Handling C++ Exceptions in MPI Applications

Jiri Jaros*
Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations
Brno, Czech Republic
jarosjir@fit.vutbr.cz

KEYWORDS

MPI, C++ Exceptions, Error Reporting.

ACM Reference Format:
Jiri Jaros. 2021. Handling C++ Exceptions in MPI Applications. In *Proceedings of Supercomputing Conference (SC'21).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

Handling error states in C++ applications is managed by exceptions [1]. This technique protects a piece of code by a `try-catch` code block. Any error arising in the try block is converted into an exception object and resolved in the catch block. If the error is not handled, the application is still properly terminated. Exceptions bring many significant benefits over standard error code checking implemented in the C language, the most important of which is decoupled program logic and error handling.

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. It only defines return codes for MPI calls. The distribution of the error message is then left to the developers. If they are not willing do so after every MPI call, the MPI runtime may be informed that all errors are fatal. This, however, leads to application termination and the error reporting from all ranks, which is not user friendly while running over thousands of ranks.

After dropping the C++ interface in the MPI-3.0 standard [8], the MPI exception interface for reporting the MPI errors was removed from the default branch as well, yet it can still be enabled during MPI compilation. Nevertheless, the MPI exceptions only cover the MPI routines and report to the local rank only. This brings a lot of troubles since any exception thrown by either the MPI runtime or the user code breaks the standard code path and jumps into the exception handler. This may cause several communication routines to be skipped leading to a deadlock. This is even an worse situation than an uncoordinated crash, since the application hangs and keeps consuming computing resources until manually terminated.

Surprisingly, there is very little work published on the error handling inside MPI applications. Most research concerns on solving MPI faulty states such as a node crash, lost interconnection, or detection and recovery of deadlock states [6, 7, 9]. However, what if the MPI itself is working properly but the application encountered an unexpected situation? As far as the author knows, there is only a single paper devoted to exception propagation in MPI applications by Engwer at.al [2]. This work uses a side channel for distributing error states over all ranks and asynchronous error checking. The technique seems to be pretty robust, but has a major drawback. All MPI calls have to be made via a custom interface which makes it unfit for applications using third party libraries.

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.

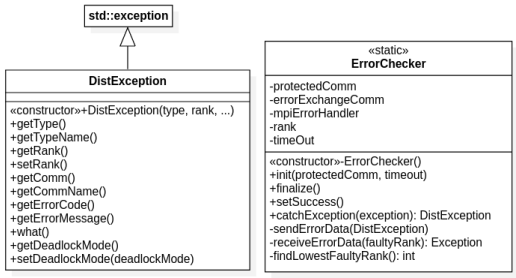

Figure 1: Class diagram of the MPI error checker.

2 PROPOSED METHOD

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 interface consists of two classes, `DistException` and `ErrorChecker`, see Fig. 1. The `DistException` class wraps all exceptions derived from the base C++ class `std::exception`, including `MPI::Exception` and maintains information about the type of the exception, faulty MPI communicator, its name, faulty rank, user error code and error message, and the presence of deadlock.

The `ErrorChecker` class is used to report and handle exceptions. The application errors are reported by instances of `DistException`. In a nutshell, `ErrorChecker` clones the MPI communicator it is supposed to protect (e.g., `MPI_COMM_WORLD`) to build a side channel where the ranks announce to have passed a given checkpoint, usually at the end of the try block, at end of the iteration, or after a long operation the user should be informed about. At these points, the `setSuccess()` method is called to inform the others that no exception was thrown by this rank. If any other rank yet threw an exception, this routine will propagate this information within the local process by throwing a private instance of `DistException`.

To handle exceptions in the catch block, `checkException()` shall be called at the first place to pass on the information about

*corresponding author.

SC'21, 2021, St. Louis, MO, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

local exceptions as well as to collect the error information from the other ranks, and to vote a suitable rank to log the error message (usually the root rank or the first alive). If the code is not deadlocked, the user can try to recover, or terminate the code gracefully otherwise. A code snippet showing the use can be seen in Listing 1.

```
int main(int argc, char** argv)
{
  // Init MPI and set the error handler.
  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 set 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 some blocking call
    // or collective communication in progress. If so, find the rank
    // which will report the error, otherwise leave if for root.
    const int reportingRank = (distException.getDeadlockMode()
                               ? distException.getRank()) : 0;
    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
```

Listing 1: Simple usage of the error checking code

3 IMPLEMENTATION DETAILS

As mentioned previously, `ErrorChecker` creates a side channel to distribute information about exceptions.

The `setSuccess()` method informs all other ranks that no exception was thrown by this rank up to this point. For this purpose, a non-blocking `MPI_Iallreduce` call with an `MPI_MIN` operation is used. Each healthy rank sends the `MAX_INT` value to announce no error. Faulty ranks will, however, skip the `setSuccess()` method, but join the communication in the `catchException()` method and provide its rank instead. Back in `setSuccess()`, all ranks are sitting in `MPI_Wait`, waiting for the result of the reduce operation. If the result is `MAX_INT`, no error happened. Otherwise, the ranks will get the information about the first faulty rank that threw an exception. Although it would be possible to collect all faulty ranks by `MPI_Iallgather`, this is usually not required for reporting purposes. Once the faulty rank is known, the reporting rank asks the faulty one about the exception details using a set of point-to-point nonblocking communications. The exception details are only stored in the reporting rank, the others only know an error happened. This behaviour can be simply changed to an inform-all mode. Once all healthy ranks know about the remote exception, they create a local instance of `DistException` and throw it at the end of `setSuccess()` to join the faulty ranks in the catch block.

The `catchException(const std::exception& e)` first checks the exception type. If the exception was thrown by `setSuccess()` method, it was obviously created by a remote rank and rethrown locally. In that case, the exception details are extracted and returned for local processing. Otherwise, it must be a new local exception thrown somewhere in the try block and has to be propagated to the other ranks. This is done by joining the already mentioned `MPI_Iallreduce` communication. Since the faulty rank left the predescribed code path, it is possible that the other ranks will never join this call (never call `setSuccess()`) since being stuck in another blocking call. This is solved by periodic testing of the `MPI_Iallreduce` communication in `catchException()`. If not finished within a predefined timeout (10s by default), the application has deadlocked and the only solution is to call `MPI_Abort`. However, before that, it is necessary to report the error message and create an error code. Since there might be multiple faulty ranks (e.g., input file is not accessible from one node), it is necessary to find all faulty, but still alive ranks, to agree which one will overtake the reporting and termination responsibility. This is done by probing all ranks within the protected communicator. Each faulty rank sends its rank value to all other ranks and waits for the answer. Simply said, a custom deadlock-free variant of `MPI_Iallgather` communication is performed. After a given timeout, communications with not responding ranks are cancelled, and the reporter is chosen to be the process with the lowest rank. Consequently an exception information is wrapped into an MPI message with the deadlock mode set to true and sent to the reporter.

If deadlock was not detected while running `MPI_Iallreduce`, the faulty rank with the lowest rank value wraps the exception into the error message and sends it to the root rank. In this situation, the code is able to recover.

4 CONCLUSIONS

The proposed exception handling mechanism was integrated into the MPI version of the acoustics k-Wave toolbox [5]. 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 utilising collective communications, distributed version of the fast Fourier transform (FFTW) [4] and the HDF5 [3] 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, custom k-Wave exceptions, and standard system exceptions such as out of memory problems, numerical errors, etc. In all situations the code has worked properly.

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 code can be downloaded from https://github.com/jarosjir/MPIErrorChecker

5 ACKNOWLEDGEMENT

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

REFERENCES

[1] Rodrigo Bonifacio, Fausto Carvalho, Guilherme N. Ramos, Uira Kulesza, and Roberta Coelho. 2015. The use of C++ exception handling constructs: A comprehensive study. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 21–30. https://doi.org/10.1109/SCAM.2015.7335398

[2] Christian Engwer, Mirco Altenbernd, Nils-Arne Dreier, and Dominik Goddeke. 2018. A High-Level C++ Approach to Manage Local Errors, Asynchrony and Faults in an MPI Application. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 714–721. https://doi.org/10.1109/PDP2018.2018.00117 arXiv:1804.04481

[3] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases - AD '11*. ACM Press, New York, New York, USA, 36–47. https://doi.org/10.1145/1966895.1966900

[4] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (feb 2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301

[5] Jiri Jaros, Alistair P. Rendell, and Bradley E. Treeby. 2016. Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications* 30, 2 (may 2016), 137–155. https://doi.org/10.1177/1094342015581024 arXiv:arXiv:1408.4675v1

[6] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. 2016. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications* 30, 3 (aug 2016), 305–319. https://doi.org/10.1177/1094342015623623

[7] Nuria Losada, Patricia González, María J. Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. (may 2020), 467–481 pages. https://doi.org/10.1016/j.future.2020.01.026

[8] Martin Ruefenacht, Derek Schafer, Anthony Skjellum, and Purushotham V. Bangalore. 2021. MPIs Language Bindings are Holding MPI Back. *CoRR* abs/2107.10566 (2021). arXiv:2107.10566 https://arxiv.org/abs/2107.10566

[9] Guang Suo, Yutong Lu, Xiangke Liao, Min Xie, and Hongjia Cao. 2016. NR-MPI: A N on-stop and Fault Resilient MPI SupportingProgrammer Defined Data Backup and Restore for E-scaleSuper Computing Systems. *Supercomputing Frontiers and Innovations* 3, 1 (jun 2016). https://doi.org/10.14529/jsfi160101