ranks), that something wrong happened, and that the application should either recover from the faulty state, or report the error and terminate gracefully. Unfortunately, the MPI standard does not provide any support for distributed error handling.

This poster presents a new approach for exceptions handling in MPI applications. **The goals are to**

(1) report any faulty state to the user in a nicely formatted way by just a single rank,

(2) ensure the application will never deadlock,

(3) propose a simple interface and ensure interoperability with other C/C++ libraries.

2 Minimalistic Interface and Simple Usage

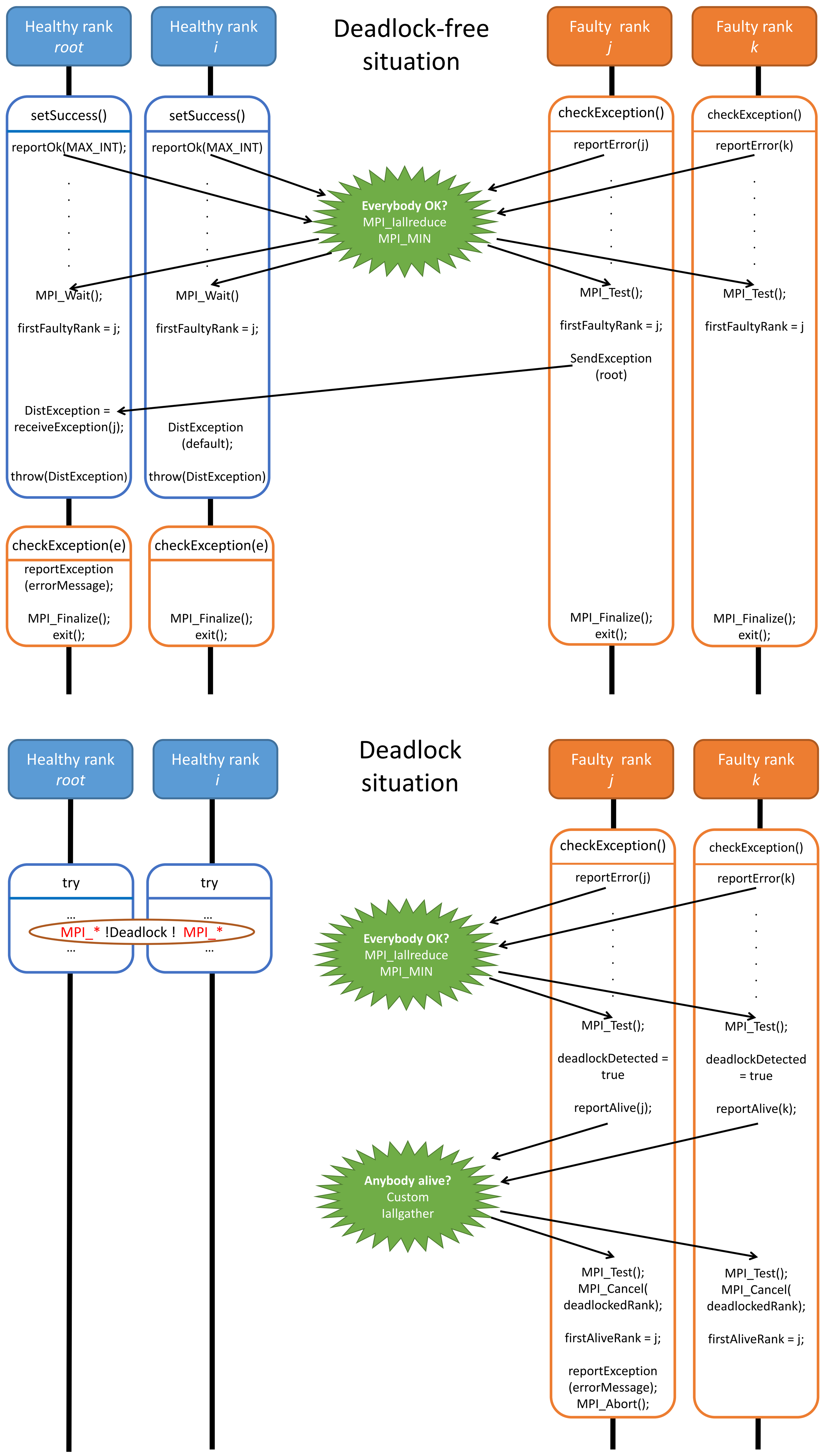
The interface consists of two classes. The `DistException` class wraps all exceptions derived from the base C++ class `std::exception` and maintains information.

The `ErrorChecker` class is used to propagate and report exceptions. The `setSuccess()` method indicates the code has passed a checkpoint at the end of the try block. The `checkException()` method finds the faulty rank, returns exception details and delegated reporting duties to a selected rank.

```cpp
int main(int argc, char** argv)
{
    // Initialize MPI.
    MPI_Init(&argc, &argv);

    // Initialize error checker (static class).
    ErrorChecker::init(MPI_COMM_WORLD, timeout);

    // Protected block of the code.
    try
    {
        // Any combination of local and MPI computation may appear here.
        MPI_Bast(...);
        foo();
        MPI_Barrier(...);
            ...
        // The very last command of the try block sets a success flag.
        ErrorChecker::setSuccess();
    }// end of try
    // Error handling.
    catch (const std::exception& e)
    {
        // Find out whether any remote rank caused an exception and
        // which rank is supposed to print out an error message.
        const auto& distException = ErrorChecker::catchException(e);

        // Check whether code has deadlocked due to a blocking MPI call
        // or collective communication in progress. If so, find the rank
        // which will report the error, otherwise leave it for root.
        const int reportingRank = (distException.getDeadlockMode()
                    ? distException.getRank()) : rootRank;
        if (reportingRank == myRank) reportError(distException);

        // Print out error message and terminate or recover.
        printErrorAndTerminate(distException);
    }// end of catch

    ErrorChecker::finalize();
    terminateApplication(EXIT_SUCCESS);
}// end of main
```

3 Under the Hood

The `ErrorChecker` class creates a side channel for error propagation. If no deadlock is detected, the root rank is informed about the problem and the code can be terminated properly. Otherwise, the remaining alive ranks vote the one to be responsible for reporting error and calling abort.



4 Integration into MPI applications

The proposed method adopts a minimalistic interface. In the simplest case, the user can use only a single try-catch block to manage error reporting in a sensible way. Nevertheless, the user is free to use as many try-catch blocks as necessary.

The advantages of the proposed solution is that no dedicated rank for testing the errors is necessary, a single reduce operation is only required to confirm the application passed a check point, deadlock in application cannot interrupt the error handling, and the application always terminates gracefully with a proper error message. The necessary support for MPI exceptions to handle MPI error states, not part of the default branch, may be seen as a disadvantage, but it can be overcome by custom MPI error handlers.

The proposed solution works well with third party libraries since their exceptions can either be propagated among ranks or the deadlock caused by blocking point-to-point or collective operations can be detected.

Classic error handling in MPI applications

Error handling with prosed code

5 Conclusions

The code was tested under different MPI implementations such as IntelMPI 19.x and OpenMPI 4.x up to 1536 ranks. As external libraries heavily utilizing collective communications, distributed version of the fast Fourier transform (FFTW) and the HDF5 I/O libraries were chosen.

The code was tested with several injected errors into multiple ranks such as non existing input file, disk quota exceeded, wrong rank in the MPI call, and standard system exceptions such as out of memory problems, numerical errors, etc. In all situations the code has worked properly.

In the future work, the code will be extended to protect of various communicators, and allow sophisticated error recovery.

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90140).