Message-Passing and MPI Programming

Error Handling

N.M. Maclaren

Computing Service

nmm1@cam.ac.uk
ext. 34761

July 2010

4.1 Introduction

Most standards get error handling hopelessly wrong; MPI gets it at least half right, and the following is a summary of its approach:

- All invalid uses are defined to be erroneous, and implementations are encouraged to detect them.
- Most undefined results (as distinct from behaviour) are detectable; e.g. they are set to a special, invalid value.
- Most nonsense is defined to be erroneous; e.g. you cannot legally create a deadlock.
- There is **no** concept of conforming but undefined; i.e. there are no valid programs with no known meaning.

C is infested with the concept of conforming but undefined, C++ uses it and even Fortran has it. MPI does have it in a very few places, but only as bugs in the standard.

- The default error handling is to stop; i.e. **not** handling errors is fairly fail-safe in MPI.

However, as you might expect, not everything is sweetness and light:

- Implementations are not required to detect errors; some errors are usually detected, but others rarely are.
- MPI has not specified a debugging mode; error detection is usually at the whim of your implementor.

Errors that can be detected locally usually are detected (e.g. providing an out-of-range process number). Inconsistencies across collectives may be detected – i.e. some implementations will, and others will simply fail horribly.

- Some errors are almost indetectable, and they include most language/MPI interface ones (e.g. incorrect datatype for the buffer type).
- Non-MPI ones are obviously not handled, and they may cause MPI to fail horribly. MPI cannot fix up other standards' defects!

There are programmable error handling facilities, but they do not allow actual recovery. This is essentially unavoidable, because the consequences of an error having occurred are almost unpredictable, and are therefore impossible to reverse. You can use them **only** for cleaning-up, which includes writing your own diagnostics, and even that is not fully reliable.

However, you should always look at the implementation documents, because MPI encourages documented enhancements and this is an area where they are quite likely.

4.2 Simple Error Handling

There are several predefined error handlers. `MPI_ERRORS_ARE_FATAL` is the default and does what it says; MPI produces some diagnostics and stops. `MPI_ERRORS_RETURN` returns an error code, which is the last argument for Fortran, and the function result for C; it is **not** recommended for use in C++. In C++, the recommended alternative is `MPI::ERRORS_THROW_EXCEPTIONS`; do not even **think** of using it in C or Fortran (including in most mixed-language programs) – heaven alone knows what it would do!

You attach the error handling setting to a communicator, which is best set early (i.e. before any serious use of MPI), once only, and consistently across processes. Those restrictions are not required but are simple, and reduce the chances of you forgetting and making an error. The function is one that has changed name (from `MPI_Errhandler_set` to `MPI_Comm_set_errhandler`). Having done that, if an MPI function returns an error code (i.e. anything that is not `MPI_SUCCESS`), call your code to diagnose, clean-up and stop.

Error codes are implementation dependent, but there is a function to map them into an error string, and you should use this for reasonable diagnostics. You do that by calling `MPI_Error_string`, which maps the error code to a textual message of maximum length `MPI_MAX_ERROR_STRING`. This is a block of text and **not** a C string (though it **is** a C string in C++); its length is returned via a separate argument in C (as in Fortran).

Warning: `MPI_ERRORS_RETURN` is **dangerous**. You must test for errors in **all** calls, because one undetected error will cause chaos later.

But the facility can be very useful to handle errors more 'gracefully'. You can write your own, helpful diagnostics in terms that may help you locate the problem. You can flush all your output to files to avoid lost data, and ensure that previous output and diagnostics are not lost. You can tidy up external state and not just crash, though doing that is advanced use.

It can also be used temporarily for debugging, to print out some of your program's data. When using it like that, it is best to set the mode just around the failing call, so as not to risk missing an error return somewhere else.

4.3 Fortran Error Handling

You set it as in the following example:

```
INTEGER :: error

CALL MPI_Comm_set_errhandler ( MPI_COMM_WORLD ,    &
    MPI_ERRORS_RETURN , error )
```

Old versions of *gfortran* do not support this, because there was a bug in its generic resolution handling, so you may get an error message saying that there is no specific subroutine for the generic 'mpi_comm_set_errhandler'. If you do, there is a truly mind-boggling bypass: just set a temporary integer variable to the value `MPI_COMM_WORLD` and use that variable as an argument in the call to `MPI_Comm_set_errhandler`.

And this is how you use it:

```
INTEGER :: error , length , temp
CHARACTER ( LEN = MPI_MAX_ERROR_STRING ) :: message

< call some MPI function >
IF ( error != MPI_SUCCESS ) THEN
    CALL MPI_Error_string ( error , message , length , temp )
    PRINT * , message(1:length)
    CALL MPI_Abort ( MPI_COMM_WORLD , 1 , temp )
END IF
```

4.4 C Error Handling

You set it as in the following example:

```
int error ;

error = MPI_Comm_set_errhandler ( MPI_COMM_WORLD ,
    MPI_ERRORS_RETURN ) ;
```

And this is how you use it:

```
int error ;
char message[MPI_MAX_ERROR_STRING] ;

< call some MPI function >
if ( error != MPI_SUCCESS ) {
    MPI_Error_string ( error , message , & length ) ;
    printf ("%.*s\n") length , message ;
    MPI_Abort ( MPI_COMM_WORLD , 1 ) ;
 }
```

Note the way that the length is returned.

4.5 C++ Error Handling

You set it as in the following example:

```
MPI::COMM_WORLD . Set_errhandler (
    MPI::ERRORS_THROW_EXCEPTIONS ) ;
```

Note that OpenMPI does not support this by default, because there are problems with some C++ compilers (though not *gcc*) and it needs to be configured with an optional setting. That is when **building** it, and not when not using it, so you usually need to contact your administrator if it does not work.. If you are your own administrator, you need to set `--enable-cxx-exceptions` when invoking `configure`.

And this is how you use it:

```
try {
    < use some MPI functions >
} catch ( MPI::Exception failure ) {
    cerr << failure . Get_error_string ( ) << endl ;
    MPI::COMM_WORLD . Abort ( 1 ) ;
}
```

Note that this **does** return a C string, but **not** a `basicstring` instance. If you need the error code, you will need to use `<exception>.Get_error_code()`, because `Exception` is an opaque class.

There is one point that is not specific to MPI, but is worth mentioning. If you really want to shoot your own foot off, write code like the following:

```
try {
    < use some MPI functions >
} catch ( ... ) {
    cerr << "Well, that didn't work - " <<
        "let's try something else" <<  endl ;
    fallback_code () ;    // Or just drop through
}
```

Now try to guess what will happen if the exception was something unexpected!

4.6 Advanced Error Handling

Few people will want to do this, but it is worth knowing what can be done.

You can map error codes into error classes with the function `MPI_Error_class`. Error classes are a documented set of 60 distinct values (of which only about 25 are relevant to this course) with names `MPI_ERR_...`. You can use these to distinguish various types of error, and you may want to do this, for advanced handling; I cannot offhand imagine why, but the facility is there.

You can define your own error handlers – these are simply functions to call when there is an error. Doing this is safer than using `MPI_ERRORS_RETURN`, provided that you code the function carefully. You use the same logic as for `MPI_ERRORS_RETURN` in the examples given above. This course does not cover it, for simplicity, but experienced programmers will have no trouble.

You need to create an error handler (i.e. register the function) before using it, and the function is another that has changed name (from `MPI_Errhandler_create` to `MPI_Comm_create__errhandler`). When it is no longer needed, it should be cleaned up by calling `MPI_Errhandler_free`. There is also a C type name `MPI_Handle_function` (class `MPI::Handle_function` in C++) and a MPI constant `MPI_ERRHANDLER_NULL` (`MPI::ERRHANDLER_NULL` in C++).

You can go beyond that, but it is not recommended; even experts will very rarely need or want to.

Error handling is purely local, and every process can have a different handler – actually, every communicator in every process can! And MPI-2 extends it to some other classes, though it is not always clear which handler will be called if an operation is associated with two objects each of which has its own handler. You can also change it whenever you want; for saving and restoring the old one, you need another function that has changed name from `MPI_Errhandler_get` to `MPI_Comm_Get_errhandler`.

And that is more-or-less all there is to MPI error handling. The only exercise is trying out a simple case of handling errors yourself.